Tina4 for Python Developers

Zero Dependencies, 55 Features, Built for AI

v3.12.4 • tina4.com

The Intelligent Native Application 4ramework

Table of Contents

Getting Started with Tina4 Python 58

1. What Is Tina4 Python . . . 58

2. Prerequisites and Installation . . . 58

What You Need . . . 58

Installing the Tina4 CLI . . . 59

Creating a New Project . . . 59

Starting the Dev Server . . . 60

3. Project Structure Walkthrough . . . 60

4. Your First Route . . . 61

Test It . . . 62

Understanding What Happened . . . 62

Adding More HTTP Methods . . . 62

5. Your First Template . . . 63

Create a Base Layout . . . 63

Create a Product Listing Page . . . 64

Create the Route That Renders the Template . . . 64

See It in the Browser . . . 65

How Template Rendering Works . . . 65

About tina4css . . . 66

6. Understanding .env . . . 66

7. The Dev Dashboard . . . 67

8. Manual Setup (No CLI) . . . 68

Step 1: Install the Package . . . 68

Step 2: Create app.py . . . 68

Step 3: Create the Folder Structure . . . 68
Step 4: Create .env . . . 68
Step 5: Run It . . . 68
9. Request & Response Fundamentals . . . 69
Reading Query Parameters . . . 69
Reading URL Path Parameters . . . 69
Reading the Request Body . . . 69
Reading Headers . . . 70
Sending JSON Responses . . . 70
Sending HTML / Template Responses . . . 70
Status Codes . . . 70
Worked Example: A Complete Route File . . . 70
10. Exercise: Greeting API + Product List Template . . . 72
Exercise Part A: Greeting API . . . 72
Exercise Part B: Product List Page . . . 72
11. Solutions . . . 73
Solution A: Greeting API . . . 73
Solution B: Product List Page . . . 74
12. Gotchas . . . 75
1. File not auto-discovered . . . 75
2. Missing import . . . 76
3. Handler not async . . . 76
4. Template not found . . . 76
5. Port already in use . . . 76
6. Changes not reflected . . . 76
7. .env not loaded . . . 76

1. How Routing Works in Tina4 . . . 78
2. HTTP Methods . . . 78
3. Path Parameters . . . 79
Typed Parameters . . . 79
Typed Parameters in Action . . . 80
4. Query Parameters . . . 81
5. Route Groups . . . 81
6. Middleware on Routes . . . 83
Middleware on a Single Route . . . 83
Blocking Middleware . . . 83
Multiple Middleware . . . 84
7. Route Decorators: @noauth() and @secured() . . . 84
@noauth() -- Public Routes . . . 84
@secured() -- Protected GET Routes . . . 84
8. Route Chaining: .secure() and .cache() . . . 85
.secure() . . . 85
.cache() . . . 85
Chaining Both . . . 85
9. Wildcard and Catch-All Routes . . . 85
Wildcard Routes . . . 85
Catch-All Route (Custom 404) . . . 86
10. Route Listing via CLI . . . 86
11. Organizing Route Files . . . 87
Pattern 1: One File Per Resource . . . 87
Pattern 2: Subdirectories by Feature . . . 87
12. Exercise: Build a Full CRUD API for Products . . . 88
Requirements . . . 88

13. Solution . . . 89

14. Gotchas . . . 91

1. Trailing slashes matter . . . 91

2. Parameter names must be unique in a path . . . 91

3. Method conflicts . . . 91

4. Route handler must return a response . . . 91

5. Decorator order matters . . . 92

6. Forgetting async def . . . 92

7. Group prefix must start with a slash . . . 92

Request and Response **93**

1. The Two Objects You Use Everywhere . . . 93

2. The Request Object . . . 93

request.method . . . 93

request.path . . . 93

Path Parameters (Function Arguments) . . . 93

request.params . . . 94

request.body . . . 94

request.headers . . . 94

request.ip . . . 95

request.cookies . . . 95

request.files . . . 95

Handling Multiple File Uploads . . . 96

Upload Size Limits . . . 97

3. The Response Object . . . 97

response.json() . . . 97

response.html() . . . 97

response.text() . . . 98

response.render() . . . 98
response.redirect() . . . 98
response.file() . . . 98
response.xml() . . . 99
response.error() . . . 99
4. Status Codes . . . 99
5. Content Negotiation . . . 101
6. Custom Headers and Cookies . . . 101
Setting Response Headers . . . 101
Setting Cookies . . . 102
7. Input Validation . . . 103
The Validator Class . . . 103
8. Real-World Example: File Upload with Validation . . . 103
9. Exercise: Build a Contact Form API . . . 105
Requirements . . . 105
Test with: . . . 105
10. Solution . . . 106
11. Gotchas . . . 107
1. request.body is None for GET requests . . . 107
2. Forgetting the Content-Type header . . . 108
3. File upload with wrong field name . . . 108
4. response.redirect() in AJAX calls . . . 108
5. Cookie not being set . . . 108
6. Large file uploads fail . . . 108
7. Chaining response methods in wrong order . . . 109
8. Header names are lowercase . . . 109

1. Beyond JSON -- Rendering HTML . . . 110
2. The @template Decorator . . . 110
3. Variables and Output . . . 111
Basic Output . . . 111
Accessing Nested Properties . . . 111
Method Calls and Slicing . . . 111
Auto-Escaping . . . 111
4. Filters . . . 112
Complete Filter Reference . . . 112
String Filters . . . 112
Array Filters . . . 113
Encoding Filters . . . 114
Numeric Filters . . . 114
JSON Filters . . . 114
Dict Filters . . . 115
Other Filters . . . 115
Chaining Filters . . . 115
The default Filter . . . 115
Dumping Values for Debugging . . . 116
5. Control Tags . . . 116
if / elif / else . . . 116
for Loops . . . 117
for / else . . . 117
set -- Local Variables . . . 117
6. Template Inheritance . . . 118
Base Template . . . 118
Child Template . . . 118

Calling Parent Blocks . . . 119
7. Include and Macro . . . 119
include . . . 119
macro -- Reusable Template Functions . . . 119
8. Comments . . . 120
9. Special Tags . . . 120
{% raw %} -- Literal Output . . . 120
{% spaceless %} -- Remove Whitespace . . . 120
{% autoescape %} -- Control Escaping . . . 121
Whitespace Control . . . 121
10. tina4css . . . 121
Layout Classes . . . 121
Buttons . . . 122
Cards . . . 122
Alerts . . . 123
Forms . . . 123
Tables . . . 123
Spacing and Typography Utilities . . . 124
11. Exercise: Build a Product Catalog Page . . . 124
Requirements . . . 124
12. Solution . . . 125
13. Gotchas . . . 127
1. Whitespace in output . . . 127
2. Variable not defined error . . . 127
3. Extends must be the first tag . . . 127
4. Macro not found . . . 127
5. Filter produces wrong type . . . 128

6. Escaped HTML when you want raw output . . . 128
7. Include file path wrong . . . 128
Database 129
1. From Lists to Real Data . . . 129
2. Configuration . . . 129
TINA4DATABASEURL . . . 129
Installing Database Drivers . . . 130
3. Creating a Connection . . . 130
Connection Pooling . . . 130
Extra Driver Options via **kwargs . . . 130
Firebird URL Forms . . . 131
Firebird Dual-Driver Support . . . 131
Lowercase Column Names . . . 131
Firebird Migration Support . . . 131
4. Running Queries . . . 132
fetch -- Get Multiple Rows . . . 132
DatabaseResult . . . 132
Properties . . . 132
Iteration . . . 132
Index Access . . . 132
Countable . . . 132
Conversion Methods . . . 133
Schema Metadata with column_info() . . . 133
fetch_one -- Get a Single Row . . . 133
execute -- Insert, Update, Delete . . . 134
5. Parameterised Queries . . . 134
6. Transactions . . . 134

7. Batch Operations with execute_many . . . 135
8. Insert, Update, and Delete Helpers . . . 136
insert . . . 136
update . . . 136
delete . . . 136
9. Schema Inspection . . . 137
get_tables() . . . 137
get_columns() . . . 137
table_exists() . . . 137
getdatabasetype() . . . 137
A Schema Info Endpoint . . . 137
10. Migrations . . . 138
File Naming . . . 138
Generating a Migration . . . 138
Writing the Up Migration . . . 138
Writing the Down Migration . . . 139
Running Migrations . . . 139
Checking Status . . . 139
Rolling Back . . . 139
Tracking Table . . . 139
Advanced SQL Splitting . . . 140
Migration Best Practices . . . 140
11. Query Caching . . . 140
12. Exercise: Build a Notes App API . . . 141
Requirements . . . 141
Test with: . . . 142
13. Solution . . . 142

Migration . . . 142
Routes . . . 143
14. Seeder -- Generating Test Data . . . 145
FakeData . . . 145
Deterministic Output . . . 146
Seeding a Table . . . 146
Overrides . . . 146
When to Use It . . . 146
15. Gotchas . . . 147
1. SQLite boolean quirk . . . 147
2. lastinsertrowid() is SQLite-specific . . . 147
3. String vs integer comparison . . . 147
4. Connection not closed . . . 147
5. Migration order matters . . . 147
6. Missing down migration . . . 148
7. Failed migrations blocking progress . . . 148
8. SQL injection through string formatting . . . 148
ORM **149**
1. From SQL to Objects . . . 149
ORM at a Glance: Four Languages, One Shape . . . 149
Defining a Model . . . 149
Common Query Operations . . . 150
2. Defining a Model . . . 151
Field Types . . . 152
Field Options . . . 152
Field Mapping . . . 153
getdbcolumn and getdbdata . . . 153

find() vs where() -- naming convention . . . 154
auto_map and Case Conversion Utilities . . . 154
3. create_table -- Schema from Models . . . 154
4. CRUD Operations . . . 155
save -- Create or Update . . . 155
create -- Build and Save in One Step . . . 155
findbyid -- Fetch One Record by Primary Key . . . 155
find -- Query by Filter Dict . . . 156
where -- Query with SQL Conditions . . . 156
delete -- Remove a Record . . . 156
Listing Records . . . 156
select_one -- Fetch a Single Record by SQL . . . 157
load -- Populate an Existing Instance . . . 157
count -- Count Records . . . 157
5. todict, tojson, and Other Serialisation . . . 157
to_dict . . . 157
to_json . . . 157
Other Serialisation Methods . . . 158
6. Relationships . . . 158
ForeignKeyField — Auto-Wired Relationships . . . 158
has_many . . . 159
has_one . . . 160
belongs_to . . . 160
7. Eager Loading . . . 160
Declarative Relationships with Descriptors . . . 161
Nested Eager Loading . . . 162
to_dict with Nested Includes . . . 162

8. Soft Delete . . . 162
Deleting and Restoring . . . 163
Including Soft-Deleted Records . . . 163
Counting with Soft Delete . . . 163
When to Use Soft Delete . . . 163
9. Auto-CRUD . . . 163
The auto_crud Flag . . . 164
Manual Registration . . . 164
Auto-Discovering Models . . . 165
What the Generated Routes Do . . . 165
Custom Routes Alongside Auto-CRUD . . . 165
Introspection . . . 165
10. Cached Queries . . . 166
11. Scopes . . . 166
12. Input Validation . . . 167
13. Exercise: Build a Blog with Relationships . . . 167
Requirements . . . 168
14. Solution . . . 168
15. Gotchas . . . 171
1. Forgetting to call save() . . . 171
2. findbyid() returns None . . . 171
3. find() vs findbyid() . . . 171
4. Circular imports with relationships . . . 171
5. to_dict() includes everything . . . 172
6. Validation only runs on validate() . . . 172
7. Foreign key not enforced . . . 172
8. N+1 query problem . . . 172

9. Auto-CRUD endpoint conflicts . . . 173
10. Soft-deleted records appearing in queries . . . 173
QueryBuilder Integration . . . 173
QueryBuilder **174**
1. The Factory: from_table() . . . 174
2. Choosing Columns: select() . . . 174
3. Filtering: where() and or_where() . . . 175
OR conditions . . . 175
Conditions without parameters . . . 175
4. Joins: join() and left_join() . . . 175
5. Aggregation: group_by() and having() . . . 176
Filtering groups with having() . . . 176
6. Sorting: order_by() . . . 176
7. Pagination: limit() . . . 177
Pagination pattern . . . 177
8. Inspecting the SQL: to_sql() . . . 178
9. Execution Methods . . . 178
get() — Multiple rows . . . 178
first() — Single row . . . 178
count() — Row count . . . 179
exists() — Boolean check . . . 179
10. Chaining — Building Complex Queries . . . 179
11. Using with ORM Models . . . 180
When to use QueryBuilder vs ORM . . . 180
NoSQL: MongoDB Queries . . . 181
Operator Mapping . . . 181
Example . . . 181

Gotchas . . . 182
1. "select() replaced my columns" . . . 182
2. "My LIMIT is not in to_sql() output" . . . 182
3. "count() returns 0 but get() returns rows" . . . 182
4. "RuntimeError: No database connection provided" . . . 182
5. "or_where() on the first condition has no effect" . . . 183
6. "Parameters are in the wrong order" . . . 183
Exercise: Product Search API . . . 183
Solution . . . 184
Authentication **186**
1. Locking the Door . . . 186
2. JWT Tokens . . . 186
Generating a Token . . . 186
Token Expiry . . . 186
Validating a Token . . . 187
Reading the Payload . . . 187
The Secret Key and Algorithm . . . 187
3. Password Hashing . . . 187
Hashing a Password . . . 187
Checking a Password . . . 188
Registration Example . . . 188
4. The Login Flow . . . 189
5. Using Tokens in Requests . . . 190
6. Protecting Routes . . . 190
Auth Middleware . . . 190
Applying Middleware to Routes . . . 190
Applying Middleware to a Group . . . 191

7. @noauth and @secured Decorators . . . 191
@noauth -- Skip Authentication . . . 191
@secured -- Require Authentication for GET Routes . . . 192
Role-Based Authorization . . . 192
8. CSRF Protection . . . 192
Generating a Token . . . 192
Validating the Token . . . 193
When to Use CSRF Tokens . . . 193
9. Sessions . . . 194
Session Configuration . . . 194
Using Sessions . . . 194
Session Options . . . 195
10. Exercise: Build Login, Register, and Profile . . . 195
Requirements . . . 195
Test with: . . . 196
11. Solution . . . 196
Migration . . . 196
Routes . . . 196
12. Gotchas . . . 200
1. Token expiry confusion . . . 200
2. Secret key management . . . 200
3. CORS with authentication . . . 200
4. Storing tokens in localStorage . . . 200
5. Forgetting @noauth on login . . . 201
6. Password hash column too short . . . 201
7. Token in URL query parameters . . . 201

1. Remembering Your Users . . . 202
2. Session Configuration . . . 202
Available Backends . . . 202
Redis Configuration . . . 202
MongoDB Configuration . . . 203
Valkey Configuration . . . 203
Database Sessions . . . 203
Session Lifetime . . . 203
Session Cookie . . . 203
3. The Session API . . . 203
Full API Reference . . . 204
4. Reading and Writing Session Data . . . 204
Writing to the Session . . . 204
Reading from the Session . . . 204
Deleting Session Data . . . 205
Checking if a Key Exists . . . 205
Getting All Session Data . . . 205
5. Flash Messages . . . 205
Setting a Flash Message . . . 205
Reading a Flash Message . . . 205
Flash Messages in Templates . . . 206
6. Cookies . . . 206
Setting a Cookie . . . 206
Reading a Cookie . . . 206
Deleting a Cookie . . . 206
Cookie Options . . . 207
When to Use Cookies vs Sessions . . . 207

7. Remember Me . . . 208
8. Session Security . . . 209
Regenerate Session ID After Login . . . 209
Destroy a Session . . . 209
Secure Cookie Settings . . . 209
9. Exercise: Build a Shopping Cart . . . 210
Requirements . . . 210
Test with: . . . 211
10. Solution . . . 211
11. Gotchas . . . 214
1. Session lost between requests . . . 214
2. Session data is not JSON-serializable . . . 214
3. Cookie not sent on cross-origin requests . . . 214
4. File-based sessions on multi-server deployments . . . 214
5. Session cookie overwritten by another app . . . 215
6. Secure cookies on localhost . . . 215
7. Flash messages shown twice . . . 215
Middleware **216**
1. The Pipeline Pattern . . . 216
2. Built-In Middleware . . . 216
CorsMiddleware . . . 216
RateLimiter . . . 217
3. Writing Custom Middleware . . . 217
Function-Based Middleware . . . 217
Class-Based Middleware . . . 218
Example: Request Timer (Class-Based) . . . 218
Example: Request Logger (Function-Based) . . . 219

Example: Security Headers (Class-Based) . . . 219
Example: JSON Content-Type Enforcer (Function-Based) . . . 219
Example: Request ID (Class-Based) . . . 219
Example: Input Sanitization (Class-Based) . . . 220
4. The @middleware Decorator . . . 220
5. Middleware on Route Groups . . . 221
6. Execution Order . . . 221
7. Short-Circuiting . . . 222
Conditional Short-Circuit . . . 223
8. Modifying Request and Response . . . 223
Adding Data to the Request . . . 223
Modifying the Response . . . 224
9. Real-World Example: JWT Authentication Middleware . . . 224
10. Real-World Example: API Key Middleware with Database Lookup . . . 225
11. Exercise: Build an API Key Middleware System . . . 225
Requirements . . . 225
Test with: . . . 226
12. Solution . . . 226
Migration . . . 226
Routes . . . 227
13. Gotchas . . . 229
1. Forgetting to await next_handler . . . 229
2. Middleware modifies response after it is sent . . . 229
3. Middleware applied to wrong routes . . . 229
4. Middleware execution order surprises . . . 229
5. Error in middleware breaks the chain . . . 229
6. Database connections in middleware . . . 230

7. Modifying request in middleware does not persist . . . 230

Security **230**

1. Secure-by-Default Routing . . . 230

Making a Write Route Public . . . 231

Protecting a GET Route . . . 231

The Rule . . . 231

2. CSRF Protection . . . 232

How It Works . . . 232

The Template . . . 232

The Middleware . . . 232

AJAX Requests . . . 232

Tokens in Query Strings — Blocked . . . 233

Skipping CSRF Validation . . . 233

Disabling CSRF Globally . . . 233

3. Session-Bound Tokens . . . 233

How Stolen Tokens Fail . . . 233

4. Security Headers . . . 234

HSTS — Enforcing HTTPS . . . 234

Customising Headers . . . 234

5. SameSite Cookies . . . 235

6. Login Flow — Complete Example . . . 235

The Login Route . . . 236

The Login Form . . . 237

Protected Pages — Checking the Session . . . 237

Logout — Destroying the Session . . . 238

7. Handling Expired Sessions . . . 238

The Pattern: Redirect to Login, Then Back . . . 238

Token Refresh . . . 239
8. Rate Limiting . . . 239
9. CORS and Credentials . . . 240
10. Security Checklist . . . 240
Gotchas . . . 241
1. "My POST route returns 401 but I didn't add auth" . . . 241
2. "CSRF validation fails on AJAX requests" . . . 241
3. "I disabled CSRF but forms still fail" . . . 241
4. "My Content-Security-Policy blocks inline scripts" . . . 241
5. "User stays logged in after session expires" . . . 241
Exercise: Secure Contact Form . . . 241
Solution . . . 242
Caching **244**
1. From 800ms to 3ms . . . 244
2. Response Caching with ResponseCache Middleware . . . 244
Cache Headers . . . 245
Caching with Query Parameters . . . 245
What Not to Cache . . . 245
3. Memory Cache (Default) . . . 246
4. Redis Cache . . . 246
Why Redis? . . . 246
5. File Cache . . . 247
When to Use File Cache . . . 247
6. Direct Cache API . . . 247
cache_set . . . 247
cache_get . . . 247

cache_delete . . . 248
Real-World Pattern: Cache-Aside . . . 248
7. Database Query Caching . . . 249
When to Use DB Cache . . . 249
When Not to Use DB Cache . . . 249
Skipping Cache for Specific Queries . . . 249
8. Cache Invalidation Strategies . . . 250
Strategy 1: Time-Based Expiry (TTL) . . . 250
Strategy 2: Event-Based Invalidation . . . 250
Strategy 3: Write-Through Cache . . . 250
9. TTL Management . . . 251
Dynamic TTL . . . 251
10. Cache Statistics . . . 252
11. Combining Cache Layers . . . 252
12. Exercise: Cache an Expensive Product Listing Endpoint . . . 253
Requirements . . . 253
Test with: . . . 254
13. Solution . . . 254
14. Gotchas . . . 257
1. Caching Authenticated Responses . . . 257
2. Cache Stampede . . . 257
3. Memory Cache Lost on Restart . . . 257
4. Stale Data After Database Update . . . 257
5. Cache Key Collisions . . . 257
6. Serialization Overhead . . . 258
7. Forgetting to Set TTL . . . 258

1. Not Everything Should Happen Right Now . . . 259
2. Queue Configuration . . . 259
Default (File Queue) . . . 259
Switching to RabbitMQ . . . 259
Switching to Kafka . . . 259
Switching to MongoDB . . . 260
3. Creating a Queue and Pushing Messages . . . 260
Convenience Method: produce . . . 260
Queue Size . . . 260
4. Consuming Messages . . . 261
The consume Pattern . . . 261
Manual Pop . . . 261
5. Job Lifecycle . . . 261
Job Methods . . . 261
6. Retry and Dead Letters . . . 262
Max Retries . . . 262
Retrying Failed Jobs . . . 262
Dead Letters . . . 262
Purging Jobs . . . 262
7. Queue in Route Handlers . . . 262
8. Switching Backends via .env . . . 263
Development: File (default) . . . 263
Production: RabbitMQ . . . 263
High-Scale Production: Kafka . . . 263
Production: MongoDB . . . 263
9. Produce and Consume Across Topics . . . 264
10. Exercise: Build an Email Queue . . . 264

Requirements . . . 264
Test with: . . . 265
11. Solution . . . 265
12. Gotchas . . . 266
1. Always call complete or fail . . . 266
2. Worker not picking up messages . . . 267
3. Payload too large . . . 267
4. Dead letters pile up . . . 267
5. File backend for production . . . 267
6. Environment-specific topic collision . . . 267
Events **268**
1. Decouple Everything with Events . . . 268
2. Registering Handlers with @on . . . 268
3. Firing Events with emit . . . 268
4. One-Time Handlers with @once . . . 269
5. Removing Handlers with off . . . 269
6. Priority Ordering . . . 269
7. Async Event Support with emit_async . . . 270
8. Real-World Pattern: Order Pipeline . . . 271
9. Exercise: User Lifecycle Events . . . 272
Requirements . . . 272
Test with: . . . 272
10. Solution . . . 272
11. Gotchas . . . 273
1. Emitting from a sync context with async handlers . . . 273
2. Handler order is non-deterministic at equal priority . . . 274
3. Forgetting once fires exactly once . . . 274

4. Mutating the payload in a handler . . . 274

Localization **275**

1. One App, Many Languages . . . 275
2. Locale Files . . . 275
3. The t() Function . . . 276
4. The I18n Class . . . 276
5. Setting the Locale at Runtime . . . 277
    From a Query Parameter . . . 277
    From the Accept-Language Header . . . 277
    From a Session or Cookie . . . 278
6. The TINA4_LOCALE Environment Variable . . . 278
7. Interpolation with {placeholder} . . . 278
8. Fallback Behaviour . . . 279
9. Translating Templates . . . 279
10. Exercise: Multi-Language API Responses . . . 280
    Requirements . . . 280
    Test with: . . . 280
11. Solution . . . 280
12. Gotchas . . . 282
    1. Missing locale file raises an error . . . 282
    2. Placeholder name mismatch . . . 282
    3. TINA4_LOCALE not set, t() uses "en" . . . 282
    4. Locale files not reloading during development . . . 283

Structured Logging **284**

1. Stop Using print() . . . 284
2. Basic Usage . . . 284

3. Log Levels . . . 284
4. The TINA4LOGLEVEL Environment Variable . . . 285
5. Logging in Route Handlers . . . 285
6. File Output and Log Rotation . . . 286
7. Logging Exceptions . . . 286
8. Request Logging Middleware . . . 286
9. Log Levels by Environment . . . 287
10. Exercise: Add Logging to an Order API . . . 287
Requirements . . . 288
Test with: . . . 288
11. Solution . . . 288
12. Gotchas . . . 290
1. DEBUG logs flooding production . . . 290
2. Logging sensitive data . . . 290
3. Log file grows without rotation . . . 290
4. Forgetting to log exceptions . . . 290
Email with Messenger **291**
1. Every App Sends Email . . . 291
2. Messenger Configuration via .env . . . 291
Common Provider Configurations . . . 292
3. Constructor Override Pattern . . . 292
4. Sending Plain Text Email . . . 293
The send() Method Signature . . . 293
5. Sending HTML Email with Text Fallback . . . 293
6. Adding Attachments . . . 294
Attachments with Custom Names and Binary Content . . . 295

7. CC, BCC, and Reply-To . . . 295
8. Custom Headers . . . 296
Default Headers on an Instance . . . 296
Per-Email Headers . . . 296
9. Reading Inbox via IMAP . . . 296
Listing Inbox Messages . . . 297
Reading a Specific Message . . . 298
Searching Messages . . . 298
Other IMAP Operations . . . 299
Testing IMAP Connectivity . . . 299
10. Dev Mode: Email Interception . . . 299
Disabling Interception . . . 299
11. Using Templates for Email Content . . . 300
Rendering and Sending . . . 301
12. Sending Email via Queues . . . 302
13. Exercise: Build a Contact Form with Email Notification . . . 303
Requirements . . . 303
Test with: . . . 303
14. Solution . . . 303
15. Gotchas . . . 307
1. Gmail Blocks "Less Secure" Apps . . . 307
2. Emails Go to Spam . . . 307
3. HTML Email Looks Broken . . . 307
4. Attachment File Not Found . . . 307
5. Dev Mode Silently Intercepts Emails . . . 307
6. Email Template Variables Not Substituted . . . 308
7. Connection Timeout on Send . . . 308

8. IMAP Host Not Configured . . . 308

Frontend with tina4css **309**

1. The Problem with Frontend Toolchains . . . 309

2. What Ships with Tina4 . . . 309

3. The Grid System . . . 309

4. Components . . . 310

Navbar . . . 310

Cards . . . 310

Buttons . . . 311

Tables . . . 311

Alerts . . . 311

Modals . . . 312

Progress Bars . . . 312

Badges . . . 312

Forms . . . 313

5. SCSS Customization . . . 313

Live SCSS Compilation . . . 314

6. frond.js -- The JavaScript Helper . . . 314

AJAX Requests . . . 314

Form Submission via AJAX . . . 315

Token Management . . . 315

Loading Indicators . . . 315

WebSocket (Covered in Chapter 23) . . . 316

7. JavaScript API Reference . . . 316

Including the Scripts . . . 316

7.1 tina4.min.js -- Core Utilities . . . 316

loadPage(url, targetId) . . . 316

saveForm(formId, url, method) . . . 316
sendRequest(url, method, data, callback) . . . 317
7.2 frond.min.js -- Template Engine Client-Side Helpers . . . 317
AJAX Form Handling . . . 317
WebSocket Auto-Reconnect . . . 317
Token Refresh . . . 317
Dynamic Template Loading . . . 317
7.3 tina4js.min.js -- Reactive Frontend Framework . . . 318
Reactive State: signal(), computed(), effect() . . . 318
DOM Rendering: html Tagged Template . . . 318
Web Components: Tina4Element . . . 318
Fetch Wrapper: api() . . . 319
WebSocket Client: ws() . . . 319
Client-Side Routing: route(), navigate() . . . 319
8. Building an Admin Dashboard . . . 319
Base Template . . . 319
Dashboard Page . . . 320
Dashboard Route . . . 322
9. Dark Mode . . . 323
Respecting System Preference . . . 323
10. Responsive Design . . . 323
Responsive Sidebar . . . 323
Responsive Tables . . . 324
Hiding Elements by Screen Size . . . 324
11. Building a Users Page with AJAX . . . 324
The Template . . . 324
The Route . . . 326

12. HTML Builder -- Generating HTML in Code . . . 326
Direct Construction . . . 326
Builder Pattern . . . 327
Helper Functions . . . 327
Void Tags and Auto-Escaping . . . 327
Use Case -- HTML from a Route . . . 328
13. Exercise: Build an Admin Dashboard . . . 328
Requirements . . . 328
Test by: . . . 328
14. Solution . . . 329
15. Gotchas . . . 329
1. CSS Not Loading . . . 329
2. frond.js Functions Not Found . . . 330
3. Dark Mode Flickers on Page Load . . . 330
4. SCSS Not Compiling . . . 330
5. Modal Does Not Open . . . 330
6. Grid Columns Do Not Stack on Mobile . . . 330
7. Static Files Return 404 . . . 331
8. Form Data Not Reaching the Server . . . 331
9. Loading Indicator Never Disappears . . . 331
Testing **332**
1. Why Tests Matter More Than You Think . . . 332
2. Your First Test . . . 332
How It Works . . . 333
3. Assertion Methods . . . 333
assert_equal(actual, expected, message) . . . 333
assert_true(value, message) . . . 333

assert_false(value, message) . . . 333
assertraises(callable, exceptionclass, message) . . . 334
assertnotequal(actual, expected, message) . . . 334
assert_none(value, message) . . . 334
assertnotnone(value, message) . . . 334
4. Testing ORM Models . . . 334
Test Database . . . 336
5. Testing Routes . . . 336
Test Client Methods . . . 337
Response Object . . . 338
6. Testing Authentication . . . 338
7. Setup and Teardown . . . 339
8. Running Tests . . . 340
Run All Tests . . . 340
Run a Specific Test File . . . 340
Run a Specific Test Method . . . 340
Verbose Output . . . 340
Failed Test Output . . . 340
9. pytest Integration . . . 341
Code Coverage . . . 341
10. Testing Best Practices . . . 342
Test One Thing Per Test . . . 342
Use Descriptive Assertion Messages . . . 342
Isolate Tests . . . 342
11. Exercise: Test a User Model and Authentication Flow . . . 343
Requirements . . . 343
Expected output: . . . 343

12. Solution . . . 344
tests/testusermodel.py . . . 344
tests/testauthflow.py . . . 346
13. Gotchas . . . 347
1. Tests Run in Order . . . 347
2. Test Database vs Development Database . . . 348
3. Unique Constraint Failures . . . 348
4. Test Method Not Discovered . . . 348
5. Assertion Arguments Reversed . . . 348
6. Cannot Test Routes That Require Authentication Setup . . . 348
7. Tests Pass Locally but Fail in CI . . . 349
Scaffolding 350
The CRUD Generator . . . 350
What Each File Contains . . . 350
Run It . . . 352
Individual Generators . . . 352
Model . . . 352
Route . . . 353
Migration . . . 353
Middleware . . . 353
Test . . . 353
Form . . . 353
View . . . 353
CRUD . . . 353
Auth . . . 354
AutoCRUD . . . 354
Usage . . . 354

The Auth Generator . . . 354
Field Types . . . 355
Table Naming Convention . . . 355
Combining Generators . . . 356
Exercise: Scaffold a Blog . . . 356
Gotchas . . . 357
Generators Do Not Overwrite . . . 357
Run Migrate After Generate . . . 357
File Naming Matters . . . 357
Field Changes Need New Migrations . . . 357
Singular Table Names . . . 357
Swagger / OpenAPI **358**
1. The 47-Endpoint Problem . . . 358
2. What Swagger/OpenAPI Is . . . 358
3. Accessing the Swagger UI . . . 358
4. Documenting Routes with Decorators . . . 359
@description -- Describe What an Endpoint Does . . . 359
@tags -- Organize Endpoints into Groups . . . 360
@example -- Document Request Body . . . 360
@example_response -- Document Response Body . . . 361
5. Documenting Path Parameters . . . 361
Documenting Query Parameters . . . 362
6. Authentication in Swagger . . . 362
7. Try-It-Out from the Swagger UI . . . 362
Authentication in Try-It-Out . . . 363
8. Customizing the Swagger Info Block . . . 363
9. Generating Client SDKs from the Spec . . . 364

Using OpenAPI Generator . . . 364
Other Generators . . . 365
10. Complete API Documentation Example . . . 365
11. Swagger Configuration . . . 367
12. Exercise: Document a User API . . . 367
Requirements . . . 367
Expected Result . . . 367
13. Solution . . . 368
14. Gotchas . . . 370
1. Swagger not showing up . . . 370
2. Route appears but has no documentation . . . 370
3. Decorator order causes issues . . . 370
4. Example does not match actual response . . . 370
5. Sensitive data in examples . . . 370
6. Swagger in production . . . 371
7. Too many tags . . . 371
8. SDK generation produces incorrect types . . . 371
API Client **372**
1. Calling External APIs . . . 372
2. Basic Requests . . . 372
3. Reading the Response . . . 373
4. Authentication Headers . . . 373
Bearer Token . . . 373
Basic Authentication . . . 373
Custom Headers . . . 374
Mixing Auth and Custom Headers . . . 374

5. SSL Verification and Timeouts . . . 374
6. Sending Query Parameters . . . 374
7. Real-World Patterns . . . 375
Payment Gateway . . . 375
Weather Service . . . 376
8. Retry Logic . . . 376
9. Exercise: Weather Dashboard API . . . 377
Requirements . . . 377
Test with: . . . 377
10. Solution . . . 377
11. Gotchas . . . 379
1. Not checking result["error"] . . . 379
2. Using verify_ssl=False in production . . . 379
3. No timeout set for slow external APIs . . . 379
4. Hardcoding API keys . . . 379
GraphQL and SOAP **380**
1. The Problem GraphQL Solves . . . 380
2. GraphQL vs REST -- A Quick Comparison . . . 380
3. Enabling GraphQL . . . 381
4. Defining a Schema . . . 381
5. Writing Resolvers . . . 382
Testing the Queries . . . 382
6. Mutations . . . 383
Mutation Resolvers . . . 384
Testing Mutations . . . 385
7. Nested Types and Relationships . . . 386
8. Auto-Generating Schema from ORM Models . . . 388

9. The GraphiQL Playground . . . 388
10. Query Variables . . . 389
11. Exercise: Build a GraphQL API for a Blog . . . 389
Requirements . . . 389
Test with these queries: . . . 390
12. Solution . . . 390
Schema . . . 390
Resolvers . . . 391
13. Input Validation . . . 393
What Gets Validated . . . 393
Example . . . 393
14. Field-Level Auth Directives . . . 393
Available Directives . . . 393
Passing Auth Context . . . 393
Using Directives in Queries . . . 394
15. Gotchas . . . 394
1. Schema File Not Found . . . 394
2. Resolver Not Called . . . 394
3. Nested Resolver Returns Wrong Data . . . 394
4. Mutation Input Not Parsed . . . 395
5. N+1 Query Problem . . . 395
6. GraphQL Playground Returns 404 . . . 395
7. Type Mismatch Between Schema and Resolver . . . 395
14. SOAP / WSDL Services . . . 396
What is SOAP? . . . 396
Defining a SOAP Service . . . 396
Type Annotations Map to XSD . . . 397

A More Complete Example . . . 397
Lifecycle Hooks . . . 398
Testing with curl . . . 398
When to Use SOAP vs REST vs GraphQL . . . 399

Real-time with WebSocket **400**

1. The Refresh Button Problem . . . 400
2. What WebSocket Is . . . 400
3. @websocket -- WebSocket as a Route . . . 400
Starting the Server . . . 401
4. Connection Events . . . 401
Open . . . 401
Message . . . 401
Close . . . 402
A Complete Handler . . . 402
5. Sending to a Single Client . . . 403
6. Broadcasting to All Clients . . . 403
Broadcast Excluding Sender . . . 404
Sending JSON . . . 404
Closing a Connection . . . 405
Connection Methods Summary . . . 405
7. Path-Scoped Isolation . . . 405
8. Building a Live Chat . . . 406
WebSocket Handler . . . 406
9. Live Notifications . . . 408
10. Connecting from JavaScript . . . 409
Basic Connection . . . 409

Auto-Reconnect . . . 409
Using Native WebSocket . . . 410
11. A Complete Chat Page . . . 410
12. Exercise: Build a Real-Time Chat Room . . . 412
Requirements . . . 412
Test by: . . . 412
13. Solution . . . 413
14. Scaling with a Backplane . . . 415
Configuration . . . 415
Requirements . . . 415
15. Gotchas . . . 416
1. WebSocket Needs a Persistent Server . . . 416
2. Messages Are Strings, Not Dicts . . . 416
3. Connection Count Is Per-Path . . . 416
4. Broadcasting Does Not Scale Across Servers . . . 416
5. Large Messages Cause Disconnects . . . 416
6. Memory Leak from Tracking Connected Users . . . 417
7. No Authentication on WebSocket Connect . . . 417
Server-Sent Events (SSE) 418
1. The Polling Problem . . . 418
2. What SSE Is . . . 418
3. Your First Stream . . . 419
4. The SSE Protocol . . . 420
The data: Field . . . 420
The event: Field . . . 420
The id: Field . . . 420
The retry: Field . . . 420

The Delimiter . . . 421
5. Frontend: EventSource . . . 421
Basic Usage . . . 421
Named Events . . . 421
Closing the Connection . . . 422
With Authentication . . . 422
6. Real Examples . . . 422
Live Dashboard . . . 422
Build Progress . . . 423
LLM Chat Streaming . . . 423
File Processing Progress . . . 424
7. Custom Content Types . . . 425
NDJSON Streaming . . . 425
Consuming NDJSON on the Client . . . 425
8. SSE vs WebSocket . . . 426
9. Production Considerations . . . 427
Nginx Proxy Buffering . . . 427
Connection Limits . . . 427
Heartbeat Messages . . . 427
10. Exercise: Build a Live Metrics Dashboard . . . 428
Requirements . . . 428
Test by: . . . 428
11. Solution . . . 428
12. What You Learned . . . 430
WSDL / SOAP **431**
1. SOAP Is Not Dead . . . 431

2. Your First SOAP Service . . . 431
3. The @wsdl_operation Decorator . . . 431
4. Auto WSDL Generation at ?wsdl . . . 432
5. Calling the Service . . . 433
6. Lifecycle Hooks: onrequest and onresult . . . 433
7. Complex Types with Dataclasses . . . 434
8. Error Handling . . . 435
9. Exercise: Build a Currency Conversion SOAP Service . . . 435
Requirements . . . 435
Test with: . . . 436
10. Solution . . . 436
11. Gotchas . . . 437
1. Missing SOAPAction header . . . 437
2. Namespace mismatch . . . 437
3. Type annotations missing . . . 437
4. Forgetting the ?wsdl route . . . 437
DI Container **439**
1. Stop Constructing Dependencies Inside Your Code . . . 439
2. The Container Class . . . 439
3. register() — Transient Services . . . 439
4. singleton() — Cached Services . . . 440
5. get() and has() . . . 440
6. reset() . . . 440
7. Building a Service Container for Your App . . . 441
8. Testing with the Container . . . 441
9. Full Example: Order Route with DI . . . 443

10. Exercise: Configurable Notification Service . . . 443
Requirements . . . 443
Test with: . . . 444
11. Solution . . . 444
12. Gotchas . . . 445
1. Mutable singleton shared across requests . . . 445
2. reset() removes singleton cache but not registrations . . . 445
3. Circular dependencies . . . 445
4. Importing container before .env is loaded . . . 445
Service Runner **446**
1. Work That Never Stops . . . 446
2. A Basic Background Service . . . 446
3. Start / Stop Lifecycle . . . 446
4. Queue Worker Service . . . 447
5. Scheduled Task Service . . . 448
6. Multiple Services . . . 449
7. Stopping Services Gracefully . . . 449
8. Integrating with the App . . . 450
9. Exercise: Cache-Warming Background Service . . . 450
Requirements . . . 450
Test with: . . . 450
10. Solution . . . 451
11. Gotchas . . . 452
1. Blocking the main thread . . . 452
2. No stop() method . . . 452
3. Exceptions crashing the service thread . . . 452
4. Service starts before .env is loaded . . . 452

MCP Dev Tools 453

1. What is MCP? . . . 453

2. How It Works . . . 453

Localhost-Only Security . . . 453

3. Connecting AI Tools . . . 453

Claude Code . . . 453

Cursor . . . 454

Manual Connection . . . 454

4. Built-in Tools . . . 454

Database . . . 454

Routes . . . 454

Templates . . . 455

Files . . . 455

Migrations . . . 455

Data and ORM . . . 455

Infrastructure . . . 456

Debugging . . . 456

Project introspection . . . 456

Persistent code index . . . 457

Framework documentation (prose) . . . 457

Live API RAG (reflection) . . . 457

Plan management . . . 457

5. Protocol Details . . . 458

SSE Endpoint (GET) . . . 458

Message Endpoint (POST) . . . 458

Lifecycle . . . 459

6. Troubleshooting . . . 459

Building Custom MCP Servers 460

1. Beyond Dev Tools . . . 460
2. Creating an MCP Server . . . 460
3. Registering Tools with @mcp_tool . . . 460
4. Registering Resources with @mcp_resource . . . 461
5. Class-Based MCP Services . . . 461
6. Securing MCP Endpoints . . . 462
7. Testing MCP Tools . . . 462
8. Complete Example: CRM MCP Server . . . 463
9. Best Practices . . . 464

Dev Tools 465

1. Debugging at 2am . . . 465
2. Enabling the Dev Dashboard . . . 465
3. Dashboard Overview . . . 465
    - System Overview . . . 465
4. The Dev Toolbar . . . 466
    - Disabling the Toolbar . . . 466
5. Error Overlay . . . 466
    - Example Error . . . 467
    - Error Overlay in Production . . . 467
6. The Gallery . . . 467
7. Live Reload . . . 468
    - How It Works . . . 468
8. Hot-Patching with jurigged . . . 469
    - When Hot-Patching Cannot Help . . . 469
9. Request Inspector . . . 469

Request Details Panel . . . 470
Filtering Requests . . . 470
Request Replay . . . 470
10. SQL Query Runner . . . 470
Safety . . . 471
11. Queue Monitor . . . 471
12. System Info . . . 472
13. Exercise: Debug a Failing Route . . . 472
Setup . . . 472
Requirements . . . 473
Solution . . . 473
14. Gotchas . . . 474
1. Dev Dashboard Accessible on Network . . . 474
2. Live Reload Causes Connection Drops . . . 474
3. Error Overlay Shows in Production . . . 474
4. Request Inspector Slows Down the Server . . . 474
5. SQL Runner Executes Destructive Queries . . . 474
6. Hot-Patching Does Not Pick Up New Routes . . . 474
7. Template Changes Not Reflected . . . 475
8. Queue Monitor Shows No Jobs . . . 475
Tina4 CLI **476**
1. Getting a New Developer Up to Speed . . . 476
2. tina4 init -- Project Scaffolding . . . 476
Language Detection . . . 476
Explicit Language Selection . . . 477
Init into an Existing Directory . . . 477
3. tina4 serve -- Dev Server . . . 477

Options . . . 477
Bypassing the CLI . . . 478
4. tina4 generate model -- ORM Scaffolding . . . 478
Adding Fields . . . 478
Field Types . . . 479
Options . . . 479
5. tina4 generate route -- CRUD Route Scaffolding . . . 480
Options . . . 481
6. tina4 generate migration -- Migration Scaffolding . . . 481
When to Generate a Migration . . . 482
7. tina4 generate middleware -- Middleware Scaffolding . . . 482
8. Generate All at Once . . . 482
9. tina4 doctor -- Health Check . . . 483
10. tina4 test -- Running Tests . . . 483
Test Options . . . 484
11. tina4 routes -- Route Listing . . . 484
Filtering . . . 484
12. tina4 migrate -- Database Migrations . . . 484
Rollback . . . 485
Migration Table Auto-Upgrade . . . 485
Status . . . 485
13. Exercise: Scaffold a Feature in 5 Commands . . . 485
Requirements . . . 485
Expected Commands . . . 486
Solution . . . 486
14. Gotchas . . . 487
1. tina4 Command Not Found . . . 487

2. Wrong Language Detected . . . 487
3. Generated Files Overwrite Existing Code . . . 487
4. Migration Order Issues . . . 488
5. tina4 serve Uses Wrong Port . . . 488
6. Doctor Shows False Warnings . . . 488
7. Generate Creates Python 3.12+ Syntax . . . 488
8. Model Name Must Be PascalCase . . . 488
15. Documentation . . . 489
16. Test Port (Dual-Port Development) . . . 489
Vibe Coding with AI **490**
Why Tina4 is Built for AI-Assisted Development . . . 490
The Zero-Dependency Advantage . . . 490
CLAUDE.md -- The AI's Instruction Manual . . . 490
Skills -- Deep Framework Knowledge . . . 491
tina4-maintainer . . . 491
tina4-developer . . . 491
tina4-js . . . 491
The Convention Advantage . . . 491
Real-World Vibe Coding Session . . . 492
Python-Specific Patterns the AI Knows . . . 492
Route Decorators . . . 493
Template Rendering . . . 493
Middleware Attachment . . . 493
WebSocket Handlers . . . 493
Supported AI Tools . . . 494
The AI Chat in Dev Dashboard . . . 494
Tips for Effective Vibe Coding with Tina4 . . . 495

Exercise . . . 495
Gotchas . . . 495
1. No CLAUDE.md? . . . 495
2. AI Suggests Wrong Import Paths . . . 496
3. AI Hallucinates a Package . . . 496
4. AI Creates Files in Wrong Location . . . 496
5. AI Code Does Not Run . . . 496
6. AI Uses Synchronous Code . . . 496
7. AI Suggests Flask/Django Patterns . . . 496
The Philosophy . . . 497
Environment Variables **498**
Core Server . . . 499
Secrets and Authentication . . . 501
Database . . . 502
CORS . . . 503
Routing . . . 503
Security Headers . . . 503
Rate Limiting . . . 504
Sessions . . . 504
Redis/Valkey session backend . . . 505
MongoDB session backend . . . 505
Templates . . . 506
Cache . . . 506
Queues . . . 507
Kafka queue backend . . . 507
RabbitMQ queue backend . . . 507

MongoDB queue backend . . . 508
WebSocket . . . 508
Email . . . 509
Logging . . . 509
Uploads . . . 511
Localisation . . . 511
Services (background tasks) . . . 511
AI and MCP Tooling . . . 511
Swagger / OpenAPI . . . 513
GraphQL . . . 513
MCP (Model Context Protocol) . . . 514
Minimal .env for Development . . . 514
Minimal .env for Production . . . 514
Deployment **515**
1. From Development to Production . . . 515
2. Production .env Configuration . . . 515
Key Differences from Development . . . 515
Sensitive Values . . . 516
3. ASGI Server Auto-Detection . . . 516
How Auto-Detection Works . . . 516
Uvicorn Configuration . . . 517
How Many Workers? . . . 517
4. Docker Deployment . . . 517
Official Base Image . . . 517
Dockerfile . . . 517
.dockerignore . . . 518

Building and Running . . . 518
Adding Database Drivers . . . 518
Docker Compose . . . 519
5. Health Checks . . . 520
6. Graceful Shutdown . . . 521
Configuring Shutdown Timeout . . . 521
Docker Stop Grace Period . . . 521
7. Log Rotation . . . 521
Using logrotate (Linux) . . . 521
Docker Logging . . . 522
8. Reverse Proxy with Nginx . . . 522
9. SSL/TLS with Let's Encrypt . . . 523
With Nginx and Certbot . . . 523
With Docker (Using Traefik as Reverse Proxy) . . . 524
Certificate Monitoring . . . 524
10. Scaling . . . 525
Multiple Workers . . . 525
Load Balancing with Nginx . . . 525
Docker Scaling . . . 525
Scaling Considerations . . . 526
11. Monitoring . . . 526
Log Aggregation . . . 526
Uptime Monitoring . . . 527
Application Performance Monitoring (APM) . . . 527
12. Exercise: Docker Deploy . . . 528
Requirements . . . 528
Solution . . . 529

13. Gotchas . . . 529
1. .env Not Loaded in Docker . . . 529
2. SQLite Database Lost on Container Restart . . . 529
3. WebSocket Connections Drop Behind Nginx . . . 529
4. Built-in Server Used in Production . . . 530
5. Static Files Slow . . . 530
6. Memory Usage Grows Over Time . . . 530
7. Container Starts Before Database Is Ready . . . 530
8. SSL Certificate Not Renewing . . . 530
9. Scaled Instances Have Different State . . . 531
Building a Complete App **532**
1. Putting It All Together . . . 532
2. Planning the App . . . 532
Models . . . 532
Relationships . . . 532
Routes . . . 533
Templates . . . 533
3. Step 1: Init Project and Set Up Database . . . 533
Create Migrations . . . 533
4. Step 2: ORM Models . . . 534
5. Step 3: Authentication Routes . . . 536
Test Registration and Login . . . 537
6. Step 4: Auth Middleware . . . 537
7. Step 5: Task CRUD Routes . . . 538
Test the Task API . . . 540
8. Step 6: Dashboard Stats with Caching . . . 541
9. Step 7: WebSocket Real-Time Updates . . . 542

10. Step 8: Email Notifications . . . 543
11. Step 9: Dashboard Template . . . 544
12. Step 10: Tests . . . 546
13. Step 11: Docker Deployment . . . 548
14. The Complete Project Structure . . . 550
15. What You Built . . . 550
16. What to Build Next . . . 551
17. Closing Thoughts -- The Tina4 Philosophy . . . 552

Release Notes **553**

v3.12.4 (2026-05-06) . . . 553
25 new env vars across all 4 frameworks . . . 553
Logging — env-driven file output + rotation . . . 553
Documentation-truth CI gate now strict on both axes . . . 554
Tests added . . . 554
Upgrade path . . . 554
v3.12.3 (2026-05-05) . . . 554
Breaking changes (Ruby + PHP only) . . . 554
Doc parity — CLAUDE.md and book chapter 33 . . . 555
Other fixes . . . 555
Genuine gaps surfaced by the parity audit (follow-up, not blocking 3.12.3) . . . 555
Upgrade path . . . 556
v3.12.2 (2026-05-05) . . . 556
Firebird URL auto-detect . . . 556
Bug fix specific to PHP — mysqli localhost+port quirk . . . 557
Other fixes . . . 557
v3.12.1 (2026-05-04) . . . 557

v3.12.0 (2026-05-04) . . . 557
Why this release . . . 558
What changed . . . 558
Bug fixes shipped alongside the rename . . . 558
Migration — every renamed var . . . 559
Names that stay un-prefixed (not framework config) . . . 560
How to upgrade . . . 560
Parity . . . 560
v3.11.32 (2026-04-25) . . . 560
v3.11.13 (2026-04-16) . . . 561
v3.11.12 (2026-04-16) . . . 562
v3.11.11 (2026-04-16) . . . 563
v3.11.10 (2026-04-15) . . . 563
v3.11.9 (2026-04-15) . . . 564
v3.10.99 (2026-04-12) . . . 564
v3.10.97 (2026-04-11) . . . 565
v3.10.93 (2026-04-11) . . . 566
v3.10.92 (2026-04-10) . . . 566
v3.10.91 (2026-04-10) . . . 567
v3.10.90 (2026-04-09) . . . 567
v3.10.89 (2026-04-09) . . . 567
v3.10.86 (2026-04-09) . . . 568
v3.10.85 (2026-04-09) . . . 568
v3.10.84 (2026-04-09) . . . 568
v3.10.83 (2026-04-08) . . . 568
v3.10.70 (2026-04-06) . . . 569

Version History . . . 569
v3.10.68 (2026-04-03) — Full Parity Release . . . 569
v3.10.67 (2026-04-03) . . . 570
v3.10.66 (2026-04-03) . . . 570
v3.10.65 (2026-04-03) . . . 570
v3.10.60 (2026-04-03) . . . 571
v3.10.57 (2026-04-02) . . . 571
v3.10.54 (2026-04-02) . . . 571
v3.10.48 — April 2, 2026 . . . 572
Bug Fixes . . . 572
Test Coverage . . . 572
v3.10.46 — April 1, 2026 . . . 572
Test Coverage . . . 572
v3.10.45 — April 1, 2026 . . . 572
Bug Fixes . . . 572
v3.10.44 — April 1, 2026 . . . 573
New Features . . . 573
Bug Fixes . . . 573
Test Coverage . . . 573
v3.10.40 — April 1, 2026 . . . 574
Bug Fixes . . . 574
v3.10.39 — April 1, 2026 . . . 574
Breaking Changes . . . 574
New Features . . . 575
Test Coverage . . . 575
v3.10.38 — April 1, 2026 . . . 575
Code Metrics & Bubble Chart . . . 575

AI Context Installer . . . 576
Dashboard Improvements . . . 576
Cleanup . . . 576
v3.10.x — Previous Releases (March 28-31, 2026) . . . 576
Highlights . . . 576
Bug Fixes . . . 577
Breaking Changes . . . 579
v3.9.x — QueryBuilder and Sessions (March 26-27, 2026) . . . 580
Features . . . 580
Breaking Changes . . . 582
v3.8.x — Pooling, Validation, and Security (March 25-26, 2026) . . . 583
Features . . . 583
v3.7.x — Template Auto-Serve and Firebird Fixes (March 25, 2026) . . . 584
Features . . . 584
v3.6.x — API Parity (March 24, 2026) . . . 584
Breaking Changes . . . 584
v3.5.x — Bundled Frontend (March 24, 2026) . . . 586
Features . . . 586
v3.3.x — WebSockets, File Uploads, and Frond Improvements (March 24, 2026) . . . 586
Features . . . 586
Breaking Changes . . . 586
v3.2.x — Flexible Route Handlers and DevReload (March 24, 2026) . . . 588
Features . . . 588
Breaking Changes . . . 589
v3.1.x — Benchmarks and Internal Improvements (March 22, 2026) . . . 590
Features . . . 590
v3.0.0 — The Rewrite (March 22, 2026) . . . 590

What Changed . . . 590
Release Candidate History . . . 591
Upgrading from v2 to v3 592
1. Overview . . . 592
Step 0: Run the Automated Upgrade Command . . . 592
2. Package and Installation . . . 592
3. Project Structure Changes . . . 593
4. Routing Changes . . . 593
Decorators . . . 593
Auth Defaults . . . 594
Decorator Order . . . 594
Middleware . . . 594
Wildcard Routes . . . 594
5. Database Changes . . . 595
Connection URL . . . 595
Keyword Arguments . . . 595
Firebird Column Names . . . 595
Firebird Drivers . . . 595
Transactions . . . 596
Connection Pooling . . . 596
6. ORM Changes . . . 596
Field Definitions . . . 596
Binding the Database . . . 596
Auto Mapping . . . 597
Field Mapping . . . 597
Relationships . . . 597

7. Template Engine Changes . . . 597
Frond Replaces Template . . . 597
Custom Filters . . . 598
Custom Globals . . . 598
New Template Features . . . 598
8. Migration Tracking Table . . . 598
9. Authentication Changes . . . 599
10. Session Changes . . . 599
11. New Features in v3 . . . 600
Common Pitfalls . . . 600
1. POST/PUT/DELETE routes now require authentication . . . 600
2. Database connection strings changed . . . 601
3. Template engine renamed . . . 601
12. Step-by-Step Migration Checklist . . . 601
Complete Feature List **603**
1. What Tina4 Covers . . . 603
2. Foundations . . . 603
3. Templates . . . 604
4. Database . . . 605
5. Authentication and Security . . . 606
6. Performance . . . 606
7. Background Processing . . . 607
8. Events and Messaging . . . 607
9. APIs . . . 608
10. Tooling and Developer Experience . . . 609
11. Cross-Language Parity . . . 609

12. Test Coverage Summary 610

13. What Is Not in Tina4 610

Getting Started with Tina4 Python

1. What Is Tina4 Python

Tina4 Python is a zero-dependency web framework for Python 3.12+. One package. Routing, ORM, template engine, authentication, queues, WebSocket, and 70 other features -- all built in.

It belongs to the Tina4 family -- four identical frameworks in Python, PHP, Ruby, and Node.js. Everything you learn here transfers to the other three. Same project structure. Same template syntax. Same CLI commands. Same `.env` variables.

Tina4 Python follows Python convention: `snake_case` for methods and functions (`fetch_one()`, `soft_delete()`, `has_many()`), `PascalCase` for classes, `UPPER_SNAKE_CASE` for constants. Route handlers are `async def` functions decorated with `@get`, `@post`, and friends.

By the end of this chapter, you will have a running Tina4 Python project with an API endpoint and a rendered HTML page.

---

2. Prerequisites and Installation

What You Need

Three tools. Nothing else.

- **Python 3.12 or later** -- check with:

```
python3 --version
```

You should see output like:

```
Python 3.12.3
```

If you see a version lower than 3.12, upgrade Python first.

- **uv** -- a fast Python package manager and project tool. Check with:

```
uv --version
```

You should see:

```
uv 0.6.9
```

If uv is not installed, get it from https://docs.astral.sh/uv/:

```
# macOS / Linux
curl -LsSf https://astral.sh/uv/install.sh | sh

# Windows (PowerShell)
powershell -ExecutionPolicy ByPass -c "irm https://astral.sh/uv/install.ps1 | iex"
```

- **The Tina4 CLI** -- a Rust-based binary that manages all four Tina4 frameworks:

macOS (Homebrew):

```
brew install tina4stack/tap/tina4
```

Linux / macOS (install script):

```
curl -fsSL https://raw.githubusercontent.com/tina4stack/tina4/main/install.sh | bash
```

Windows (PowerShell):

```
irm https://raw.githubusercontent.com/tina4stack/tina4/main/install.ps1 | iex
```

Verify the CLI is installed:

```
tina4 --version
```

```
tina4 0.1.0
```

Installing the Tina4 CLI

```
cargo install tina4
```

Or download from GitHub Releases.

The `tina4` CLI manages project scaffolding, development servers, migrations, and more across all Tina4 frameworks.

Creating a New Project

One command. One package. No dependency tree.

```
tina4 init python my-store
```

`tina4 init` installs the Tina4 CLI globally (via cargo, homebrew, or direct download), then scaffolds a complete project with routes, templates, database, and configuration.

You should see:

```
Creating Tina4 project in ./my-store ...
  Detected language: Python (pyproject.toml)
  Created .env
  Created .env.example
  Created .gitignore
  Created src/routes/
  Created src/orm/
  Created migrations/
  Created src/seeds/
  Created src/templates/
  Created src/templates/errors/
  Created src/public/
  Created src/public/js/
  Created src/public/css/
  Created src/public/scss/
  Created src/public/images/
  Created src/public/icons/
  Created src/locales/
  Created data/
  Created logs/
  Created secrets/
  Created tests/

Project created! Next steps:
```

```
  cd my-store
  uv sync
  tina4 serve
```

Install the Python dependencies:

```
cd my-store
uv sync
```

```
Resolved 1 package in 0.8s
Installed 1 package in 0.3s
 + tina4-python==3.1.0
```

One package. No dependency tree. No version conflicts. Just `tina4-python`.

Starting the Dev Server

```
tina4 serve
```

```
  _____ _              _  _
|_   _(_)_ __   __ _| || |
  | | | | '_ \ / _` | || |_
  | | | | | | | (_| |__   _|
  |_| |_|_| |_|\__,_|  |_|

  Tina4 Python v3.2.0
  Server running at http://0.0.0.0:7146
  Debug mode: ON
  Database: sqlite:///data/app.db
  Press Ctrl+C to stop
```

Open your browser to `http://localhost:7146`. The Tina4 welcome page greets you.

Open `http://localhost:7146/health` in your browser or curl it:

```
curl http://localhost:7146/health
```

```
{
  "status": "ok",
  "database": "connected",
  "uptime_seconds": 12,
  "version": "3.2.0",
  "framework": "tina4-python"
}
```

Your Tina4 Python project is running.

---

3. Project Structure Walkthrough

Here is what `tina4 init` created:

```
my-store/
■■■ .env                    # Your configuration (gitignored)
■■■ .env.example            # Template for other developers
■■■ .gitignore              # Pre-configured
■■■ pyproject.toml          # Python project definition (uv / pip)
■■■ .venv/                  # Virtual environment (gitignored)
■■■ migrations/             # SQL migration files (project root)
■■■ src/
```

```
■   ■■■ routes/             # Your route handlers go here
■   ■■■ orm/                # Your ORM model classes go here
■   ■■■ seeds/              # Database seed files
■   ■■■ templates/          # Frond templates
■   ■   ■■■ errors/         # Custom 404.html, 500.html
■   ■■■ public/             # Static files (CSS, JS, images)
■   ■   ■■■ js/
■   ■   ■   ■■■ frond.js    # Auto-provided JS helper library
■   ■   ■■■ css/
■   ■   ■   ■■■ tina4.css   # Built-in CSS utility framework
■   ■   ■■■ scss/
■   ■   ■■■ images/
■   ■   ■■■ icons/
■   ■■■ locales/            # Translation files
■       ■■■ en.json
■■■ data/                   # SQLite databases (gitignored)
■■■ logs/                   # Log files (gitignored)
■■■ secrets/                # JWT keys (gitignored)
■■■ tests/                  # Your test files
```

Key directories:

- `src/routes/` -- Every `.py` file here is auto-loaded at startup. Drop your route definitions here. Organize into subdirectories if you want.
- `src/orm/` -- Every `.py` file here is auto-loaded. Drop your ORM model classes here.
- `src/templates/` -- Frond (Tina4's built-in template engine -- see Chapter 4: Templates) looks here when you call `response.render("my-page.html", data)`.
- `src/public/` -- Files served directly. `src/public/images/logo.png` lives at `/images/logo.png`.
- `data/` -- The default SQLite database (`app.db`) lives here. Gitignored because databases do not belong in version control.

---

4. Your First Route

Create the file `src/routes/greeting.py`:

```
from tina4_python.core.router import get

@get("/api/greeting/{name}")
async def greeting(name, request, response):
    from datetime import datetime
    return response.json({
        "message": f"Hello, {name}!",
        "timestamp": datetime.now().isoformat()
    })
```

In Python, path parameters arrive as function arguments. The parameter name in the function signature must match the `{name}` in the route pattern.

> ***Cross-language note.*** *Python is the odd one out here. PHP, Ruby, and Node read path parameters from* `$request->params` / `request.params` / `req.params` *instead. Same concept, two shapes — pick the chapter that matches your runtime.*

Save the file. The dev server picks up the change through live reload. If not, restart with `tina4 serve`.

Test It

Open your browser to:

```
http://localhost:7146/api/greeting/Alice
```

You should see:

```
{
  "message": "Hello, Alice!",
  "timestamp": "2026-03-22T14:30:00.000000"
}
```

Or use curl:

```
curl http://localhost:7146/api/greeting/Alice
```

```
{"message":"Hello, Alice!","timestamp":"2026-03-22T14:30:00.000000"}
```

The browser shows pretty-printed JSON (browser extensions or dev mode). Curl shows compact JSON. Force pretty output with `?pretty=true`:

```
curl "http://localhost:7146/api/greeting/Alice?pretty=true"
```

```
{
  "message": "Hello, Alice!",
  "timestamp": "2026-03-22T14:30:00.000000"
}
```

Understanding What Happened

Five steps. No magic.

- You created a file in `src/routes/`. Tina4 auto-discovered it at startup.
- `@get("/api/greeting/{name}")` registered a GET route with a path parameter `{name}`.
- When you requested `/api/greeting/Alice`, the router matched the pattern and called your handler.
- The router extracted `"Alice"` from the URL and passed it as the `name` argument to your function.
- `response.json(...)` serialized the dictionary to JSON, set `Content-Type: application/json`, and returned a `200 OK`.

Adding More HTTP Methods

Update `src/routes/greeting.py`:

```
from tina4_python.core.router import get, post

@get("/api/greeting/{name}")
async def greeting(name, request, response):
    from datetime import datetime
    return response.json({
        "message": f"Hello, {name}!",
        "timestamp": datetime.now().isoformat()
```

```
    })

@post("/api/greeting")
async def create_greeting(request, response):
    name = request.body.get("name", "World")
    language = request.body.get("language", "en")

    greetings = {
        "en": "Hello",
        "es": "Hola",
        "fr": "Bonjour",
        "de": "Hallo",
        "ja": "Konnichiwa"
    }

    greeting_word = greetings.get(language, greetings["en"])

    return response.json({
        "message": f"{greeting_word}, {name}!",
        "language": language
    }, 201)
```

Test the POST endpoint:

```
curl -X POST http://localhost:7146/api/greeting \
  -H "Content-Type: application/json" \
  -d '{"name": "Carlos", "language": "es"}'
```

```
{"message":"Hola, Carlos!","language":"es"}
```

The HTTP status code is `201 Created` (the second argument to `response.json()`).

---

5. Your First Template

Tina4 uses the **Frond** template engine -- a zero-dependency, Twig-compatible engine built from scratch. If you have used Twig, Jinja2, or Nunjucks, Frond will feel familiar.

Create a Base Layout

Create `src/templates/base.html`:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>{% block title %}My Store{% endblock %}</title>
    <link rel="stylesheet" href="/css/tina4.css">
    <style>
        body { font-family: -apple-system, BlinkMacSystemFont, "Segoe UI", Roboto, sans-serif; m
        .container { max-width: 960px; margin: 0 auto; padding: 20px; }
        .product-card { border: 1px solid #ddd; border-radius: 8px; padding: 16px; margin: 8px 0
        .product-card h3 { margin-top: 0; }
        .price { color: #2d8f2d; font-weight: bold; font-size: 1.2em; }
        .badge { display: inline-block; padding: 2px 8px; border-radius: 4px; font-size: 0.8em;
        .badge-success { background: #d4edda; color: #155724; }
        .badge-danger { background: #f8d7da; color: #721c24; }
```

```
        nav { background: #333; color: white; padding: 12px 20px; }
        nav a { color: white; text-decoration: none; margin-right: 16px; }
    </style>
</head>
<body>
    <nav>
        <a href="/">Home</a>
        <a href="/products">Products</a>
    </nav>
    <div class="container">
        {% block content %}{% endblock %}
    </div>
    <script src="/js/frond.js"></script>
</body>
</html>
```

This base layout defines two blocks (`title` and `content`) that child templates override. It includes `tina4.css` (the built-in CSS framework) and `frond.js` (the built-in JS helper library).

Create a Product Listing Page

Create `src/templates/products.html`:

```
{% extends "base.html" %}

{% block title %}Products - My Store{% endblock %}

{% block content %}
    <h1>Our Products</h1>
    <p>Showing {{ products | length }} product{{ "s" if products|length != 1 else "" }}</p>

    {% if products %}
        {% for product in products %}
            <div class="product-card">
                <h3>{{ product.name }}</h3>
                <p>{{ product.description }}</p>
                <p class="price">${{ "%.2f"|format(product.price) }}</p>
                {% if product.in_stock %}
                    <span class="badge badge-success">In Stock</span>
                {% else %}
                    <span class="badge badge-danger">Out of Stock</span>
                {% endif %}
            </div>
        {% endfor %}
    {% else %}
        <p>No products available at the moment.</p>
    {% endif %}
{% endblock %}
```

Create the Route That Renders the Template

Create `src/routes/pages.py`:

```
from tina4_python.core.router import get

@get("/products")
async def products_page(request, response):
    products = [
        {
            "name": "Wireless Keyboard",
```

```
            "description": "Ergonomic wireless keyboard with backlit keys.",
            "price": 79.99,
            "in_stock": True
        },
        {
            "name": "USB-C Hub",
            "description": "7-port USB-C hub with HDMI, SD card reader, and Ethernet.",
            "price": 49.99,
            "in_stock": True
        },
        {
            "name": "Monitor Stand",
            "description": "Adjustable aluminum monitor stand with cable management.",
            "price": 129.99,
            "in_stock": False
        },
        {
            "name": "Mechanical Mouse",
            "description": "High-precision wireless mouse with 16,000 DPI sensor.",
            "price": 59.99,
            "in_stock": True
        }
    ]

    return response.render("products.html", {"products": products})
```

See It in the Browser

Open `http://localhost:7146/products`. You should see:

- A dark navigation bar with "Home" and "Products" links
- The heading "Our Products"
- A subheading showing "Showing 4 products"
- Four product cards, each with name, description, price, and stock badge
- The "Monitor Stand" card shows a red "Out of Stock" badge
- The other three show green "In Stock" badges

How Template Rendering Works

Eight steps. All automatic.

- `response.render("products.html", {"products": products})` tells Frond to render `src/templates/products.html` with the given data.
- Frond sees `{% extends "base.html" %}` and loads the base template.
- The `{% block content %}` in `products.html` replaces the same block in `base.html`.
- `{{ product.name }}` outputs the value, auto-escaped for HTML safety.
- `{{ "%.2f"|format(product.price) }}` formats the number with 2 decimal places.
- `{% for product in products %}` loops through the list.
- `{% if product.in_stock %}` renders the stock badge conditionally.
- `{{ products | length }}` returns the item count.

About tina4css

The `tina4.css` file is Tina4's built-in CSS utility framework. Layout utilities. Typography. Common UI patterns. No Bootstrap. No Tailwind. No download. It ships with every scaffolded project.

---

6. Understanding .env

Open the `.env` file at the root of your project:

```
TINA4_DEBUG=true
```

That is likely all you see. The scaffold creates a minimal `.env` with debug mode enabled. Everything else uses defaults.

The important defaults for development:

Variable	Default Value	What It Means
`TINA4_PORT`	`7146`	Default server port (override with `tina4 serve --port`)
`TINA4_DATABASE_URL`	`sqlite:///data/app.db`	SQLite database in the `data/` directory
`TINA4_LOG_LEVEL`	`ALL`	All log messages are output
`CORS_ORIGINS`	`*`	All origins allowed (fine for development)
`TINA4_RATE_LIMIT`	`60`	60 requests per minute per IP

Log levels control how much output Tina4 produces:

Level	Behaviour
`ALL` / `DEBUG`	Full verbose output. DevReload active (live-reload, error overlay, hot-patching).
`INFO`	Standard logging. Startup messages, request summaries.
`WARNING`	Warnings and errors only.
`ERROR`	Errors only. Minimal output.

Set `TINA4_LOG_LEVEL=DEBUG` during development for maximum visibility. Use `WARNING` or `ERROR` in production.

To change the port, use the CLI flag or `.env`:

```
tina4 serve --port 8080
```

Or add it to your `.env` file:

```
TINA4_DEBUG=true
TINA4_PORT=8080
```

Restart the server. It now runs on port 8080.

How port resolution works: The Rust CLI (`tina4 serve`) determines the port using this priority order:

- **CLI flag** (highest priority): `tina4 serve --port 8080`
- `.env` **file**: `TINA4_PORT=8080`
- **Environment variable**: `PORT=8080`
- **Framework default** (PHP: 7145, Python: 7146, Ruby: 7147, Node.js: 7148)

The CLI reads your `.env` file and checks for `TINA4_PORT` (and falls back to `PORT`). The resolved port is passed to the Python server. All three methods work -- use whichever fits your workflow.

For the complete `.env` reference with all 68 variables, see Environment Variables.

---

7. The Dev Dashboard

With `TINA4_DEBUG=true` in your `.env`, Tina4 provides a built-in development dashboard. No additional environment variables are needed.

Restart the server and navigate to:

```
http://localhost:7146/__dev
```

The dashboard shows:

- **System Overview** -- framework version, Python version, uptime, memory usage, database status
- **Request Inspector** -- recent HTTP requests with method, path, status, duration, and request ID. Click any request to see full headers, body, database queries, and template renders.
- **Error Log** -- unhandled exceptions with stack traces and occurrence counts
- **Queue Manager** -- queue status (pending, reserved, failed, dead-letter messages)
- **WebSocket Monitor** -- active WebSocket connections with metadata
- **Routes** -- all registered routes with their methods, paths, and middleware

The dev dashboard shows you what your application is doing without adding print statements or log calls.

When you visit any HTML page (like `/products`), a **debug overlay** appears -- a toolbar at the bottom showing:

- Request details (method, URL, duration)
- Database queries executed (with timing)

- Template renders (with timing)
- Session data
- Recent log entries

This overlay is only visible when `TINA4_DEBUG=true`. Production never sees it.

---

8. Manual Setup (No CLI)

The `tina4` CLI scaffolds everything for you. But sometimes you create a project from scratch — an empty folder, a fresh virtual environment, no CLI installed yet. Here is the minimum you need.

Step 1: Install the Package

```
pip install tina4-python
```

Or with `uv`:

```
uv add tina4-python
```

Step 2: Create `app.py`

This is the entry point. Create a file called `app.py` in your project root:

```
"""Tina4 Application."""
from tina4_python.core import run

if __name__ == "__main__":
    run()
```

That is the entire file. The `run()` function starts the web server, scans for routes, and loads templates.

Step 3: Create the Folder Structure

Tina4 expects this layout:

```
my-project/
■■■ app.py
■■■ .env
■■■ src/
    ■■■ routes/       # Route files go here
    ■■■ templates/    # Twig templates go here
    ■■■ public/       # Static files (CSS, JS, images)
```

Create the directories:

```
mkdir -p src/routes src/templates src/public
```

Step 4: Create `.env`

```
TINA4_DEBUG=true
```

Step 5: Run It

```
tina4 serve
```

The server starts on `http://localhost:7146`. You should see the Tina4 welcome page. From here, add route files in `src/routes/` and templates in `src/templates/` — the same way as a CLI-scaffolded project.

> ***Note:*** *Tina4 Python refuses to start without the Rust CLI. If you genuinely need to bypass it (e.g. inside a Docker image that already wraps the framework), set* `TINA4_OVERRIDE_CLIENT=true` *in* `.env` *and run* `uv run python app.py` *directly.*

---

9. Request & Response Fundamentals

Before jumping into the exercises, let's consolidate how route handlers work in Tina4 Python. Every handler receives two objects: `request` (what the client sent) and `response` (what you send back). Here is the complete picture.

Reading Query Parameters

Query parameters are the key-value pairs after the `?` in a URL. Access them through `request.params`:

```
# URL: /api/search?q=laptop&page=2
request.params.get("q", "")       # "laptop"
request.params.get("page", "1")   # "2" (always a string)
request.params.get("sort", "name") # "name" (default -- param was not sent)
```

Reading URL Path Parameters

Route patterns like `/users/{id}` capture segments of the URL. In Tina4 Python, path parameters arrive as function arguments:

```
from tina4_python.core.router import get

@get("/users/{id:int}/posts/{slug}")
async def user_post(id, slug, request, response):
    # id = 5 (int, because of :int type hint)
    # slug = "hello-world" (string)
    return response.json({"user_id": id, "slug": slug})
```

The `{id:int}` syntax tells Tina4 to convert the value to an integer. Without `:int`, it stays a string.

Reading the Request Body

POST, PUT, and PATCH requests carry a body. Tina4 parses JSON bodies into a dictionary automatically (as long as the client sends `Content-Type: application/json`):

```
from tina4_python.core.router import post

@post("/api/items")
async def create_item(request, response):
    name = request.body.get("name", "")
    price = request.body.get("price", 0)
    return response.json({"received_name": name, "received_price": price})
```

Reading Headers

Headers are available as a dictionary. Header names are case-insensitive:

```
content_type = request.headers.get("Content-Type", "not set")
auth_token = request.headers.get("Authorization", "")
custom = request.headers.get("X-Custom-Header", "")
```

Sending JSON Responses

`response.json()` converts a dictionary to JSON and sets the correct `Content-Type`. Pass a status code as the second argument:

```
return response.json({"id": 1, "name": "Widget"})        # 200 OK (default)
return response.json({"id": 1, "name": "Widget"}, 201)    # 201 Created
return response.json({"error": "Not found"}, 404)          # 404 Not Found
```

Sending HTML / Template Responses

`response.render()` renders a Frond template from `src/templates/` and passes data to it:

```
return response.render("products.html", {"products": product_list, "title": "Our Products"})
```

For raw HTML without a template file:

```
return response.html("<h1>Hello</h1><p>This works too.</p>")
```

Status Codes

The most common status codes you will use:

Code	Meaning	When to Use
`200`	OK	Successful GET (default)
`201`	Created	Successful POST that created something
`400`	Bad Request	Client sent invalid input
`404`	Not Found	Resource does not exist
`500`	Internal Server Error	Something broke on the server

Worked Example: A Complete Route File

Here is a full route file that ties everything together. It builds a small book lookup API with query parameters, path parameters, JSON responses, and proper status codes. Read through it before attempting the exercises -- it is your reference.

Create `src/routes/books.py`:

```
from tina4_python.core.router import get, post

# In-memory data store
books = [
    {"id": 1, "title": "Dune", "author": "Frank Herbert", "year": 1965},
```

```
    {"id": 2, "title": "Neuromancer", "author": "William Gibson", "year": 1984},
    {"id": 3, "title": "Snow Crash", "author": "Neal Stephenson", "year": 1992}
]


@get("/api/books")
async def list_books(request, response):
    """List all books. Supports ?author= filter and ?sort=year."""
    author = request.params.get("author", "")
    sort_by = request.params.get("sort", "")

    result = books

    # Filter by author if the query param is present
    if author:
        result = [b for b in result if author.lower() in b["author"].lower()]

    # Sort by year if requested
    if sort_by == "year":
        result = sorted(result, key=lambda b: b["year"])

    return response.json({"books": result, "count": len(result)})


@get("/api/books/{id:int}")
async def get_book(id, request, response):
    """Get a single book by ID. Returns 404 if not found."""
    book = next((b for b in books if b["id"] == id), None)

    if book is None:
        return response.json({"error": f"Book with id {id} not found"}, 404)

    return response.json(book)


@post("/api/books")
async def create_book(request, response):
    """Create a new book from the JSON body. Returns 201 on success."""
    title = request.body.get("title", "")
    author = request.body.get("author", "")
    year = request.body.get("year", 0)

    if not title or not author:
        return response.json({"error": "title and author are required"}, 400)

    new_book = {
        "id": max(b["id"] for b in books) + 1,
        "title": title,
        "author": author,
        "year": year
    }
    books.append(new_book)

    return response.json(new_book, 201)
```

Test it:

```
# List all books
curl http://localhost:7146/api/books
```

```
# Filter by author
curl "http://localhost:7146/api/books?author=gibson"

# Sort by year
curl "http://localhost:7146/api/books?sort=year"

# Get a single book
curl http://localhost:7146/api/books/2

# Get a book that does not exist (returns 404)
curl http://localhost:7146/api/books/99

# Create a new book
curl -X POST http://localhost:7146/api/books \
  -H "Content-Type: application/json" \
  -d '{"title": "Foundation", "author": "Isaac Asimov", "year": 1951}'
```

This example covers every building block the exercises use: reading query parameters, reading path parameters, reading the request body, returning JSON with different status codes, and handling missing data. Refer back to it as you work through the exercises below.

---

10. Exercise: Greeting API + Product List Template

Build the following two features from scratch, without looking at the examples above.

Exercise Part A: Greeting API

Create an API endpoint at `GET /api/greet` that:

- Accepts a query parameter `name` (e.g., `/api/greet?name=Sarah`)
- If `name` is missing, defaults to `"Stranger"`
- Returns JSON like:

```
{
  "greeting": "Welcome, Sarah!",
  "time_of_day": "afternoon"
}
```

- The `time_of_day` should be calculated from the server's current hour:
- 5:00 - 11:59 = "morning"
- 12:00 - 16:59 = "afternoon"
- 17:00 - 20:59 = "evening"
- 21:00 - 4:59 = "night"

Test your endpoint with:

```
curl "http://localhost:7146/api/greet?name=Sarah"
curl "http://localhost:7146/api/greet"
```

Exercise Part B: Product List Page

Create a page at `GET /store` that:

- Displays a list of at least 5 products (hardcoded for now)
- Each product has: name, category, price, and a boolean `featured` flag
- Featured products get a visual highlight (different background color, border, or badge)
- The page shows the total number of products and the number of featured products
- Uses template inheritance -- a layout template and a page template that extends it
- Includes `tina4.css` and `frond.js`

Your products data should look like this in your route handler:

```
products = [
    {"name": "Espresso Machine", "category": "Kitchen", "price": 299.99, "featured": True},
    {"name": "Yoga Mat", "category": "Fitness", "price": 29.99, "featured": False},
    {"name": "Standing Desk", "category": "Office", "price": 549.99, "featured": True},
    {"name": "Noise-Canceling Headphones", "category": "Electronics", "price": 199.99, "featured
    {"name": "Water Bottle", "category": "Fitness", "price": 24.99, "featured": False}
]
```

Expected browser output:

- A page titled "Our Store"
- Text showing "5 products, 3 featured"
- A list of product cards with name, category, price, and a "Featured" badge on highlighted items
- Featured products have a distinct visual style (your choice -- different border color, background, star icon, etc.)

---

11. Solutions

Solution A: Greeting API

Create `src/routes/greet.py`:

```
from tina4_python.core.router import get

@get("/api/greet")
async def greet(request, response):
    name = request.params.get("name", "Stranger")

    from datetime import datetime
    hour = datetime.now().hour

    if 5 <= hour < 12:
        time_of_day = "morning"
    elif 12 <= hour < 17:
        time_of_day = "afternoon"
    elif 17 <= hour < 21:
        time_of_day = "evening"
    else:
        time_of_day = "night"

    return response.json({
```

```
        "greeting": f"Welcome, {name}!",
        "time_of_day": time_of_day
    })
```

Test:

```
curl "http://localhost:7146/api/greet?name=Sarah"

{"greeting":"Welcome, Sarah!","time_of_day":"afternoon"}

curl "http://localhost:7146/api/greet"

{"greeting":"Welcome, Stranger!","time_of_day":"afternoon"}
```

Solution B: Product List Page

Create `src/templates/store-layout.html`:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>{% block title %}Store{% endblock %}</title>
    <link rel="stylesheet" href="/css/tina4.css">
    <style>
        body { font-family: -apple-system, BlinkMacSystemFont, "Segoe UI", Roboto, sans-serif; m
        .container { max-width: 960px; margin: 0 auto; padding: 20px; }
        header { background: #1a1a2e; color: white; padding: 16px 20px; }
        header h1 { margin: 0; }
        .stats { color: #888; margin: 8px 0 20px; }
        .product-grid { display: grid; gap: 16px; }
        .product-card { background: white; border: 2px solid #e0e0e0; border-radius: 8px; paddin
        .product-card.featured { border-color: #ffc107; background: #fffdf0; }
        .product-name { font-size: 1.2em; font-weight: bold; margin: 0 0 4px; }
        .product-category { color: #666; font-size: 0.9em; }
        .product-price { color: #2d8f2d; font-weight: bold; font-size: 1.1em; margin-top: 8px; }
        .featured-badge { background: #ffc107; color: #333; padding: 2px 8px; border-radius: 4px
    </style>
</head>
<body>
    <header>
        <h1>{% block header %}Store{% endblock %}</h1>
    </header>
    <div class="container">
        {% block content %}{% endblock %}
    </div>
    <script src="/js/frond.js"></script>
</body>
</html>
```

Create `src/templates/store.html`:

```
{% extends "store-layout.html" %}

{% block title %}Our Store{% endblock %}
{% block header %}Our Store{% endblock %}

{% block content %}
    <p class="stats">{{ products | length }} products, {{ featured_count }} featured</p>

    <div class="product-grid">
```

```
        {% for product in products %}
            <div class="product-card{{ ' featured' if product.featured else '' }}">
                <p class="product-name">
                    {{ product.name }}
                    {% if product.featured %}
                        <span class="featured-badge">Featured</span>
                    {% endif %}
                </p>
                <p class="product-category">{{ product.category }}</p>
                <p class="product-price">${{ "%.2f"|format(product.price) }}</p>
            </div>
        {% endfor %}
    </div>
{% endblock %}
```

Create `src/routes/store.py`:

```
from tina4_python.core.router import get

@get("/store")
async def store_page(request, response):
    products = [
        {"name": "Espresso Machine", "category": "Kitchen", "price": 299.99, "featured": True},
        {"name": "Yoga Mat", "category": "Fitness", "price": 29.99, "featured": False},
        {"name": "Standing Desk", "category": "Office", "price": 549.99, "featured": True},
        {"name": "Noise-Canceling Headphones", "category": "Electronics", "price": 199.99, "feat
        {"name": "Water Bottle", "category": "Fitness", "price": 24.99, "featured": False}
    ]

    featured_count = sum(1 for p in products if p["featured"])

    return response.render("store.html", {
        "products": products,
        "featured_count": featured_count
    })
```

Open `http://localhost:7146/store` in your browser. You should see:

- A dark header reading "Our Store"
- Text showing "5 products, 3 featured"
- Five product cards in a grid
- Three cards (Espresso Machine, Standing Desk, Noise-Canceling Headphones) have a yellow border, light yellow background, and a "Featured" badge
- Two cards (Yoga Mat, Water Bottle) have a standard white background with gray border
- Each card shows the product name, category, and price formatted with two decimal places

---

12. Gotchas

1. File not auto-discovered

Problem: You created a route file but nothing happens when you visit the URL.

Cause: The file is not in `src/routes/`. It must be inside `src/routes/` (or a subdirectory), and the file must end with `.py`.

Fix: Move the file to `src/routes/your-file.py` and restart the server.

2. Missing import

Problem: `NameError: name 'get' is not defined` or similar.

Cause: You forgot to import the route decorator.

Fix: Start your route file with the correct import: `from tina4_python.core.router import get, post` (include whichever decorators you need).

3. Handler not async

Problem: Your route handler runs but returns an error about a coroutine object.

Cause: You defined the handler with `def` instead of `async def`. Tina4 Python expects all route handlers to be async.

Fix: Change `def greeting(request, response):` to `async def greeting(request, response):`. Every route handler must be `async def`.

4. Template not found

Problem: `Template "my-page.html" not found` error.

Cause: The template file is not in `src/templates/`, or there is a typo in the filename.

Fix: Check that the file exists at `src/templates/my-page.html`. The name in `response.render()` is relative to `src/templates/`.

5. Port already in use

Problem: `Error: Address already in use (port 7146)`

Cause: Another process (or another Tina4 instance) is using port 7146.

Fix: Stop the other process, or change the port with the CLI flag:

```
tina4 serve --port 8080
```

The CLI will also auto-increment the port if it detects the default port is in use.

6. Changes not reflected

Problem: You edited a file but the browser shows the old version.

Cause: Live reload may not be active. Browser caching can serve stale versions.

Fix: Hard-refresh the browser (`Ctrl+Shift+R` or `Cmd+Shift+R`). If that fails, restart the dev server with `Ctrl+C` and `tina4 serve`.

7. .env not loaded

Problem: Environment variables have no effect.

Cause: The `.env` file must be at the project root (same directory as `pyproject.toml`). A subdirectory placement hides it from Tina4.

Fix: Move `.env` to the project root.

Routing

1. How Routing Works in Tina4

Every web application maps URLs to code. A browser requests `/products`. The framework finds the handler for `/products`, runs it, sends back the result. That mapping is routing.

In Tina4 Python, routes live in Python files inside `src/routes/`. Every `.py` file in that directory (and its subdirectories) is auto-loaded at startup. No registration file. No central config. Drop a file in. It works.

The simplest route:

```
from tina4_python.core.router import get

@get("/hello")
async def hello(request, response):
    return response.json({"message": "Hello, World!"})
```

Save that as `src/routes/hello.py`, start the server with `tina4 serve`, and visit `http://localhost:7146/hello`:

```
{"message":"Hello, World!"}
```

One decorator. One function. Done.

---

2. HTTP Methods

Tina4 supports all five standard HTTP methods. Each has a decorator:

```
from tina4_python.core.router import get, post, put, patch, delete

@get("/products")
async def list_products(request, response):
    return response.json({"action": "list all products"})

@post("/products")
async def create_product(request, response):
    return response.json({"action": "create a product"}, 201)

@put("/products/{id}")
async def replace_product(id, request, response):
    return response.json({"action": f"replace product {id}"})

@patch("/products/{id}")
async def update_product(id, request, response):
    return response.json({"action": f"update product {id}"})

@delete("/products/{id}")
async def delete_product(id, request, response):
    return response.json({"action": f"delete product {id}"})
```

Test each one:

```
curl http://localhost:7146/products
```

```
{"action":"list all products"}
```

```
curl -X POST http://localhost:7146/products \
  -H "Content-Type: application/json" \
  -d '{"name": "Widget"}'
```

```
{"action":"create a product"}
```

```
curl -X PUT http://localhost:7146/products/42
```

```
{"action":"replace product 42"}
```

```
curl -X PATCH http://localhost:7146/products/42
```

```
{"action":"update product 42"}
```

```
curl -X DELETE http://localhost:7146/products/42
```

```
{"action":"delete product 42"}
```

`GET` reads. `POST` creates. `PUT` replaces. `PATCH` patches. `DELETE` removes. REST convention. Predictable API.

---

3. Path Parameters

Path parameters capture values from the URL. Wrap the parameter name in curly braces:

```
from tina4_python.core.router import get

@get("/users/{id}/posts/{post_id}")
async def user_post(id, post_id, request, response):
    return response.json({
        "user_id": id,
        "post_id": post_id
    })
```

```
curl http://localhost:7146/users/5/posts/99
```

```
{"user_id":"5","post_id":"99"}
```

Notice `user_id` came back as the string `"5"`, not the integer `5`. Path parameters are strings by default. In Python, path parameters are passed as function arguments -- the parameter names in the function signature must match the `{name}` placeholders in the route pattern.

> ***Auto-casting:*** *Tina4 automatically casts path parameter values that are purely numeric to integers. For example, requesting `/users/42/posts/99` will pass `id` as the integer `42` and `post_id` as the integer `99` to your handler -- no explicit `:int` type hint required. The `:int` type hint adds validation (rejecting non-numeric values with a 404), but the auto-casting happens regardless.*

Typed Parameters

Enforce a type by adding a colon and the type after the parameter name:

```
from tina4_python.core.router import get

@get("/orders/{id:int}")
async def get_order(id, request, response):
    # id is already an integer thanks to :int
```

```
    return response.json({
        "order_id": id,
        "type": type(id).__name__
    })
```

```
curl http://localhost:7146/orders/42
```

```
{"order_id":42,"type":"int"}
```

Pass a non-integer value and the route does not match. A 404:

```
curl http://localhost:7146/orders/abc
```

```
{"error":"Not found","path":"/orders/abc","status":404}
```

Supported types:

Type	Matches	Auto-cast	Example
`int`	Digits only	Integer	`{id:int}` matches `42` but not `abc`
`float`	Decimal numbers	Float	`{price:float}` matches `19.99`
`path`	All remaining path segments (catch-all)	String	`{slug:path}` matches `docs/api/auth`

The `{name}` form (no type) matches any single path segment and returns it as a string.

Typed Parameters in Action

Here is a complete example showing the most commonly used typed parameters together:

```
from tina4_python.core.router import get

# Integer parameter -- only digits match, auto-cast to int
@get("/products/{id:int}")
async def get_product(id, request, response):
    # id is an integer, e.g. 42
    return response.json({
        "product_id": id,
        "type": type(id).__name__
    })

# Float parameter -- decimal numbers, auto-cast to float
@get("/products/{id:int}/price/{price:float}")
async def check_price(id, price, request, response):
    return response.json({
        "product_id": id,
        "price": price,
        "type": type(price).__name__
    })

# Path parameter -- catch-all, captures remaining segments as a string
@get("/files/{filepath:path}")
async def serve_file(filepath, request, response):
    # filepath could be "images/photos/cat.jpg"
```

```
    return response.json({
        "filepath": filepath,
        "type": type(filepath).__name__
    })
```

```
# Integer route -- matches digits, returns an int
curl http://localhost:7146/products/42
```

```
{"product_id":42,"type":"int"}
```

```
# Integer route -- non-integer gives a 404
curl http://localhost:7146/products/abc
```

```
{"error":"Not found","path":"/products/abc","status":404}
```

```
# Path catch-all -- captures everything after /files/
curl http://localhost:7146/files/images/photos/cat.jpg
```

```
{"filepath":"images/photos/cat.jpg","type":"str"}
```

The `:int` and `:float` types act as both a constraint and a converter. If the URL segment does not match the expected pattern, the route is skipped entirely and Tina4 moves on to the next registered route (or returns 404 if nothing matches). The `:path` type is greedy -- it consumes all remaining segments, making it ideal for file paths and documentation URLs.

---

4. Query Parameters

Query parameters are the key-value pairs after the `?` in a URL. Access them through `request.params`:

```
from tina4_python.core.router import get

@get("/search")
async def search(request, response):
    q = request.params.get("q", "")
    page = int(request.params.get("page", 1))
    limit = int(request.params.get("limit", 10))

    return response.json({
        "query": q,
        "page": page,
        "limit": limit,
        "offset": (page - 1) * limit
    })
```

```
curl "http://localhost:7146/search?q=keyboard&page=2&limit=20"
```

```
{"query":"keyboard","page":2,"limit":20,"offset":20}
```

If a query parameter is missing, `request.params.get("key")` returns `None`. Use `.get()` with a default value.

---

5. Route Groups

A set of routes sharing a common prefix belongs in a `group()`:

```
from tina4_python.core.router import Router, get, post

Router.group("/api/v1", lambda: [
    Router.get("/users", list_users),
    Router.get("/users/{id:int}", get_user),
    Router.post("/users", create_user),
    Router.get("/products", list_products),
])

async def list_users(request, response):
    return response.json({"users": []})

async def get_user(id, request, response):
    return response.json({"user": {"id": id, "name": "Alice"}})

async def create_user(request, response):
    return response.json({"created": True}, 201)

async def list_products(request, response):
    return response.json({"products": []})
```

These routes register as `/api/v1/users`, `/api/v1/users/{id}`, and `/api/v1/products`. Short paths inside the group. Tina4 prepends the prefix. `Router.group()` is a classmethod that takes a prefix, a callback, and an optional middleware list.

```
curl http://localhost:7146/api/v1/users
```

```
{"users":[]}
```

```
curl http://localhost:7146/api/v1/products
```

```
{"products":[]}
```

Groups nest:

```
from tina4_python.core.router import Router

async def v1_status(request, response):
    return response.json({"version": "1.0"})

async def v2_status(request, response):
    return response.json({"version": "2.0"})

Router.group("/api", lambda: [
    Router.group("/v1", lambda: [
        Router.get("/status", v1_status),
    ]),
    Router.group("/v2", lambda: [
        Router.get("/status", v2_status),
    ]),
])
```

```
curl http://localhost:7146/api/v1/status
```

```
{"version":"1.0"}
```

```
curl http://localhost:7146/api/v2/status
```

```
{"version":"2.0"}
```

---

6. Middleware on Routes

Middleware is code that runs before or after your route handler. Authentication. Logging. Rate limiting. Input validation. Anything that belongs on multiple routes. Chapter 10 covers middleware in depth. Here is how it connects to routes.

Middleware on a Single Route

Attach middleware with the `@middleware` decorator. Middleware classes use `before_*` and `after_*` methods:

```
from tina4_python.core.router import get, middleware
import time

class LogRequest:
    def before_request(self, request, response):
        print(f"[{time.strftime('%Y-%m-%d %H:%M:%S')}] {request.method} {request.path}")
        return request, response

    def after_request(self, request, response):
        print(f"  Completed: {response.status_code}")
        return request, response

@middleware(LogRequest)
@get("/api/data")
async def get_data(request, response):
    return response.json({"data": [1, 2, 3]})
```

Middleware classes have `before_*` methods (called before the handler) and `after_*` methods (called after). If a `before_*` method sets `response.status_code` to 400 or above, the handler is skipped entirely. Useful for blocking unauthorized requests.

Blocking Middleware

Middleware that checks for an API key:

```
from tina4_python.core.router import get, middleware

class RequireApiKey:
    def before_check(self, request, response):
        api_key = request.headers.get("x-api-key", "")
        if api_key != "my-secret-key":
            return request, response.json({"error": "Invalid API key"}, 401)
        return request, response

@middleware(RequireApiKey)
@get("/api/secret")
async def secret_data(request, response):
    return response.json({"secret": "The answer is 42"})
```

```
curl http://localhost:7146/api/secret
```

```
{"error":"Invalid API key"}
```

Status: `401 Unauthorized`.

```
curl http://localhost:7146/api/secret -H "X-API-Key: my-secret-key"
```

```
{"secret":"The answer is 42"}
```

Multiple Middleware

Chain multiple middleware classes as arguments:

```
@middleware(LogRequest, RequireApiKey)
@get("/api/important")
async def important_data(request, response):
    return response.json({"data": "important stuff"})
```

Middleware runs left to right: `LogRequest` first, then `RequireApiKey`, then the route handler. If any middleware's `before_*` method returns a response with status 400+, the chain stops and the handler is skipped.

---

7. Route Decorators: @noauth() and @secured()

Two decorators control authentication at the route level.

@noauth() -- Public Routes

When your application has global authentication middleware, `@noauth()` marks a route as public:

```
from tina4_python.core.router import get, noauth

@noauth()
@get("/api/public/info")
async def public_info(request, response):
    return response.json({
        "app": "My Store",
        "version": "1.0.0"
    })
```

`@noauth()` tells Tina4 to skip authentication checks for this route, even if global auth middleware is configured in `.env` or applied to the parent group.

@secured() -- Protected GET Routes

`@secured()` marks a GET route as requiring authentication:

```
from tina4_python.core.router import get, secured

@secured()
@get("/api/profile")
async def profile(request, response):
    # request.user is populated by the auth middleware
    return response.json({
        "user": request.user
    })
```

By default, `POST`, `PUT`, `PATCH`, and `DELETE` routes are secured. `GET` routes are public unless you add `@secured()`. This matches the common pattern: reading data is public, modifying data requires authentication.

---

8. Route Chaining: .secure() and .cache()

Route decorators return a chainable object. Two methods you can call on any route: `.secure()` and `.cache()`.

.secure()

`.secure()` requires a valid bearer token in the `Authorization` header. If the token is missing or invalid, the route returns `401 Unauthorized` without ever reaching your handler:

```
from tina4_python.core.router import get

@get("/api/account")
async def get_account(request, response):
    return response.json({"account": request.user})

get_account.secure()
```

Or chain it inline using the `Router` class directly:

```
from tina4_python.core.router import Router

Router.get("/api/account", get_account).secure()
```

```
curl http://localhost:7146/api/account
# 401 Unauthorized

curl http://localhost:7146/api/account -H "Authorization: Bearer eyJhbGci..."
# 200 OK
```

.cache()

`.cache()` enables response caching for the route. Once the handler runs and produces a response, subsequent requests to the same URL return the cached result without re-executing the handler:

```
Router.get("/api/catalog", list_catalog).cache()
```

Chaining Both

Chain `.secure()` and `.cache()` together:

```
Router.get("/api/data", handler).secure().cache()
```

This route requires a bearer token and caches the response. Order does not matter -- `.cache().secure()` produces the same result.

---

9. Wildcard and Catch-All Routes

Wildcard Routes

Use `*` at the end of a path to match anything after it:

```
from tina4_python.core.router import get

@get("/docs/*")
```

```
async def docs_handler(request, response):
    path = request.params.get("*", "")
    return response.json({
        "section": "docs",
        "path": path
    })
```

```
curl http://localhost:7146/docs/getting-started
```

```
{"section":"docs","path":"getting-started"}
```

```
curl http://localhost:7146/docs/api/authentication/jwt
```

```
{"section":"docs","path":"api/authentication/jwt"}
```

Catch-All Route (Custom 404)

A catch-all handles any unmatched URL:

```
from tina4_python.core.router import get

@get("/*")
async def not_found(request, response):
    return response.json({
        "error": "Page not found",
        "path": request.path
    }, 404)
```

Define this route last (or in a file that sorts alphabetically after your other route files). Tina4 matches routes in registration order -- first match wins.

You can also create a custom 404 page by placing a template at `src/templates/errors/404.html`:

```
{% extends "base.html" %}

{% block title %}Not Found{% endblock %}

{% block content %}
    <h1>404 - Page Not Found</h1>
    <p>The page you are looking for does not exist.</p>
    <a href="/">Go back home</a>
{% endblock %}
```

Tina4 uses this template for any unmatched route when the file exists.

---

10. Route Listing via CLI

As your application grows, see all registered routes at a glance:

```
tina4 routes
```

```
Method   Path                          Middleware          Auth
------   ----                          ----------          ----
GET      /hello                        -                   public
GET      /products                     -                   public
POST     /products                     -                   secured
PUT      /products/{id}                -                   secured
PATCH    /products/{id}                -                   secured
```

```
DELETE   /products/{id}              -                  secured
GET      /api/v1/users               -                  public
GET      /api/v1/users/{id:int}      -                  public
POST     /api/v1/users               -                  secured
GET      /api/public/info            -                  @noauth
GET      /api/profile                -                  @secured
GET      /search                     -                  public
GET      /docs/*                     -                  public
```

The `Auth` column shows whether a route is public, secured (default for non-GET methods), `@noauth`, or `@secured`.

Filter by method:

```
tina4 routes --method POST
```

```
Method   Path                         Middleware         Auth
------   ----                         ----------         ----
POST     /products                    -                  secured
POST     /api/v1/users                -                  secured
```

Search for a path pattern:

```
tina4 routes --filter users
```

```
Method   Path                         Middleware         Auth
------   ----                         ----------         ----
GET      /api/v1/users                -                  public
GET      /api/v1/users/{id:int}       -                  public
POST     /api/v1/users                -                  secured
```

---

11. Organizing Route Files

Organize route files any way you want. Tina4 loads every `.py` file in `src/routes/` recursively. Two common patterns:

Pattern 1: One File Per Resource

```
src/routes/
■■■ products.py     # All product routes
■■■ users.py        # All user routes
■■■ orders.py       # All order routes
■■■ pages.py        # HTML page routes
```

Pattern 2: Subdirectories by Feature

```
src/routes/
■■■ api/
■   ■■■ products.py
■   ■■■ users.py
■   ■■■ orders.py
■■■ admin/
■   ■■■ dashboard.py
■   ■■■ settings.py
■■■ pages/
    ■■■ home.py
    ■■■ about.py
```

Both work identically. The directory structure has no effect on URL paths -- only the route definitions inside the files matter. Choose whichever keeps your project navigable.

---

12. Exercise: Build a Full CRUD API for Products

Build a complete REST API for managing products. All data stored in a Python list (no database yet -- Chapter 5 adds that).

Requirements

Create a file `src/routes/product_api.py` with the following routes:

Method	Path	Description
`GET`	`/api/products`	List all products. Support `?category=` filter.
`GET`	`/api/products/{id:int}`	Get a single product by ID. Return 404 if not found.
`POST`	`/api/products`	Create a new product. Return 201.
`PUT`	`/api/products/{id:int}`	Replace a product. Return 404 if not found.
`DELETE`	`/api/products/{id:int}`	Delete a product. Return 204 with no body.

Each product has: `id` (int), `name` (string), `category` (string), `price` (float), `in_stock` (bool).

Start with this seed data:

```
products = [
    {"id": 1, "name": "Wireless Keyboard", "category": "Electronics", "price": 79.99, "in_stock"
    {"id": 2, "name": "Yoga Mat", "category": "Fitness", "price": 29.99, "in_stock": True},
    {"id": 3, "name": "Coffee Grinder", "category": "Kitchen", "price": 49.99, "in_stock": False
    {"id": 4, "name": "Standing Desk", "category": "Office", "price": 549.99, "in_stock": True},
    {"id": 5, "name": "Running Shoes", "category": "Fitness", "price": 119.99, "in_stock": True}
]
```

Test with:

```
# List all
curl http://localhost:7146/api/products

# Filter by category
curl "http://localhost:7146/api/products?category=Fitness"

# Get one
curl http://localhost:7146/api/products/3

# Create
curl -X POST http://localhost:7146/api/products \
```

```
  -H "Content-Type: application/json" \
  -d '{"name": "Desk Lamp", "category": "Office", "price": 39.99, "in_stock": true}'

# Update
curl -X PUT http://localhost:7146/api/products/3 \
  -H "Content-Type: application/json" \
  -d '{"name": "Burr Coffee Grinder", "category": "Kitchen", "price": 59.99, "in_stock": true}'

# Delete
curl -X DELETE http://localhost:7146/api/products/3

# Not found
curl http://localhost:7146/api/products/999
```

---

13. Solution

Create `src/routes/product_api.py`:

```
from tina4_python.core.router import get, post, put, delete

# In-memory product store (resets on server restart)
products = [
    {"id": 1, "name": "Wireless Keyboard", "category": "Electronics", "price": 79.99, "in_stock"
    {"id": 2, "name": "Yoga Mat", "category": "Fitness", "price": 29.99, "in_stock": True},
    {"id": 3, "name": "Coffee Grinder", "category": "Kitchen", "price": 49.99, "in_stock": False
    {"id": 4, "name": "Standing Desk", "category": "Office", "price": 549.99, "in_stock": True},
    {"id": 5, "name": "Running Shoes", "category": "Fitness", "price": 119.99, "in_stock": True}
]

next_id = 6


# List all products, optionally filter by category
@get("/api/products")
async def list_products(request, response):
    category = request.params.get("category")

    if category is not None:
        filtered = [p for p in products if p["category"].lower() == category.lower()]
        return response.json({"products": filtered, "count": len(filtered)})

    return response.json({"products": products, "count": len(products)})


# Get a single product by ID
@get("/api/products/{id:int}")
async def get_product(id, request, response):
    for product in products:
        if product["id"] == id:
            return response.json(product)

    return response.json({"error": "Product not found", "id": id}, 404)


# Create a new product
@post("/api/products")
async def create_product(request, response):
```

```
    global next_id
    body = request.body

    if not body.get("name"):
        return response.json({"error": "Name is required"}, 400)

    product = {
        "id": next_id,
        "name": body["name"],
        "category": body.get("category", "Uncategorized"),
        "price": float(body.get("price", 0)),
        "in_stock": bool(body.get("in_stock", True))
    }
    next_id += 1

    products.append(product)

    return response.json(product, 201)


# Replace a product
@put("/api/products/{id:int}")
async def replace_product(id, request, response):
    body = request.body

    for i, product in enumerate(products):
        if product["id"] == id:
            products[i] = {
                "id": id,
                "name": body.get("name", product["name"]),
                "category": body.get("category", product["category"]),
                "price": float(body.get("price", product["price"])),
                "in_stock": bool(body.get("in_stock", product["in_stock"]))
            }
            return response.json(products[i])

    return response.json({"error": "Product not found", "id": id}, 404)


# Delete a product
@delete("/api/products/{id:int}")
async def delete_product(id, request, response):
    for i, product in enumerate(products):
        if product["id"] == id:
            products.pop(i)
            return response.json(None, 204)

    return response.json({"error": "Product not found", "id": id}, 404)
```

Expected output for the test commands:

List all:

```
{"products":[{"id":1,"name":"Wireless Keyboard","category":"Electronics","price":79.99,"in_stock
```

Filter by category:

```
{"products":[{"id":2,"name":"Yoga Mat","category":"Fitness","price":29.99,"in_stock":true},{"id"
```

Get one:

```
{"id":3,"name":"Coffee Grinder","category":"Kitchen","price":49.99,"in_stock":false}
```

Create:

```
{"id":6,"name":"Desk Lamp","category":"Office","price":39.99,"in_stock":true}
```

(Status: `201 Created`)

Update:

```
{"id":3,"name":"Burr Coffee Grinder","category":"Kitchen","price":59.99,"in_stock":true}
```

Delete: empty response with status `204 No Content`.

Not found:

```
{"error":"Product not found","id":999}
```

(Status: `404 Not Found`)

---

14. Gotchas

1. Trailing slashes matter

Problem: `/products` works but `/products/` returns a 404 (or vice versa).

Cause: Tina4 treats `/products` and `/products/` as different routes by default.

Fix: Pick one convention and stick with it. If you want both to work, register the route without a trailing slash -- Tina4 redirects `/products/` to `/products` when `TINA4_TRAILING_SLASH_REDIRECT=true` is set in `.env`.

2. Parameter names must be unique in a path

Problem: `/users/{id}/posts/{id}` behaves wrong -- both parameters share a name.

Cause: The second `{id}` overwrites the first in `request.params`.

Fix: Use distinct names: `/users/{user_id}/posts/{post_id}`.

3. Method conflicts

Problem: You defined `@get("/items/{id}")` and `@get("/items/{action}")` and the wrong handler runs.

Cause: Both patterns match `/items/42`. The first one registered wins.

Fix: Use typed parameters to disambiguate: `@get("/items/{id:int}")` matches integers only, leaving `/items/export` free for the other route. Or restructure your paths: `/items/{id:int}` and `/items/actions/{action}`.

4. Route handler must return a response

Problem: Your route handler runs but the browser shows an empty page or a 500 error.

Cause: You forgot the `return` statement. Without `return`, the handler returns `None` and Tina4 has nothing to send back.

Fix: Every handler must return something: `return response.json(...)` or `return response.html(...)` or `return response.render(...)`.

5. Decorator order matters

Problem: Your `@middleware` decorator has no effect, or your `@noauth()` is ignored.

Cause: Python decorators apply bottom-up. Wrong stacking order breaks registration.

Fix: Put the route decorator (`@get`, `@post`, etc.) first (closest to the function), then additional decorators above it:

```
@middleware(require_api_key)  # Applied second (wraps the route)
@get("/api/secret")           # Applied first (registers the route)
async def secret(request, response):
    ...
```

6. Forgetting async def

Problem: Your route handler raises a `TypeError` about a coroutine or the response is a coroutine object instead of JSON.

Cause: You used `def` instead of `async def`.

Fix: Every route handler in Tina4 Python must be `async def`. The framework runs on an async server. Change `def my_handler(request, response):` to `async def my_handler(request, response):`.

7. Group prefix must start with a slash

Problem: `Router.group("api/v1", ...)` produces routes that do not match.

Cause: The group prefix should start with `/` for consistency.

Fix: Start group prefixes with `/`: `Router.group("/api/v1", ...)`.

Request and Response

1. The Two Objects You Use Everywhere

Every route handler receives two arguments: `request` and `response`. The request carries what the client sent. The response builds what you send back. These two objects handle every HTTP scenario you will face.

A file-sharing service demonstrates the pattern. A user uploads a document. You validate the file type. You check the session. You return success JSON or an error page. Every step flows through `request` and `response`.

---

2. The Request Object

The `request` object opens every piece of the incoming HTTP request. Headers, body, cookies, files, IP address -- all accessible through named properties.

request.method

The HTTP method arrives as an uppercase string:

```
from tina4_python.core.router import get, post

@get("/api/check")
async def check_method(request, response):
    return response.json({"method": request.method})
```

```
curl http://localhost:7146/api/check
```

```
{"method":"GET"}
```

request.path

The URL path strips query parameters:

```
@get("/api/info")
async def info(request, response):
    return response.json({"path": request.path})
```

```
curl "http://localhost:7146/api/info?foo=bar"
```

```
{"path":"/api/info"}
```

Path Parameters (Function Arguments)

Path parameters captured from the URL pattern arrive as function arguments. The parameter names in your function signature must match the `{name}` placeholders in the route pattern:

```
@get("/users/{id:int}/posts/{slug}")
async def user_post(id, slug, request, response):
    return response.json({
        "user_id": id,
        "slug": slug
    })
```

```
curl http://localhost:7146/users/5/posts/hello-world
```

```
{"user_id":5,"slug":"hello-world"}
```

request.params

Query string parameters live in a dictionary. Each key holds a single value unless the same key appears multiple times -- then Tina4 stores a list:

```
@get("/search")
async def search(request, response):
    return response.json({
        "q": request.params.get("q", ""),
        "page": int(request.params.get("page", 1)),
        "sort": request.params.get("sort", "relevance")
    })
```

```
curl "http://localhost:7146/search?q=laptop&page=2&sort=price"
```

```
{"q":"laptop","page":2,"sort":"price"}
```

request.body

The parsed request body. JSON requests become a dictionary. Form submissions contain form fields. GET requests produce `None`:

```
@post("/api/feedback")
async def feedback(request, response):
    name = request.body.get("name", "Anonymous")
    message = request.body.get("message", "")
    rating = request.body.get("rating", 0)

    return response.json({
        "received": {
            "name": name,
            "message": message,
            "rating": rating
        }
    }, 201)
```

```
curl -X POST http://localhost:7146/api/feedback \
  -H "Content-Type: application/json" \
  -d '{"name": "Alice", "message": "Great product!", "rating": 5}'
```

```
{"received":{"name":"Alice","message":"Great product!","rating":5}}
```

request.headers

A dictionary of HTTP headers. Tina4 normalizes header names to lowercase:

```
@get("/api/headers")
async def show_headers(request, response):
    return response.json({
        "content_type": request.headers.get("content-type", "not set"),
        "user_agent": request.headers.get("user-agent", "not set"),
        "accept": request.headers.get("accept", "not set"),
        "custom": request.headers.get("x-custom-header", "not set")
    })
```

```
curl http://localhost:7146/api/headers -H "X-Custom-Header: hello-tina4"
```

```
{"content_type":"not set","user_agent":"curl/8.1.2","accept":"*/*","custom":"hello-tina4"}
```

request.ip

The client's IP address. Tina4 respects `X-Forwarded-For` and `X-Real-IP` headers behind a reverse proxy:

```
@get("/api/whoami")
async def whoami(request, response):
    return response.json({"ip": request.ip})
```

```
curl http://localhost:7146/api/whoami
```

```
{"ip":"127.0.0.1"}
```

request.cookies

A dictionary of cookies the client sent:

```
@get("/api/cookies")
async def show_cookies(request, response):
    return response.json({
        "cookies": request.cookies,
        "session_id": request.cookies.get("session_id", "none")
    })
```

```
curl http://localhost:7146/api/cookies -b "session_id=abc123; theme=dark"
```

```
{"cookies":{"session_id":"abc123","theme":"dark"},"session_id":"abc123"}
```

request.files

For multipart file uploads, `request.files` holds the uploaded files. Each file is a dictionary with four keys: `filename`, `type`, `content` (raw bytes), and `size` (in bytes):

```
@post("/api/upload")
async def upload(request, response):
    if "document" not in request.files:
        return response.json({"error": "No file uploaded"}, 400)

    file = request.files["document"]

    return response.json({
        "filename": file["filename"],
        "content_type": file["type"],
        "size": file["size"]
    })
```

```
curl -X POST http://localhost:7146/api/upload \
  -F "document=@report.pdf"
```

```
{"filename":"report.pdf","content_type":"application/pdf","size":45231}
```

To save the file to disk:

```
@post("/api/upload/save")
async def upload_and_save(request, response):
    if "photo" not in request.files:
        return response.json({"error": "No photo uploaded"}, 400)

    file = request.files["photo"]

    # Validate file type
    allowed = ["image/jpeg", "image/png", "image/webp"]
```

```
    if file["type"] not in allowed:
        return response.json({"error": f"File type {file['type']} not allowed"}, 400)

    # Save to public directory
    import os
    save_path = os.path.join("src", "public", "images", file["filename"])
    with open(save_path, "wb") as f:
        f.write(file["content"])

    return response.json({
        "message": "Photo uploaded",
        "url": f"/images/{file['filename']}"
    }, 201)
```

Handling Multiple File Uploads

When a form sends multiple files, each file field name maps to one entry in `request.files`. Use distinct field names for each file input:

```
@post("/api/upload-many")
async def upload_many(request, response):
    results = []

    for key, file in request.files.items():
        if not isinstance(file, dict) or "filename" not in file:
            continue

        ext = os.path.splitext(file["filename"])[1]
        unique_name = f"{uuid.uuid4().hex}{ext}"
        save_path = os.path.join("src", "public", "uploads", unique_name)

        os.makedirs(os.path.dirname(save_path), exist_ok=True)
        with open(save_path, "wb") as f:
            f.write(file["content"])

        results.append({
            "original_name": file["filename"],
            "saved_as": unique_name,
            "url": f"/uploads/{unique_name}"
        })

    if not results:
        return response.json({"error": "No files uploaded"}, 400)

    return response.json({"uploaded": results, "count": len(results)}, 201)
```

```
curl -X POST http://localhost:7146/api/upload-many \
  -F "photo=@sunset.jpg" \
  -F "document=@invoice.pdf" \
  -F "avatar=@profile.png"
```

```
{
  "uploaded": [
    {"original_name": "sunset.jpg", "saved_as": "a1b2c3d4.jpg", "url": "/uploads/a1b2c3d4.jpg"},
    {"original_name": "invoice.pdf", "saved_as": "e5f6a7b8.pdf", "url": "/uploads/e5f6a7b8.pdf"},
    {"original_name": "profile.png", "saved_as": "c9d0e1f2.png", "url": "/uploads/c9d0e1f2.png"}
  ],
  "count": 3
}
```

The HTML form uses distinct `name` attributes for each file input. Each field name becomes a key in `request.files`.

Upload Size Limits

Tina4 enforces a maximum upload size through the `TINA4_MAX_UPLOAD_SIZE` environment variable. The value is in bytes. The default is `10485760` (10 MB).

```
TINA4_MAX_UPLOAD_SIZE=10485760
```

When a client sends a body larger than this limit, Tina4 raises a `PayloadTooLarge` exception and returns a `413 Payload Too Large` response. Your handler never runs. The check fires before body parsing begins.

To allow 50 MB uploads, set this in your `.env` file:

```
TINA4_MAX_UPLOAD_SIZE=52428800
```

Calculate the value by multiplying: megabytes times 1,048,576. For 25 MB, that is `26214400`. For 100 MB, `104857600`. Choose the smallest limit that covers your use case. Large limits invite abuse.

---

3. The Response Object

The `response` object builds HTTP responses. Every route handler must return one. Methods chain, so you can set headers, cookies, and content in a single expression.

response.json()

Return JSON data with an optional status code:

```
@get("/api/users")
async def users(request, response):
    users = [
        {"id": 1, "name": "Alice"},
        {"id": 2, "name": "Bob"}
    ]
    return response.json({"users": users, "count": len(users)})

@post("/api/users")
async def create_user(request, response):
    # 201 Created
    return response.json({"id": 3, "name": request.body["name"]}, 201)

@get("/api/error")
async def error_example(request, response):
    # 400 Bad Request
    return response.json({"error": "Something went wrong"}, 400)
```

response.html()

Return raw HTML:

```
@get("/welcome")
async def welcome(request, response):
    return response.html("""
```

```
        <!DOCTYPE html>
        <html>
        <body>
            <h1>Welcome to My Store</h1>
            <p>Browse our <a href="/products">products</a>.</p>
        </body>
        </html>
    """)
```

response.text()

Return plain text:

```
@get("/robots.txt")
async def robots(request, response):
    return response.text("User-agent: *\nAllow: /")
```

response.render()

Render a Frond template with data (Chapter 4: Templates covers the engine in full):

```
@get("/dashboard")
async def dashboard(request, response):
    return response.render("dashboard.html", {
        "user_name": "Alice",
        "notifications": 3,
        "recent_orders": [
            {"id": 101, "total": 59.99},
            {"id": 102, "total": 124.50}
        ]
    })
```

response.redirect()

Send the client to another URL:

```
@get("/old-page")
async def old_page(request, response):
    return response.redirect("/new-page")

@post("/api/logout")
async def logout(request, response):
    # 303 See Other -- redirect after POST
    return response.redirect("/login", 303)

@get("/moved")
async def moved(request, response):
    # 301 Permanent redirect
    return response.redirect("/new-location", 301)
```

The default status code is `302 Found` (temporary redirect).

response.file()

Send a file as the response:

```
@get("/download/report")
async def download_report(request, response):
    return response.file("data/reports/monthly-report.pdf")
```

Tina4 auto-detects the content type from the file extension. The browser displays or downloads based on the MIME type.

To force a download instead of inline display, pass a `download_name`:

```
@get("/download/data")
async def download_data(request, response):
    return response.file("data/export.csv", download_name="sales-data.csv")
```

This sets the `Content-Disposition: attachment` header with your chosen filename.

response.xml()

Return XML content:

```
@get("/api/feed")
async def feed(request, response):
    return response.xml("<feed><entry><title>Hello</title></entry></feed>")
```

response.error()

Return a structured error envelope:

```
@post("/api/things")
async def create_thing(request, response):
    return response.error("VALIDATION_FAILED", "Name is required", 400)
```

This produces:

```
{"error":true,"code":"VALIDATION_FAILED","message":"Name is required","status":400}
```

Three arguments: an error code string, a human-readable message, and the HTTP status code. Clients check `error: true` and switch on the `code` field.

---

4. Status Codes

Every response method accepts a status code. Here are the codes you will use most:

Code	Meaning	When to Use
`200`	OK	Default. Successful GET, PUT, PATCH.
`201`	Created	Successful POST that created a resource.
`204`	No Content	Successful DELETE. No body needed.
`301`	Moved Permanently	URL has changed forever.
`302`	Found	Temporary redirect.
`400`	Bad Request	Invalid input from the client.

`401`	Unauthorized	Missing or invalid authentication.
`403`	Forbidden	Authenticated but not permitted.
`404`	Not Found	Resource does not exist.
`409`	Conflict	Duplicate or conflicting data. Two users claim the same username.
`413`	Payload Too Large	Body exceeds `TINA4_MAX_UPLOAD_SIZE`.
`422`	Unprocessable Entity	Valid JSON but fails business rules. The data parses but the logic rejects it.
`500`	Internal Server Error	Something broke on the server.

```
# 200 OK (default)
return response.json({"ok": True})

# 201 Created
return response.json({"id": 1}, 201)

# 204 No Content
return response.json(None, 204)

# 400 Bad Request
return response.json({"error": "Invalid input"}, 400)

# 401 Unauthorized
return response.json({"error": "Login required"}, 401)

# 403 Forbidden
return response.json({"error": "Not allowed"}, 403)

# 404 Not Found
return response.json({"error": "Not found"}, 404)

# 409 Conflict
return response.json({"error": "Username already taken"}, 409)

# 422 Unprocessable Entity
return response.json({"error": "Start date must precede end date"}, 422)

# 500 Internal Server Error
return response.json({"error": "Server error"}, 500)
```

You can also chain the status with `response.status()`:

```
return response.status(201).json({"id": 7, "created": True})
```

---

5. Content Negotiation

One endpoint can return different formats. The client declares what it wants through the `Accept` header. Your handler inspects that header and picks the right response method:

```
@get("/api/products/{id:int}")
async def product_detail(id, request, response):
    product = {
        "id": id,
        "name": "Wireless Keyboard",
        "price": 79.99
    }

    accept = request.headers.get("accept", "application/json")

    if "text/html" in accept:
        return response.render("product-detail.html", {"product": product})
    elif "text/plain" in accept:
        text = f"Product #{id}: {product['name']} - ${product['price']}"
        return response.text(text)
    elif "application/xml" in accept:
        xml = f'<product><id>{id}</id><name>{product["name"]}</name><price>{product["price"]}</p
        return response.xml(xml)
    else:
        return response.json(product)
```

```
# JSON (default)
curl http://localhost:7146/api/products/1
```

```
{"id":1,"name":"Wireless Keyboard","price":79.99}
```

```
# Plain text
curl http://localhost:7146/api/products/1 -H "Accept: text/plain"
```

```
Product #1: Wireless Keyboard - $79.99
```

```
# HTML (renders the template)
curl http://localhost:7146/api/products/1 -H "Accept: text/html"
```

```
<!DOCTYPE html>
<html>...rendered template...</html>
```

The pattern: check `request.headers.get("accept")`, match against known MIME types, fall back to JSON. Most API clients send `application/json` or `*/*`. Browsers send `text/html`. The same route serves both.

---

6. Custom Headers and Cookies

Setting Response Headers

Add custom headers with the `header` method. Chain calls before the final response:

```
@get("/api/data")
async def data_with_headers(request, response):
    return response.header("X-Custom-Header", "my-value") \
```

```
                    .header("X-Request-Id", "abc-123") \
                    .json({"data": [1, 2, 3]})
```

```
curl -i http://localhost:7146/api/data
```

```
HTTP/1.1 200 OK
Content-Type: application/json
X-Custom-Header: my-value
X-Request-Id: abc-123

{"data":[1,2,3]}
```

Setting Cookies

Set cookies on the response:

```
@post("/api/login")
async def login(request, response):
    # Set a session cookie
    return response.cookie("session_id", "abc123",
        path="/",
        max_age=3600,
        http_only=True,
        secure=True,
        same_site="Lax"
    ).json({"message": "Logged in"})

@post("/api/logout")
async def logout(request, response):
    # Delete a cookie by setting max_age to 0
    return response.cookie("session_id", "",
        path="/",
        max_age=0
    ).json({"message": "Logged out"})
```

Cookie keyword arguments:

Argument	Type	Default	Description
`path`	str	`"/"`	URL path scope
`max_age`	int	`3600`	Lifetime in seconds (0 deletes the cookie)
`http_only`	bool	`True`	JavaScript cannot access the cookie
`secure`	bool	`False`	Cookie travels over HTTPS only
`same_site`	str	`"Lax"`	`"Strict"`, `"Lax"`, or `"None"`

---

7. Input Validation

Tina4 ships a `Validator` class for declarative input validation. Chain rules together, then check the result.

The Validator Class

```
from tina4_python.validator import Validator

@post("/api/users")
async def create_user(request, response):
    v = Validator(request.body)
    v.required("name", "email").email("email").min_length("name", 2)

    if not v.is_valid():
        return response.error("VALIDATION_FAILED", v.errors()[0]["message"], 400)

    # proceed with valid data
```

The `Validator` accepts the request body (a dictionary) and provides chainable methods:

Method	Description
`required(*fields)`	Fields must be present and non-empty
`email(field)`	Field must be a valid email address
`min_length(field, n)`	Field must have at least `n` characters
`max_length(field, n)`	Field must have at most `n` characters
`integer(field)`	Field must be an integer
`min(field, n)`	Numeric field must be >= `n`
`max(field, n)`	Numeric field must be <= `n`
`in_list(field, values)`	Field must be one of the allowed values
`regex(field, pattern)`	Field must match the regular expression

Call `v.is_valid()` to check all rules. Call `v.errors()` to get the list of validation failures -- each entry holds a `field` and `message` key.

Note that `required()` accepts multiple field names in one call: `v.required("name", "email", "subject")`. This validates all three fields in a single statement.

---

8. Real-World Example: File Upload with Validation

A complete file upload endpoint. It validates type and size, generates a unique filename, and returns the URL:

```
from tina4_python.core.router import post
import os
import uuid
```

```
UPLOAD_DIR = "src/public/uploads"
MAX_FILE_SIZE = 5 * 1024 * 1024  # 5 MB
ALLOWED_TYPES = {
    "image/jpeg": ".jpg",
    "image/png": ".png",
    "image/webp": ".webp",
    "application/pdf": ".pdf"
}

@post("/api/files/upload")
async def upload_file(request, response):
    if "file" not in request.files:
        return response.json({"error": "No file provided. Send a file with field name 'file'"},

    file = request.files["file"]

    # Check file type
    if file["type"] not in ALLOWED_TYPES:
        return response.json({
            "error": f"File type '{file['type']}' not allowed",
            "allowed": list(ALLOWED_TYPES.keys())
        }, 400)

    # Check file size
    if file["size"] > MAX_FILE_SIZE:
        return response.json({
            "error": f"File too large. Maximum size is {MAX_FILE_SIZE // (1024 * 1024)} MB",
            "size": file["size"]
        }, 400)

    # Generate a unique filename
    ext = ALLOWED_TYPES[file["type"]]
    unique_name = f"{uuid.uuid4().hex}{ext}"

    # Save the file
    os.makedirs(UPLOAD_DIR, exist_ok=True)
    save_path = os.path.join(UPLOAD_DIR, unique_name)
    with open(save_path, "wb") as f:
        f.write(file["content"])

    return response.json({
        "message": "File uploaded successfully",
        "file": {
            "original_name": file["filename"],
            "saved_as": unique_name,
            "url": f"/uploads/{unique_name}",
            "size": file["size"],
            "content_type": file["type"]
        }
    }, 201)
```

```
curl -X POST http://localhost:7146/api/files/upload \
  -F "file=@photo.jpg"
```

```
{
  "message": "File uploaded successfully",
  "file": {
    "original_name": "photo.jpg",
    "saved_as": "a1b2c3d4e5f6.jpg",
    "url": "/uploads/a1b2c3d4e5f6.jpg",
```

```
    "size": 245760,
    "content_type": "image/jpeg"
  }
}
```

---

9. Exercise: Build a Contact Form API

Build an API that processes contact form submissions with full validation.

Requirements

Create `src/routes/contact.py` with these endpoints:

Method	Path	Description
`POST`	`/api/contact`	Submit a contact form. Validate all fields. Return 201 on success.
`GET`	`/api/contact/submissions`	List all submissions. Support `?status=` filter.

The contact form body must include: `name` (required, 2-100 chars), `email` (required, must contain @), `subject` (required), `message` (required, 10+ chars), `urgency` (optional, one of "low", "medium", "high", default "medium").

Validation rules:

- Return 400 with an `errors` list if any field fails
- Store submissions in a Python list with an auto-incremented ID and timestamp
- Each submission starts with a `status` of "new"

Test with:

```
# Valid submission
curl -X POST http://localhost:7146/api/contact \
  -H "Content-Type: application/json" \
  -d '{"name": "Alice", "email": "alice@example.com", "subject": "Question", "message": "I have

# Invalid submission (missing fields, short message)
curl -X POST http://localhost:7146/api/contact \
  -H "Content-Type: application/json" \
  -d '{"name": "A", "email": "bad-email", "message": "Short"}'

# List all
curl http://localhost:7146/api/contact/submissions

# Filter by status
curl "http://localhost:7146/api/contact/submissions?status=new"
```

---

10. Solution

```
from tina4_python.core.router import get, post
from datetime import datetime

submissions = []
next_id = 1


@post("/api/contact")
async def submit_contact(request, response):
    global next_id
    body = request.body
    errors = []

    # Validate name
    name = body.get("name", "")
    if not name:
        errors.append("Name is required")
    elif len(name) < 2 or len(name) > 100:
        errors.append("Name must be between 2 and 100 characters")

    # Validate email
    email = body.get("email", "")
    if not email:
        errors.append("Email is required")
    elif "@" not in email:
        errors.append("Email must contain @")

    # Validate subject
    subject = body.get("subject", "")
    if not subject:
        errors.append("Subject is required")

    # Validate message
    message = body.get("message", "")
    if not message:
        errors.append("Message is required")
    elif len(message) < 10:
        errors.append("Message must be at least 10 characters")

    # Validate urgency
    urgency = body.get("urgency", "medium")
    if urgency not in ("low", "medium", "high"):
        errors.append("Urgency must be one of: low, medium, high")

    if errors:
        return response.json({"errors": errors}, 400)

    submission = {
        "id": next_id,
        "name": name,
        "email": email,
        "subject": subject,
        "message": message,
        "urgency": urgency,
        "status": "new",
        "created_at": datetime.now().isoformat()
    }
```

```
    next_id += 1
    submissions.append(submission)

    return response.json({
        "message": "Contact form submitted successfully",
        "submission": submission
    }, 201)


@get("/api/contact/submissions")
async def list_submissions(request, response):
    status = request.params.get("status")

    if status:
        filtered = [s for s in submissions if s["status"] == status]
        return response.json({"submissions": filtered, "count": len(filtered)})

    return response.json({"submissions": submissions, "count": len(submissions)})
```

Valid submission output:

```
{
  "message": "Contact form submitted successfully",
  "submission": {
    "id": 1,
    "name": "Alice",
    "email": "alice@example.com",
    "subject": "Question",
    "message": "I have a question about your product pricing.",
    "urgency": "high",
    "status": "new",
    "created_at": "2026-03-22T14:30:00.000000"
  }
}
```

(Status: `201 Created`)

Invalid submission output:

```
{
  "errors": [
    "Name must be between 2 and 100 characters",
    "Email must contain @",
    "Subject is required",
    "Message must be at least 10 characters"
  ]
}
```

(Status: `400 Bad Request`)

---

11. Gotchas

1. request.body is None for GET requests

Problem: Accessing `request.body` in a GET handler returns `None`.

Cause: GET requests carry no body. Browsers and curl send none by convention.

Fix: Use `request.params` for GET parameters. The body populates only for POST, PUT, and PATCH requests.

2. Forgetting the Content-Type header

Problem: `request.body` is empty even though you sent JSON.

Cause: The `Content-Type: application/json` header is missing. Without it, Tina4 does not parse the body as JSON.

Fix: Include `-H "Content-Type: application/json"` in your curl command. In JavaScript `fetch()`, set `headers: {"Content-Type": "application/json"}`.

3. File upload with wrong field name

Problem: `request.files["photo"]` raises a `KeyError`.

Cause: The form field name does not match the key you look up.

Fix: Check that the form uses `<input type="file" name="photo">` and curl uses `-F "photo=@file.jpg"`. The string after `-F` must match the key in `request.files`.

4. response.redirect() in AJAX calls

Problem: Your JavaScript `fetch()` call receives a redirect response, but the browser does not navigate.

Cause: `fetch()` follows redirects silently. It returns the final response. It does not trigger browser navigation.

Fix: For AJAX calls, return JSON with a redirect URL. Handle navigation in JavaScript: `window.location.href = data.redirect_url`. Use `response.redirect()` only for traditional form submissions and browser navigation.

5. Cookie not being set

Problem: You called `response.cookie()` but the cookie does not appear in the browser.

Cause: Setting `secure=True` restricts the cookie to HTTPS. On `http://localhost`, the browser drops it.

Fix: During development on HTTP, use `secure=False` (the default). Enable `secure=True` only in production with HTTPS configured.

6. Large file uploads fail

Problem: Uploading a large file returns a 413 error or empty `request.files`.

Cause: The file exceeds `TINA4_MAX_UPLOAD_SIZE`. The default limit is 10 MB (10,485,760 bytes).

Fix: Set `TINA4_MAX_UPLOAD_SIZE` in your `.env` file. For 50 MB: `TINA4_MAX_UPLOAD_SIZE=52428800`. The check happens before your handler runs -- there is nothing to catch in your route code.

---

7. Chaining response methods in wrong order

Problem: Headers or cookies set after calling `response.json()` have no effect.

Cause: `response.json()` finalizes and returns the response object. Calling methods on a separate line after the return is dead code.

Fix: Chain headers and cookies before the final response method:

```
# Correct
return response.header("X-Custom", "value").cookie("token", "abc").json({"ok": True})

# Wrong -- header is set after json() already returned
result = response.json({"ok": True})
response.header("X-Custom", "value")  # Too late
return result
```

8. Header names are lowercase

Problem: `request.headers.get("Content-Type")` returns `None` even though the header exists.

Cause: Tina4 normalizes all header names to lowercase during parsing. The ASGI protocol delivers them this way.

Fix: Use lowercase keys: `request.headers.get("content-type")`. This applies to all headers -- `"authorization"`, `"user-agent"`, `"x-custom-header"`.

Templates

1. Beyond JSON -- Rendering HTML

Every route so far returns JSON. That works for APIs. Web applications need HTML -- product listings, dashboards, login forms, email templates. Tina4 uses the **Frond** template engine for this work.

Frond is a zero-dependency template engine built from scratch. Its syntax matches Twig, Jinja2, and Nunjucks. Three constructs drive the entire engine: `{{ }}` for output, `{% %}` for logic, `{# #}` for comments. That is the whole grammar.

This chapter builds toward a product catalog page. Items in a grid. Featured products highlighted. Prices formatted. Layout inherited from a shared template. One engine handles it all.

---

2. The @template Decorator

The shortest path to a rendered page:

```
from tina4_python.core.router import get, template

@get("/about")
@template("about.html")
async def about_page(request, response):
    return {
        "title": "About Us",
        "company": "My Store",
        "founded": 2020
    }
```

> ***Decorator order matters.*** `@template` *must sit* ***below*** *the route decorator (*`@get`*/*`@post`*/...) so the template wrapper is what gets registered with the router. If* `@template` *is above the route decorator, the router only sees the raw handler and the template wrapping is never applied — the page renders a bare dict.*

Create `src/templates/about.html`:

```
<!DOCTYPE html>
<html>
<head><title>{{ title }}</title></head>
<body>
    <h1>{{ title }}</h1>
    <p>{{ company }} was founded in {{ founded }}.</p>
</body>
</html>
```

Visit `http://localhost:7146/about` and the rendered page appears.

The `@template` decorator stacks above `@get` (or `@post`, etc.). When the handler returns a dictionary, the decorator passes that dict to `response.render()` with the named template. If the handler returns something other than a dict -- an already-built Response, for instance -- the decorator passes it through unchanged. You can also call `response.render()` directly in any

route handler. The decorator is shorthand.

---

3. Variables and Output

Basic Output

Double curly braces print a variable:

```
<h1>Hello, {{ name }}!</h1>
<p>Your balance is {{ balance }}.</p>
```

With data `{"name": "Alice", "balance": 150.50}`, this renders:

```
<h1>Hello, Alice!</h1>
<p>Your balance is 150.5.</p>
```

Accessing Nested Properties

Dot notation reaches into dictionaries and object attributes:

```
<p>{{ user.name }}</p>
<p>{{ user.address.city }}</p>
<p>{{ order.items.0.name }}</p>
```

With data:

```
{
    "user": {
        "name": "Alice",
        "address": {"city": "Cape Town"}
    },
    "order": {
        "items": [{"name": "Keyboard"}, {"name": "Mouse"}]
    }
}
```

Renders:

```
<p>Alice</p>
<p>Cape Town</p>
<p>Keyboard</p>
```

Method Calls and Slicing

Frond supports calling methods on values inside templates. If a dictionary value is callable, you invoke it with arguments:

```
{{ user.t("greeting") }}       {# calls user.t("greeting") #}
{{ text[:10] }}                {# slice syntax -- first 10 characters #}
{{ items[2:5] }}               {# slice from index 2 to 5 #}
```

Operators inside quoted function arguments (such as `+`, `-`, `*`) parse without breaking the expression.

Auto-Escaping

Frond escapes HTML characters in output by default. XSS attacks die here:

```
<p>{{ user_input }}</p>
```

With `{"user_input": "<script>alert('hacked')</script>"}`, this renders:

```
<p>&lt;script&gt;alert(&#39;hacked&#39;)&lt;/script&gt;</p>
```

The script tag becomes plain text. It never executes. If you need raw HTML output and you trust the source, use the `|safe` filter:

```
<div>{{ trusted_html | safe }}</div>
```

---

4. Filters

Filters transform output. The pipe `|` applies them:

```
{{ name | upper }}          {# ALICE #}
{{ name | lower }}          {# alice #}
{{ name | capitalize }}     {# Alice #}
{{ name | title }}          {# Alice Smith #}
{{ bio | trim }}            {# Removes leading/trailing whitespace #}
```

Complete Filter Reference

String Filters

Filter	Example	Description	
`upper`	`{{ name |	upper }}`	Convert to uppercase
`lower`	`{{ name |	lower }}`	Convert to lowercase
`capitalize`	`{{ name |	capitalize }}`	Capitalize first letter
`title`	`{{ name |	title }}`	Capitalize each word
`trim`	`{{ name |	trim }}`	Strip leading/trailing whitespace
`ltrim`	`{{ name |	ltrim }}`	Strip leading whitespace
`rtrim`	`{{ name |	rtrim }}`	Strip trailing whitespace
`slug`	`{{ title |	slug }}`	Convert to URL-friendly slug
`wordwrap(80)`	`{{ text |	wordwrap(80) }}`	Wrap text at N characters
`truncate(100)`	`{{ text |	truncate(100) }}`	Truncate to N characters with ellipsis
`nl2br`	`{{ text |	nl2br }}`	Convert newlines to ` ` tags
`striptags`	`{{ html |	striptags }}`	Remove all HTML tags
`replace("a", "b")`	`{{ text |	replace("old", "new") }}`	Replace occurrences of a substring

Array Filters

Filter	Example	Description	
`length`	`{{ items \	length }}`	Count items in array or string length
`reverse`	`{{ items \	reverse }}`	Reverse order of items
`sort`	`{{ items \	sort }}`	Sort items ascending
`shuffle`	`{{ items \	shuffle }}`	Randomly shuffle items
`first`	`{{ items \	first }}`	Get the first item
`last`	`{{ items \	last }}`	Get the last item
`join(", ")`	`{{ items \	join(", ") }}`	Join array items with separator
`split(",")`	`{{ csv \	split(",") }}`	Split string into array
`unique`	`{{ items \	unique }}`	Remove duplicate values
`filter`	`{{ items \	filter }}`	Remove falsy values from array
`map("name")`	`{{ items \	map("name") }}`	Extract a property from each item
`column("name")`	`{{ items \	column("name") }}`	Extract a column from array of objects
`batch(3)`	`{{ items \	batch(3) }}`	Group items into batches of N
`slice(0, 3)`	`{{ items \	slice(0, 3) }}`	Extract a slice from offset with length

Encoding Filters

Filter	Example	Description	
`escape` (`e`)	`{{ text | escape }}`	HTML-escape special characters
`raw` (`safe`)	`{{ html | raw }}`	Output without auto-escaping
`url_encode`	`{{ text | url_encode }}`	URL-encode a string
`base64_encode` (`base64encode`)	`{{ text | base64_encode }}`	Base64-encode a string
`base64_decode` (`base64decode`)	`{{ data | base64_decode }}`	Base64-decode a string
`md5`	`{{ text | md5 }}`	Compute MD5 hash
`sha256`	`{{ text | sha256 }}`	Compute SHA-256 hash

Numeric Filters

Filter	Example	Description	
`abs`	`{{ num | abs }}`	Absolute value
`round(2)`	`{{ price | round(2) }}`	Round to N decimal places
`number_format(2)`	`{{ price | number_format(2) }}`	Format with decimals and thousands separator
`int`	`{{ val | int }}`	Cast to integer
`float`	`{{ val | float }}`	Cast to float
`string`	`{{ val | string }}`	Cast to string

JSON Filters

Filter	Example	Description	
`json_encode`	`{{ data | json_encode }}`	Encode value as JSON string
`to_json` (`tojson`)	`{{ data | to_json }}`	Encode value as JSON string (alias)
`json_decode`	`{{ str | json_decode }}`	Decode JSON string to object
`js_escape`	`{{ text | js_escape }}`	Escape string for safe use in JavaScript

Dict Filters

Filter	Example	Description	
`keys`	`{{ obj | keys }}`	Get dictionary keys as array	
`values`	`{{ obj | values }}`	Get dictionary values as array	
`merge(other)`	`{{ defaults | merge(overrides) }}`	Merge two dictionaries	

Other Filters

Filter	Example	Description	
`default("fallback")`	`{{ name | default("Guest") }}`	Fallback when value is empty or undefined	
`date("Y-m-d")`	`{{ created | date("Y-m-d") }}`	Format a date value	
`format(val)`	`{{ "%.2f" | format(price) }}`	Format string with value (sprintf-style)	
`data_uri`	`{{ content | data_uri }}`	Convert to a data URI string	
`dump`	`{{ var | dump }}` or `{{ dump(var) }}`	Debug output — gated on `TINA4_DEBUG=true` (see *Dumping Values*)	
`form_token`	`{{ form_token() }}`	Generate a CSRF hidden input with token	
`formTokenValue`	`{{ formTokenValue("context") }}`	Return the raw JWT token string	
`to_json`	`{{ data | to_json }}`	JSON-encode a value (no double-escaping)	
`js_escape`	`{{ text | js_escape }}`	Escape for safe use in JavaScript strings	

Chaining Filters

Filters chain left to right:

```
{{ name | trim | lower | capitalize }}
{# "  alice smith  " → "alice smith" → "Alice smith" #}

{{ items | sort | reverse | first }}
{# Sort, reverse, take first = largest item #}
```

The default Filter

A fallback when a variable is empty or undefined:

```
{{ username | default("Guest") }}
```

```
{{ bio | default("No bio provided.") }}
{{ theme | default("light") }}
```

Dumping Values for Debugging

The `dump` helper lets you inspect any variable mid-template. Two interchangeable forms are supported:

```
{{ user | dump }}
{{ dump(user) }}
```

Both produce the same `<pre>`-wrapped, HTML-escaped `repr()` of the value. Handles nested dicts, lists, class instances, and cyclic references without crashing — Python's `repr()` prints `{...}` for back-edges.

```
{{ dump(order) }}

{# Output: #}
{# <pre>{'id': 42, 'items': [...], 'total': 99.99}</pre> #}
```

dump is gated on `TINA4_DEBUG=true`. In production (env var unset or `false`) **both** the filter and function form silently return an empty string. This prevents accidental leaks of internal state, object shapes, or sensitive values into rendered HTML if a developer leaves a `{{ dump(x) }}` call in a template.

```
# .env — dev
TINA4_DEBUG=true    # dump() outputs the value

# .env — production
TINA4_DEBUG=false   # dump() is a no-op
```

You can rely on this gate for safety, but treat `dump` as a development-only convenience. For structured output in production code paths, use `to_json`.

---

5. Control Tags

if / elif / else

```
{% if user.role == "admin" %}
    <span class="badge">Admin</span>
{% elif user.role == "moderator" %}
    <span class="badge">Moderator</span>
{% else %}
    <span class="badge">Member</span>
{% endif %}
```

Comparisons and logical operators:

```
{% if price > 100 and in_stock %}
    <p>Premium item, available now!</p>
{% endif %}

{% if not user.verified %}
    <p>Please verify your email.</p>
{% endif %}
```

```
{% if not items %}
    <p>Your cart is empty.</p>
{% endif %}
```

for Loops

```
{% for product in products %}
    <div class="product-card">
        <h3>{{ product.name }}</h3>
        <p>${{ "%.2f"|format(product.price) }}</p>
    </div>
{% endfor %}
```

Inside a for loop, the `loop` variable provides iteration context:

Variable	Description
`loop.index`	Current iteration (1-based)
`loop.index0`	Current iteration (0-based)
`loop.first`	True on the first iteration
`loop.last`	True on the last iteration
`loop.length`	Total number of items

```
<ol>
{% for item in items %}
    <li class="{{ 'first' if loop.first else '' }} {{ 'last' if loop.last else '' }}">
        {{ loop.index }}. {{ item.name }}
    </li>
{% endfor %}
</ol>
```

for / else

The `{% else %}` block inside `{% for %}` runs when the list is empty:

```
{% for product in products %}
    <div class="product-card">{{ product.name }}</div>
{% else %}
    <p>No products found.</p>
{% endfor %}
```

set -- Local Variables

Create or update a variable inside a template:

```
{% set greeting = "Hello" %}
{% set full_name = user.first_name ~ " " ~ user.last_name %}
{% set total = price * quantity %}
{% set discount = total - rebate %}

<p>{{ greeting }}, {{ full_name }}!</p>
<p>Total: {{ total }}, After discount: {{ discount }}</p>
```

The `~` operator concatenates strings. Arithmetic operators (`+`, `-`, `*`, `/`, `//`, `%`, `**`) work in `set` and expressions.

When combining filters with arithmetic, assign the filtered values first:

```
{% set dr = account.dr|default(0) %}
{% set cr = account.cr|default(0) %}
{% set balance = dr - cr %}
<p>Balance: {{ balance }}</p>
```

---

6. Template Inheritance

Template inheritance kills duplication. A base template defines blocks. Child templates override them. One layout file controls every page.

Base Template

Create `src/templates/base.html`:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>{% block title %}My Store{% endblock %}</title>
    <link rel="stylesheet" href="/css/tina4.css">
    {% block head %}{% endblock %}
</head>
<body>
    <nav>
        <a href="/">Home</a>
        <a href="/products">Products</a>
        <a href="/about">About</a>
    </nav>

    <main>
        {% block content %}{% endblock %}
    </main>

    <footer>
        <p>&copy; 2026 My Store</p>
    </footer>

    <script src="/js/frond.js"></script>
    {% block scripts %}{% endblock %}
</body>
</html>
```

Child Template

Create `src/templates/home.html`:

```
{% extends "base.html" %}

{% block title %}Home - My Store{% endblock %}

{% block content %}
    <h1>Welcome to My Store</h1>
    <p>Browse our collection of quality products.</p>
{% endblock %}
```

When Frond renders `home.html`:

- It sees `{% extends "base.html" %}` and loads the base template.
- The `{% block title %}` in `home.html` replaces the one in `base.html`.
- The `{% block content %}` in `home.html` replaces the one in `base.html`.
- Blocks not overridden (`head`, `scripts`) keep their default content -- empty here.

Calling Parent Blocks

Use `{{ parent() }}` to include the parent block's content:

```
{% extends "base.html" %}

{% block head %}
    {{ parent() }}
    <link rel="stylesheet" href="/css/products.css">
{% endblock %}
```

The `head` block now contains everything from the base plus the extra stylesheet.

---

7. Include and Macro

include

Pull in another template file:

```
{% include "partials/header.html" %}

<main>
    <h1>Products</h1>
</main>

{% include "partials/footer.html" %}
```

Pass variables to included templates:

```
{% include "partials/product-card.html" with {"product": featured_product} %}
```

macro -- Reusable Template Functions

Macros are functions for templates. Define once, call everywhere:

Create `src/templates/macros/forms.html`:

```
{% macro input(name, label, type="text", value="", required=false) %}
    <div class="form-group">
        <label for="{{ name }}">{{ label }}{% if required %} *{% endif %}</label>
        <input type="{{ type }}" name="{{ name }}" id="{{ name }}"
               value="{{ value }}" {{ "required" if required else "" }}>
    </div>
{% endmacro %}

{% macro textarea(name, label, value="", rows=4) %}
    <div class="form-group">
        <label for="{{ name }}">{{ label }}</label>
```

```
        <textarea name="{{ name }}" id="{{ name }}" rows="{{ rows }}">{{ value }}</textarea>
    </div>
{% endmacro %}

{% macro button(text, type="submit", class="btn-primary") %}
    <button type="{{ type }}" class="btn {{ class }}">{{ text }}</button>
{% endmacro %}
```

Use them:

```
{% import "macros/forms.html" as forms %}

<form method="POST" action="/api/contact">
    {{ forms.input("name", "Your Name", required=true) }}
    {{ forms.input("email", "Email Address", type="email", required=true) }}
    {{ forms.textarea("message", "Your Message", rows=6) }}
    {{ forms.button("Send Message") }}
</form>
```

Change the macro once and every form in your application updates. Consistent markup across the entire project.

---

8. Comments

Template comments use `{# #}`. Frond strips them from the output:

```
{# This comment will not appear in the HTML source #}
<p>Visible content</p>

{#
    Multi-line comments work too.
    Use them to document template logic.
#}
```

HTML comments (`<!-- -->`) reach the browser. Frond comments never do.

---

9. Special Tags

{% raw %} -- Literal Output

Output literal `{{ }}` or `{% %}` without processing. This tag saves you when embedding Vue.js or Angular templates:

```
{% raw %}
    <div id="app">
        {{ message }}
    </div>
{% endraw %}
```

Frond outputs the literal text `{{ message }}`. No variable lookup. No expression parsing.

{% spaceless %} -- Remove Whitespace

Strip whitespace between HTML tags:

```
{% spaceless %}
    <div>
        <span>Hello</span>
    </div>
{% endspaceless %}
```

Output:

```
<div><span>Hello</span></div>
```

Inline elements create visible gaps when whitespace sits between them. The `spaceless` tag eliminates those gaps.

{% autoescape %} -- Control Escaping

Override auto-escaping for a block of content:

```
{% autoescape false %}
    {{ trusted_html }}
{% endautoescape %}
```

Everything inside outputs without HTML escaping. This works the same as `| raw` on every variable, but handles large blocks of trusted content with less repetition. Never use this with user-submitted data.

Whitespace Control

Template tags occupy a full line and produce blank lines in the output. Use `{%-` and `-%}` to strip surrounding whitespace:

```
{%- for item in items -%}
    <li>{{ item.name }}</li>
{%- endfor -%}
```

The `-` on the left strips whitespace before the tag. The `-` on the right strips whitespace after. The output contains no blank lines between list items.

---

10. tina4css

The `tina4.css` file is Tina4's built-in CSS utility framework. It ships with every project. Layout utilities. Typography. Spacing. Common UI patterns. No Bootstrap. No Tailwind. No separate download.

Include it in your base template:

```
<link rel="stylesheet" href="/css/tina4.css">
```

Layout Classes

The grid system uses a 12-column layout:

```
<div class="container">
    <div class="row">
        <div class="col-6">Left half</div>
        <div class="col-6">Right half</div>
    </div>
```

```
</div>
```

Flex layout for alignment:

```
<div class="flex justify-between items-center">
    <h1>Title</h1>
    <button class="btn btn-primary">Action</button>
</div>
```

CSS grid for card layouts:

```
<div class="grid grid-cols-3 gap-4">
    <div class="card">Item 1</div>
    <div class="card">Item 2</div>
    <div class="card">Item 3</div>
</div>
```

Buttons

Five button styles cover most use cases:

```
<button class="btn btn-primary">Primary</button>
<button class="btn btn-secondary">Secondary</button>
<button class="btn btn-success">Success</button>
<button class="btn btn-warning">Warning</button>
<button class="btn btn-danger">Danger</button>
```

Link-style buttons use the same classes on anchor tags:

```
<a href="/dashboard" class="btn btn-primary">Go to Dashboard</a>
<a href="/cancel" class="btn btn-secondary">Cancel</a>
```

Cards

Cards group related content with optional header and footer sections:

```
<div class="card">
    <div class="card-header">Order Summary</div>
    <div class="card-body">
        <p>3 items in your cart</p>
        <p class="text-primary">Total: $149.97</p>
    </div>
    <div class="card-footer">
        <button class="btn btn-primary">Checkout</button>
    </div>
</div>
```

Cards work well inside a grid for catalog-style layouts:

```
<div class="grid grid-cols-3 gap-4">
    {% for product in products %}
    <div class="card">
        <div class="card-header">{{ product.name }}</div>
        <div class="card-body">
            <p>${{ "%.2f"|format(product.price) }}</p>
        </div>
    </div>
    {% endfor %}
</div>
```

Alerts

Alert boxes communicate status messages to the user:

```
<div class="alert alert-success">Order placed. Check your email for confirmation.</div>
<div class="alert alert-danger">Payment failed. Your card was declined.</div>
<div class="alert alert-warning">Your session expires in 5 minutes.</div>
<div class="alert alert-info">New features are available. See the changelog.</div>
```

Forms

Form controls use `form-group` for spacing and `form-control` for input styling:

```
<form method="POST" action="/api/contact">
    <div class="form-group">
        <label for="name">Full Name</label>
        <input type="text" id="name" name="name" class="form-control" required>
    </div>

    <div class="form-group">
        <label for="email">Email Address</label>
        <input type="email" id="email" name="email" class="form-control" required>
    </div>

    <div class="form-group">
        <label for="message">Message</label>
        <textarea id="message" name="message" class="form-control" rows="4"></textarea>
    </div>

    <div class="form-group">
        <label for="priority">Priority</label>
        <select id="priority" name="priority" class="form-control">
            <option value="low">Low</option>
            <option value="medium" selected>Medium</option>
            <option value="high">High</option>
        </select>
    </div>

    <button type="submit" class="btn btn-primary">Send Message</button>
    <button type="reset" class="btn btn-secondary">Clear</button>
</form>
```

Tables

Tables gain borders and row striping with tina4css classes:

```
<table class="table">
    <thead>
        <tr>
            <th>#</th>
            <th>Product</th>
            <th>Price</th>
        </tr>
    </thead>
    <tbody>
        {% for product in products %}
        <tr>
            <td>{{ loop.index }}</td>
            <td>{{ product.name }}</td>
            <td>${{ "%.2f"|format(product.price) }}</td>
```

```
        </tr>
        {% endfor %}
    </tbody>
</table>
```

Spacing and Typography Utilities

```
{# Spacing #}
<div class="p-4 m-2">Padded and margined</div>
<div class="mt-4">Margin top</div>
<div class="mb-2">Margin bottom</div>
<div class="px-3">Horizontal padding</div>

{# Typography #}
<p class="text-lg text-gray-600">Large gray text</p>
<p class="text-center">Centered text</p>
<p class="text-right">Right-aligned text</p>
<span class="text-muted">Gray text</span>
<span class="text-primary">Primary color text</span>
```

No external dependencies. If you prefer Bootstrap or Tailwind, swap the `<link>` tag. Tina4 does not care which CSS framework you choose.

---

11. Exercise: Build a Product Catalog Page

Build a product catalog page with categories, filtering, and a detail view.

Requirements

- Create a route at `GET /catalog` that renders a product catalog page
- Create a route at `GET /catalog/{id:int}` that renders a single product detail page
- Use template inheritance with a base layout
- Display products in a grid with:
- Product name, price (formatted to 2 decimal places), category
- "In Stock" / "Out of Stock" badge
- Featured products get a highlighted border
- Show category filter links at the top (all categories from the data)
- Support `?category=` query parameter to filter products
- The detail page shows full product info including description
- Use at least one macro (e.g., for the product card)
- Use the `|default` filter somewhere meaningful

Use this data:

```
products = [
    {"id": 1, "name": "Espresso Machine", "category": "Kitchen", "price": 299.99, "in_stock": Tr
    {"id": 2, "name": "Yoga Mat", "category": "Fitness", "price": 29.99, "in_stock": True, "feat
    {"id": 3, "name": "Standing Desk", "category": "Office", "price": 549.99, "in_stock": True,
    {"id": 4, "name": "Noise-Canceling Headphones", "category": "Electronics", "price": 199.99,
```

```
    {"id": 5, "name": "Water Bottle", "category": "Fitness", "price": 24.99, "in_stock": True, "
    {"id": 6, "name": "Desk Lamp", "category": "Office", "price": 79.99, "in_stock": True, "feat
]
```

---

12. Solution

Create `src/templates/macros/catalog.html`:

```
{% macro product_card(product) %}
    <div class="product-card {{ 'featured' if product.featured else '' }}">
        {% if product.featured %}
            <span class="featured-badge">Featured</span>
        {% endif %}
        <h3><a href="/catalog/{{ product.id }}">{{ product.name }}</a></h3>
        <p class="category">{{ product.category }}</p>
        <p class="price">${{ "%.2f"|format(product.price) }}</p>
        {% if product.in_stock %}
            <span class="badge badge-success">In Stock</span>
        {% else %}
            <span class="badge badge-danger">Out of Stock</span>
        {% endif %}
    </div>
{% endmacro %}
```

Create `src/templates/catalog.html`:

```
{% extends "base.html" %}
{% import "macros/catalog.html" as catalog %}

{% block title %}{{ page_title | default("Product Catalog") }}{% endblock %}

{% block content %}
    <h1>{{ page_title | default("Product Catalog") }}</h1>

    <div class="category-filters">
        <a href="/catalog" class="{{ 'active' if active_category is not defined else '' }}">All<
        {% for cat in categories %}
            <a href="/catalog?category={{ cat }}"
               class="{{ 'active' if active_category == cat else '' }}">{{ cat }}</a>
        {% endfor %}
    </div>

    <p class="stats">Showing {{ products | length }} product{{ "s" if products|length != 1 else

    <div class="product-grid">
        {% for product in products %}
            {{ catalog.product_card(product) }}
        {% else %}
            <p>No products found in this category.</p>
        {% endfor %}
    </div>
{% endblock %}
```

Create `src/templates/product_detail.html`:

```
{% extends "base.html" %}

{% block title %}{{ product.name }} - My Store{% endblock %}
```

```
{% block content %}
    <a href="/catalog">&larr; Back to catalog</a>

    <div class="product-detail">
        <h1>{{ product.name }}</h1>
        <p class="category">{{ product.category }}</p>
        <p class="price">${{ "%.2f"|format(product.price) }}</p>
        {% if product.in_stock %}
            <span class="badge badge-success">In Stock</span>
        {% else %}
            <span class="badge badge-danger">Out of Stock</span>
        {% endif %}
        {% if product.featured %}
            <span class="featured-badge">Featured</span>
        {% endif %}
        <div class="description">
            <h2>Description</h2>
            <p>{{ product.description | default("No description available.") }}</p>
        </div>
    </div>
{% endblock %}
```

Create src/routes/catalog.py:

```
from tina4_python.core.router import get

products = [
    {"id": 1, "name": "Espresso Machine", "category": "Kitchen", "price": 299.99, "in_stock": Tr
    {"id": 2, "name": "Yoga Mat", "category": "Fitness", "price": 29.99, "in_stock": True, "feat
    {"id": 3, "name": "Standing Desk", "category": "Office", "price": 549.99, "in_stock": True,
    {"id": 4, "name": "Noise-Canceling Headphones", "category": "Electronics", "price": 199.99,
    {"id": 5, "name": "Water Bottle", "category": "Fitness", "price": 24.99, "in_stock": True, "
    {"id": 6, "name": "Desk Lamp", "category": "Office", "price": 79.99, "in_stock": True, "feat
]


@get("/catalog")
async def catalog_page(request, response):
    category = request.params.get("category")
    categories = sorted(set(p["category"] for p in products))

    if category:
        filtered = [p for p in products if p["category"].lower() == category.lower()]
    else:
        filtered = products

    return response.render("catalog.html", {
        "products": filtered,
        "categories": categories,
        "active_category": category,
        "page_title": f"{category} Products" if category else "Product Catalog"
    })


@get("/catalog/{id:int}")
async def product_detail(id, request, response):
    product_id = id

    for product in products:
```

```
        if product["id"] == product_id:
            return response.render("product_detail.html", {"product": product})

    return response.render("errors/404.html", {}, 404)
```

Open `http://localhost:7146/catalog` **in your browser.** You should see:

- A heading "Product Catalog"
- Category filter links: All, Electronics, Fitness, Kitchen, Office
- A count of products shown
- Product cards in a grid with names, prices, category labels, and stock badges
- Featured products wear a highlighted style and a "Featured" badge
- Clicking a product name navigates to the detail page
- Clicking a category link filters the list

---

13. Gotchas

1. Whitespace in output

Problem: Rendered HTML contains unexpected blank lines or spaces.

Cause: Template tags produce whitespace on the lines they occupy.

Fix: Use whitespace control with `{%-` and `-%}` to strip whitespace around tags:

```
{%- for item in items -%}
    <li>{{ item.name }}</li>
{%- endfor -%}
```

2. Variable not defined error

Problem: Frond raises an error when a variable does not exist in the context.

Cause: You used `{{ user.name }}` but did not pass `user` in the template data.

Fix: Use the `|default` filter: `{{ user.name | default("Guest") }}`. Or check first: `{% if user is defined %}{{ user.name }}{% endif %}`.

3. Extends must be the first tag

Problem: `{% extends "base.html" %}` has no effect. The page renders without the layout.

Cause: `{% extends %}` must be the first tag in the template. Any text, HTML, or tags before it cause Frond to treat the template as standalone.

Fix: Move `{% extends "base.html" %}` to the first line. Nothing before it.

4. Macro not found

Problem: `{{ forms.input(...) }}` produces an error about `forms` being undefined.

Cause: The `{% import %}` statement is missing, or the import path is wrong.

Fix: Add `{% import "macros/forms.html" as forms %}` at the top of the template (after `{% extends %}` if using inheritance). The path is relative to `src/templates/`.

5. Filter produces wrong type

Problem: `{{ "%.2f"|format(price) }}` shows an error instead of a formatted number.

Cause: The variable `price` is a string, not a number. Filters expect specific types.

Fix: Pass correct types from your route handler. Use `float(price)` in Python before passing to the template, or convert in the template: `{{ "%.2f"|format(price|float) }}`.

6. Escaped HTML when you want raw output

Problem: HTML content shows as text with visible `<tags>` instead of rendering.

Cause: Frond auto-escapes all `{{ }}` output to prevent XSS.

Fix: Use the `|safe` filter: `{{ trusted_html | safe }}`. Only use this with content you trust -- never with user input.

7. Include file path wrong

Problem: `{% include "header.html" %}` produces a "template not found" error even though the file exists.

Cause: The path in `{% include %}` is relative to `src/templates/`, not the current template file.

Fix: Use the full path from the templates root: `{% include "partials/header.html" %}` for a file at `src/templates/partials/header.html`.

Database

1. From Lists to Real Data

Every example so far stored data in Python lists. Restart the server. Data gone. A real application demands persistence.

Tina4 Python makes database access straightforward. Set a `TINA4_DATABASE_URL` in your `.env`. Create a `Database()` connection. Run SQL. No ORM required (Chapter 6 adds that). The SQL you already know carries over unchanged.

Picture a notes application. Users create, edit, and delete notes. Those notes must survive restarts, support searching, and handle concurrent users. A database delivers all three.

---

2. Configuration

TINA4*DATABASE*URL

Set your database connection in `.env`:

```
TINA4_DATABASE_URL=sqlite:///data/app.db
```

That is the default -- a SQLite database stored in `data/`. SQLite ships with Python. No install required.

Tina4 supports six database engines:

Engine	TINA4*DATABASE*URL Format	Package Required
SQLite	`sqlite:///data/app.db`	None (stdlib)
PostgreSQL	`postgresql://user:pass@host:5432/dbname`	`psycopg2`
MySQL	`mysql://user:pass@host:3306/dbname`	`mysql-connector-python`
MSSQL	`mssql://user:pass@host:1433/dbname`	`pymssql`
Firebird	`firebird://user:pass@host:3050/path/to/db.fdb`	`firebird-driver`
ODBC	`odbc://DSN_NAME`	`pyodbc`

Switch databases by changing the URL. Your Python code stays the same. All drivers implement the same adapter interface.

Installing Database Drivers

SQLite requires nothing. For other databases, install the driver with uv:

```
# PostgreSQL
uv add psycopg2-binary

# MySQL
uv add mysql-connector-python

# MSSQL
uv add pymssql

# Firebird
uv add firebird-driver

# ODBC
uv add pyodbc
```

---

3. Creating a Connection

```
from tina4_python.database.connection import Database

db = Database()
```

One line. `Database()` reads `TINA4_DATABASE_URL` from your `.env` and connects. If the SQLite file does not exist, the adapter creates one.

You can also pass a URL directly:

```
db = Database("sqlite:///data/test.db")
```

Connection Pooling

Applications that handle many concurrent requests benefit from connection pooling. Pass a `pool` argument:

```
db = Database("postgres://localhost/mydb", pool=5)  # 5 connections, round-robin
```

The `pool` parameter controls how many database connections the pool maintains:

- `pool=0` (the default) -- a single connection serves all queries
- `pool=N` (where N > 0) -- N connections rotate round-robin across queries

Pooled connections are thread-safe. Each query dispatches to the next available connection. This eliminates contention when multiple route handlers query the database at the same time.

Extra Driver Options via **kwargs

Any additional keyword arguments pass through to the underlying driver's `connect()` call. This suits engine-specific options:

```
db = Database("firebird://localhost:3050//data/legacy.fdb", charset="ISO8859_1")
```

Firebird URL Forms

Firebird is the awkward one in the stack: every other engine has a server-side database name (`postgres://host:port/dbname`), but Firebird wants either an absolute file path on the server, a Windows drive-letter path, or an alias. The classic URI form needs a double slash to keep the leading `/` of an absolute path through `urlparse`, which is unintuitive.

Tina4 normalises five equivalent forms. Pick whichever reads best:

```
# Classic double-slash absolute path -- the URL spec way
db = Database("firebird://SYSDBA:masterkey@localhost:3050//firebird/data/app.fdb")

# Single-slash absolute path -- what most people instinctively type
db = Database("firebird://SYSDBA:masterkey@localhost:3050/firebird/data/app.fdb")

# Windows drive-letter path
db = Database("firebird://SYSDBA:masterkey@host:3050/C:/Data/app.fdb")

# Windows with URL-encoded colon
db = Database("firebird://SYSDBA:masterkey@host:3050/C%3A/Data/app.fdb")

# Firebird alias (single token, no slashes)
db = Database("firebird://SYSDBA:masterkey@localhost:3050/employee")
```

For ops setups that keep the server URL and the database location in separate config layers -- or for Windows backslash paths that fight URL encoding -- set `TINA4_DATABASE_FIREBIRD_PATH`:

```
TINA4_DATABASE_FIREBIRD_PATH=C:\firebird\data\app.fdb
TINA4_DATABASE_URL=firebird://SYSDBA:masterkey@localhost:3050/ignored
```

The env override wins over whatever path is in the URL.

Firebird Dual-Driver Support

The Firebird adapter tries `firebird-driver` (the modern package) first and falls back to the legacy `fdb` package if `firebird-driver` is missing. Install whichever is available -- the adapter handles the difference internally.

Lowercase Column Names

Firebird returns column names in uppercase by default. Tina4 normalises them to lowercase, so `result["first_name"]` works regardless of how the column was defined in the schema.

Firebird Migration Support

As of 3.10.8, the migration runner handles Firebird correctly. It uses a generator (sequence) for auto-increment IDs instead of `AUTOINCREMENT`, and emits `VARCHAR(4096)` instead of `TEXT` (which Firebird does not support as a column type). No changes to existing migrations are needed -- the runner detects the engine on its own.

---

4. Running Queries

fetch -- Get Multiple Rows

```
from tina4_python.core.router import get
from tina4_python.database.connection import Database

@get("/api/notes")
async def list_notes(request, response):
    db = Database()
    notes = db.fetch("SELECT id, title, content, created_at FROM notes ORDER BY created_at DESC"

    return response.json({"notes": notes, "count": len(notes)})
```

`fetch()` returns a `DatabaseResult` containing a list of dictionaries. Each dictionary represents one row:

```
{
  "notes": [
    {"id": 1, "title": "Shopping List", "content": "Milk, eggs, bread", "created_at": "2026-03-2
    {"id": 2, "title": "Meeting Notes", "content": "Discuss Q2 roadmap", "created_at": "2026-03-
  ],
  "count": 2
}
```

DatabaseResult

`fetch()` returns a `DatabaseResult` object. It behaves like a list but carries extra metadata about the query.

Properties

```
result = db.fetch("SELECT * FROM users WHERE active = ?", [1])

result.records      # [{"id": 1, "name": "Alice"}, {"id": 2, "name": "Bob"}]
result.columns      # ["id", "name", "email", "active"]
result.count        # total number of matching rows
result.limit        # query limit (if set)
result.offset       # query offset (if set)
```

Iteration

A `DatabaseResult` is iterable. Use it directly in a `for` loop:

```
for user in result:
    print(user["name"])
```

Index Access

Access rows by index:

```
first_user = result[0]
```

Countable

`len()` works on the result:

```
print(len(result))  # number of records in this result set
```

Conversion Methods

```
result.to_list()              # plain list of dictionaries (same as result.records)
result.to_paginate()          # {"data": [...], "total": 42, "page": 1, "per_page": 20, ...}
result.to_paginate(page=2, per_page=10)  # custom page and page size
```

`to_list()` returns the records as a plain list. `to_paginate(page=1, per_page=20)` bundles the records with total count, page, per*page, total*pages, has*next, and has*prev in a single dictionary. It is built for paginated API responses.

Schema Metadata with column_info()

`column_info()` returns detailed metadata about the columns in the result set. The data loads lazily -- it queries the database schema only when you call the method for the first time:

```
info = result.column_info()
# [
#     {"name": "id", "type": "INTEGER", "size": None, "decimals": None, "nullable": False, "prim
#     {"name": "name", "type": "TEXT", "size": None, "decimals": None, "nullable": False, "prima
#     {"name": "email", "type": "TEXT", "size": 255, "decimals": None, "nullable": True, "primar
#     ...
# ]
```

Each column entry contains:

Field	Description
`name`	Column name
`type`	Database type (e.g. `INTEGER`, `TEXT`, `REAL`)
`size`	Maximum size (or `None` if not applicable)
`decimals`	Decimal places (or `None`)
`nullable`	Whether the column allows `NULL`
`primary_key`	Whether the column is part of the primary key

This powers dynamic forms, documentation generation, and data validation before insert.

fetch_one -- Get a Single Row

```
@get("/api/notes/{id:int}")
async def get_note(id, request, response):
    db = Database()
    note = db.fetch_one(
        "SELECT id, title, content, created_at FROM notes WHERE id = ?",
        [id]
    )

    if note is None:
        return response.json({"error": "Note not found"}, 404)

    return response.json(note)
```

`fetch_one()` returns a single dictionary or `None` if no row matches.

execute -- Insert, Update, Delete

```
from tina4_python.core.router import post, put, delete
from tina4_python.database.connection import Database

@post("/api/notes")
async def create_note(request, response):
    db = Database()
    body = request.body

    if not body.get("title"):
        return response.json({"error": "Title is required"}, 400)

    result = db.execute(
        "INSERT INTO notes (title, content) VALUES (?, ?)",
        [body["title"], body.get("content", "")]
    )

    note = db.fetch_one("SELECT * FROM notes WHERE id = ?", [result.last_id])

    return response.json({"message": "Note created", "note": note}, 201)
```

`execute()` runs an INSERT, UPDATE, or DELETE statement and returns a `DatabaseResult` with `affected_rows` and `last_id` properties.

---

5. Parameterised Queries

Never concatenate user input into SQL strings. Parameterised queries stand between you and SQL injection:

```
# WRONG -- SQL injection vulnerability
db.fetch(f"SELECT * FROM notes WHERE title = '{user_input}'")

# CORRECT -- parameterised query
db.fetch("SELECT * FROM notes WHERE title = ?", [user_input])
```

Positional parameters use the `?` placeholder. Pass a list of values as the second argument:

```
db.fetch(
    "SELECT * FROM notes WHERE category = ? AND created_at > ?",
    ["work", "2026-03-01"]
)
```

Tina4 handles parameter escaping and type conversion for all database engines.

---

6. Transactions

Multiple operations must succeed or fail together. Transactions enforce that contract:

```
@post("/api/transfer")
async def transfer_funds(request, response):
    db = Database()
    body = request.body

    from_account = body["from_account"]
```

```
    to_account = body["to_account"]
    amount = float(body["amount"])

    db.start_transaction()

    try:
        # Deduct from sender
        db.execute(
            "UPDATE accounts SET balance = balance - ? WHERE id = ? AND balance >= ?",
            [amount, from_account, amount]
        )

        # Check if deduction succeeded
        sender = db.fetch_one(
            "SELECT balance FROM accounts WHERE id = ?",
            [from_account]
        )

        if sender is None:
            db.rollback()
            return response.json({"error": "Insufficient funds or account not found"}, 400)

        # Credit receiver
        db.execute(
            "UPDATE accounts SET balance = balance + ? WHERE id = ?",
            [amount, to_account]
        )

        db.commit()

        return response.json({"message": f"Transferred {amount} successfully"})

    except Exception as e:
        db.rollback()
        return response.json({"error": str(e)}, 500)
```

The three transaction methods:

- `db.start_transaction()` -- begin a transaction
- `db.commit()` -- save all changes since the transaction started
- `db.rollback()` -- undo all changes since the transaction started

Without transactions, each `execute()` call auto-commits on its own.

---

7. Batch Operations with execute_many

Inserting or updating many rows at once calls for `execute_many()`. It batches operations for speed:

```
@post("/api/notes/import")
async def import_notes(request, response):
    db = Database()
    notes = request.body.get("notes", [])

    if not notes:
```

```
        return response.json({"error": "No notes provided"}, 400)

    params_list = [
        [note["title"], note.get("content", "")]
        for note in notes
    ]

    db.execute_many(
        "INSERT INTO notes (title, content) VALUES (?, ?)",
        params_list
    )

    return response.json({"message": f"Imported {len(notes)} notes"}, 201)
```

```
curl -X POST http://localhost:7146/api/notes/import \
  -H "Content-Type: application/json" \
  -d '{"notes": [{"title": "Note 1", "content": "First"}, {"title": "Note 2", "content": "Second
```

```
{"message":"Imported 3 notes"}
```

`execute_many()` runs far faster than `execute()` in a loop. One round trip instead of many.

---

8. Insert, Update, and Delete Helpers

Tina4 provides shorthand methods that cut the boilerplate:

insert

```
result = db.insert("notes", {
    "title": "Quick Note",
    "content": "Created with insert helper"
})
# result.last_id contains the new row's ID
```

Returns a `DatabaseResult`. Access `result.last_id` for the inserted row's ID.

update

```
result = db.update("notes", {"title": "Updated Title", "content": "New content"}, "id = ?", [1])
```

The third argument is the WHERE clause, the fourth is a list of parameter values. Returns a `DatabaseResult` with `affected_rows`.

delete

```
result = db.delete("notes", "id = ?", [1])
```

Returns a `DatabaseResult` with `affected_rows`.

These helpers eliminate boilerplate INSERT/UPDATE/DELETE SQL. For complex queries, `execute()` is always available.

---

9. Schema Inspection

Tina4 exposes methods to inspect your database structure at runtime. The `Database` object delegates to the underlying adapter, so these work across all six engines.

get_tables()

```
db = Database()
tables = db.get_tables()
```

Returns a list of table names:

```
["notes", "users", "tina4_migration"]
```

get_columns()

```
columns = db.get_columns("notes")
```

Returns column definitions as a list of dictionaries:

```
[
    {"name": "id", "type": "INTEGER", "nullable": False, "default": None, "primary_key": True},
    {"name": "title", "type": "TEXT", "nullable": False, "default": None, "primary_key": False},
    {"name": "content", "type": "TEXT", "nullable": True, "default": "''", "primary_key": False}
    {"name": "created_at", "type": "TEXT", "nullable": True, "default": "CURRENT_TIMESTAMP", "pr
]
```

Each entry includes the column name, data type, whether it accepts NULL, its default value, and whether it belongs to the primary key.

table_exists()

```
if db.table_exists("notes"):
    # Table exists, safe to query
    notes = db.fetch("SELECT * FROM notes")
```

Returns `True` if the table exists, `False` otherwise.

get*database*type()

```
engine = db.get_database_type()
# "sqlite", "postgresql", "mysql", "mssql", or "firebird"
```

Returns a lowercase string identifying the active database engine. This is useful when you need engine-specific SQL in a multi-database setup.

A Schema Info Endpoint

Combine these methods to build a schema browser:

```
from tina4_python.core.router import get
from tina4_python.database.connection import Database

@get("/api/schema")
async def schema_info(request, response):
    db = Database()
    tables = db.get_tables()

    schema = {}
    for table in tables:
```

```
        schema[table] = db.get_columns(table)

    return response.json({"tables": schema})
```

```
{
  "tables": {
    "notes": [
      {"name": "id", "type": "INTEGER", "nullable": false, "default": null, "primary_key": true}
      {"name": "title", "type": "TEXT", "nullable": false, "default": null, "primary_key": false
    ],
    "users": [
      {"name": "id", "type": "INTEGER", "nullable": false, "default": null, "primary_key": true}
      {"name": "email", "type": "TEXT", "nullable": false, "default": null, "primary_key": false
    ]
  }
}
```

Schema inspection powers admin dashboards, migration generators, and dynamic form builders. The database tells you its own structure -- no guesswork required.

---

10. Migrations

Migrations are SQL files that version your database schema. Write them once. Apply them in order. Roll them back when needed.

File Naming

Migration files live in a `migrations/` directory. Two naming patterns are supported:

Pattern	Example
Sequential	`000001_create_users.sql`
Timestamp	`20260322160000_create_notes.sql`

Files sort alphabetically when they run. Pick one pattern and stick with it -- `000001_` sorts before `20260322_`, so mixing them leads to unexpected execution order.

Generating a Migration

```
tina4 generate migration create_notes_table
```

This creates two files:

```
migrations/000001_create_notes_table.sql
migrations/000001_create_notes_table.down.sql
```

The first is your "up" migration. The second is the matching "down" migration for rollbacks.

Writing the Up Migration

Open the generated `.sql` file and add your schema changes:

```
-- migrations/000001_create_notes_table.sql
CREATE TABLE notes (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
```

```
    title TEXT NOT NULL,
    content TEXT DEFAULT '',
    category TEXT DEFAULT 'general',
    created_at TEXT DEFAULT CURRENT_TIMESTAMP,
    updated_at TEXT DEFAULT CURRENT_TIMESTAMP
);

CREATE INDEX idx_notes_category ON notes (category);
CREATE INDEX idx_notes_created_at ON notes (created_at);
```

Writing the Down Migration

The `.down.sql` file undoes whatever the up migration did:

```
-- migrations/000001_create_notes_table.down.sql
DROP INDEX IF EXISTS idx_notes_created_at;
DROP INDEX IF EXISTS idx_notes_category;
DROP TABLE IF EXISTS notes;
```

Down migrations are optional. Everything works without them -- until you try to roll back. At that point Tina4 fails with a clear error about the missing file.

The `.down.sql` file must share the exact same base name as the up file. If your up file is `000001_create_notes_table.sql`, the down file must be `000001_create_notes_table.down.sql`.

Running Migrations

```
tina4 migrate
# or
tina4python migrate
```

Tina4 finds all pending `.sql` files in `migrations/`, sorts them alphabetically, and executes them in order. Each run receives a **batch number**. The batch groups every migration applied in that single run.

Checking Status

```
tina4python migrate:status
```

This shows which migrations have been applied and which are still pending. Run it before deploying to see what will execute.

Rolling Back

```
tina4python migrate:rollback
```

Rollback undoes the **entire last batch**. If your last `tina4 migrate` run applied three migration files, rollback reverts all three by running their `.down.sql` counterparts in reverse order.

Tracking Table

Tina4 creates a `tina4_migration` table in your database to track what has run:

Column	Purpose
`id`	Primary key

`migration_id`	The migration filename
`description`	Human-readable description
`batch`	Which batch this migration belonged to
`executed_at`	When it ran
`passed`	`1` if it succeeded, `0` if it failed

Failed migrations are recorded with `passed = 0`. On the next `tina4 migrate` run, the runner retries them.

Advanced SQL Splitting

Tina4's migration runner is not a naive line splitter. It correctly handles:

- `$$` **delimited blocks** -- PostgreSQL stored procedures and functions that contain semicolons
- `//` **blocks** -- alternative delimiter blocks
- `/* */` **block comments** -- skipped during splitting
- `--` **line comments** -- skipped during splitting

Write PostgreSQL stored procedures in your migration files without worrying about the runner choking on internal semicolons.

Migration Best Practices

- **One migration per change** -- do not pile multiple table changes into one file
- **Always write a down migration** -- so you can roll back cleanly
- **Never edit a migration that has been applied** -- create a new migration instead
- **Use descriptive names** -- `create_users_table`, `add_email_to_orders`, `create_category_index`
- **Pick one naming pattern** -- use either `000001_` or `YYYYMMDDHHMMSS_`, not both

---

11. Query Caching

Expensive queries that return the same result on every call deserve caching. Tina4 builds caching in. Enable it via environment variables in your `.env`:

```
TINA4_DB_CACHE=true
TINA4_DB_CACHE_TTL=30
```

When caching is active, all `fetch()` and `fetch_one()` calls cache their results. The `TINA4_DB_CACHE_TTL` value (in seconds) controls how long results stay cached. Write operations (`execute()`, `insert()`, `update()`, `delete()`) invalidate the entire cache.

To clear the cache manually (for example, after external data changes):

```
@post("/api/notes")
```

```
async def create_note(request, response):
    db = Database()
    result = db.execute(
        "INSERT INTO notes (title, content, category) VALUES (?, ?, ?)",
        [request.body["title"], request.body.get("content", ""), request.body.get("category", "g
    )

    # Cache is auto-invalidated on writes, but you can also clear manually:
    db.cache_clear()

    note = db.fetch_one("SELECT * FROM notes WHERE id = ?", [result.last_id])
    return response.json({"note": note}, 201)
```

Check cache performance with `db.cache_stats()`. It returns hits, misses, size, and TTL.

---

12. Exercise: Build a Notes App API

Build a complete notes application API with database persistence.

Requirements

- Create a migration for a `notes` table with: `id`, `title` (required), `content`, `category` (default "general"), `pinned` (boolean, default false), `created_at`, `updated_at`
- Build these endpoints:

Method	Path	Description
`GET`	`/api/notes`	List all notes. Support `?category=` and `?pinned=true` filters.
`GET`	`/api/notes/{id:int}`	Get a single note. 404 if not found.
`POST`	`/api/notes`	Create a note. Title required. Return 201.
`PUT`	`/api/notes/{id:int}`	Update a note. Return 404 if not found.
`DELETE`	`/api/notes/{id:int}`	Delete a note. Return 204.
`GET`	`/api/notes/categories`	List all distinct categories with counts.

Test with:

```
# Create notes
curl -X POST http://localhost:7146/api/notes \
  -H "Content-Type: application/json" \
  -d '{"title": "Shopping List", "content": "Milk, eggs, bread", "category": "personal"}'

curl -X POST http://localhost:7146/api/notes \
  -H "Content-Type: application/json" \
  -d '{"title": "Sprint Planning", "content": "Review backlog, assign tasks", "category": "work"

# List all
curl http://localhost:7146/api/notes

# Filter by category
curl "http://localhost:7146/api/notes?category=work"

# Filter by pinned
curl "http://localhost:7146/api/notes?pinned=true"

# Get categories
curl http://localhost:7146/api/notes/categories

# Update
curl -X PUT http://localhost:7146/api/notes/1 \
  -H "Content-Type: application/json" \
  -d '{"title": "Updated Shopping List", "content": "Milk, eggs, bread, butter"}'

# Delete
curl -X DELETE http://localhost:7146/api/notes/2
```

---

13. Solution

Migration

Generate the migration:

```
tina4 generate migration create_notes_table
```

Edit `migrations/000001_create_notes_table.sql`:

```
CREATE TABLE notes (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    title TEXT NOT NULL,
    content TEXT DEFAULT '',
    category TEXT NOT NULL DEFAULT 'general',
    pinned INTEGER NOT NULL DEFAULT 0,
    created_at TEXT DEFAULT CURRENT_TIMESTAMP,
    updated_at TEXT DEFAULT CURRENT_TIMESTAMP
);

CREATE INDEX idx_notes_category ON notes (category);
CREATE INDEX idx_notes_pinned ON notes (pinned);
```

Edit `migrations/000001_create_notes_table.down.sql`:

```
DROP INDEX IF EXISTS idx_notes_pinned;
DROP INDEX IF EXISTS idx_notes_category;
```

```
DROP TABLE IF EXISTS notes;
```

Run the migration:

```
tina4 migrate
```

Routes

Create `src/routes/notes.py`:

```
from tina4_python.core.router import get, post, put, delete
from tina4_python.database.connection import Database


@get("/api/notes")
async def list_notes(request, response):
    db = Database()
    category = request.params.get("category")
    pinned = request.params.get("pinned")

    sql = "SELECT * FROM notes"
    params = []
    conditions = []

    if category:
        conditions.append("category = ?")
        params.append(category)

    if pinned == "true":
        conditions.append("pinned = 1")

    if conditions:
        sql += " WHERE " + " AND ".join(conditions)

    sql += " ORDER BY pinned DESC, created_at DESC"

    notes = db.fetch(sql, params)

    return response.json({"notes": notes, "count": len(notes)})


@get("/api/notes/categories")
async def list_categories(request, response):
    db = Database()
    categories = db.fetch(
        "SELECT category, COUNT(*) as count FROM notes GROUP BY category ORDER BY count DESC"
    )
    return response.json({"categories": categories})


@get("/api/notes/{id:int}")
async def get_note(request, response):
    db = Database()
    note = db.fetch_one(
        "SELECT * FROM notes WHERE id = ?",
        [request.params["id"]]
    )

    if note is None:
        return response.json({"error": "Note not found"}, 404)
```

```
    return response.json(note)


@post("/api/notes")
async def create_note(request, response):
    db = Database()
    body = request.body

    if not body.get("title"):
        return response.json({"error": "Title is required"}, 400)

    result = db.execute(
        "INSERT INTO notes (title, content, category, pinned) VALUES (?, ?, ?, ?)",
        [body["title"], body.get("content", ""), body.get("category", "general"), 1 if body.get(
    )

    note = db.fetch_one("SELECT * FROM notes WHERE id = ?", [result.last_id])
    db.cache_clear()

    return response.json({"message": "Note created", "note": note}, 201)


@put("/api/notes/{id:int}")
async def update_note(request, response):
    db = Database()
    note_id = request.params["id"]
    body = request.body

    existing = db.fetch_one("SELECT * FROM notes WHERE id = ?", [note_id])
    if existing is None:
        return response.json({"error": "Note not found"}, 404)

    db.execute(
        """UPDATE notes
           SET title = ?, content = ?, category = ?,
               pinned = ?, updated_at = CURRENT_TIMESTAMP
           WHERE id = ?""",
        [
            body.get("title", existing["title"]),
            body.get("content", existing["content"]),
            body.get("category", existing["category"]),
            1 if body.get("pinned", existing["pinned"]) else 0,
            note_id
        ]
    )

    updated = db.fetch_one("SELECT * FROM notes WHERE id = ?", [note_id])
    db.cache_clear()

    return response.json({"message": "Note updated", "note": updated})


@delete("/api/notes/{id:int}")
async def delete_note(request, response):
    db = Database()
    note_id = request.params["id"]

    existing = db.fetch_one("SELECT * FROM notes WHERE id = ?", [note_id])
    if existing is None:
```

```
        return response.json({"error": "Note not found"}, 404)

    db.execute("DELETE FROM notes WHERE id = ?", [note_id])
    db.cache_clear()

    return response.json(None, 204)
```

Expected output for creating a note:

```
{
  "message": "Note created",
  "note": {
    "id": 1,
    "title": "Shopping List",
    "content": "Milk, eggs, bread",
    "category": "personal",
    "pinned": 0,
    "created_at": "2026-03-22 16:00:00",
    "updated_at": "2026-03-22 16:00:00"
  }
}
```

(Status: `201 Created`)

Expected output for categories:

```
{
  "categories": [
    {"category": "personal", "count": 1},
    {"category": "work", "count": 1}
  ]
}
```

---

14. Seeder -- Generating Test Data

Testing with an empty database tells you nothing. Testing with hand-typed rows is slow and brittle. The `FakeData` class generates realistic test data, and `seed_table()` inserts it in bulk.

FakeData

```
from tina4_python.seeder import FakeData

fake = FakeData()

fake.name()       # "Grace Lopez"
fake.email()      # "bob.anderson@demo.net"
fake.phone()      # "+1 (547) 382-9104"
fake.sentence()   # "Magna exercitation lorem ipsum dolor sit amet consectetur."
fake.paragraph()  # Four sentences of filler text
fake.integer()    # 7342
fake.decimal()    # 481.29
fake.date()       # "2023-07-14"
fake.uuid()       # "a3f1b2c4-d5e6-f7a8-b9c0-d1e2f3a4b5c6"
fake.address()    # "742 Oak Ave, Tokyo"
fake.boolean()    # True
```

Every method draws from built-in word banks -- no network calls, no external packages.

Deterministic Output

Pass a seed to get reproducible results. The same seed produces the same sequence every time:

```
fake = FakeData(seed=42)
fake.name()   # Always "Wendy White" with seed 42
fake.email()  # Always the same email with seed 42
```

Deterministic data means deterministic assertions. This matters for tests.

Seeding a Table

`seed_table()` combines `FakeData` with your database. Pass a field map -- a dictionary where each key is a column name and each value is a callable that generates data:

```
from tina4_python.seeder import FakeData, seed_table
from tina4_python.database.connection import Database

db = Database()
fake = FakeData(seed=1)

seed_table(db, "users", 100, {
    "name": fake.name,
    "email": fake.email,
    "phone": fake.phone,
    "bio": fake.sentence,
})
```

This inserts 100 rows into the `users` table. Each row calls `fake.name()`, `fake.email()`, and so on to generate its values. The function commits after all rows are inserted.

Overrides

Static values that apply to every row go in the `overrides` dictionary:

```
seed_table(db, "users", 50,
    field_map={
        "name": fake.name,
        "email": fake.email,
    },
    overrides={
        "role": "member",
        "active": 1,
    },
)
```

Every row gets `role = "member"` and `active = 1`. The field map generates the rest.

When to Use It

- Populating a development database with realistic data
- Writing integration tests that need rows in the database
- Load testing with thousands of records
- Demos and screenshots that look real without using real data

---

15. Gotchas

1. SQLite boolean quirk

Problem: Boolean values come back as `0` and `1` instead of `false` and `true` in JSON.

Cause: SQLite has no native boolean type. It stores booleans as integers.

Fix: This is expected. In your route handler, convert them: `note["pinned"] = bool(note["pinned"])`. Or handle it in the frontend. The ORM (Chapter 6) does this conversion with `BooleanField`.

2. last*insert*rowid() is SQLite-specific

Problem: `SELECT * FROM notes WHERE id = last_insert_rowid()` does not work on PostgreSQL or MySQL.

Cause: `last_insert_rowid()` is a SQLite function. Other databases use different mechanisms.

Fix: Use `db.insert()` which returns the last inserted ID regardless of engine. Or use database-specific syntax: PostgreSQL uses `RETURNING id` in the INSERT statement, MySQL uses `LAST_INSERT_ID()`.

3. String vs integer comparison

Problem: `WHERE id = ?` does not find the row even though the ID exists.

Cause: Path parameters arrive as strings by default. If `id` is `"5"` (string) and the column is an integer, some databases handle this differently.

Fix: Use typed path parameters (`{id:int}`) so the value is already an integer, or cast explicitly: `[int(request.params["id"])]`.

4. Connection not closed

Problem: After many requests, the application runs out of database connections.

Cause: You create `Database()` instances without them being cleaned up.

Fix: Tina4's `Database()` manages connection pooling internally. Creating `Database()` in each handler is fine because it reuses connections from the pool. If you see connection issues, check that you are not holding transactions open too long.

5. Migration order matters

Problem: A migration fails because it references a table that does not exist yet.

Cause: Migrations run in alphabetical order. If migration B depends on the table created by migration A, migration A must sort earlier.

Fix: Use `tina4 generate migration` which auto-generates sequential numbers. Do not mix `000001_` and `YYYYMMDDHHMMSS_` patterns in the same project -- `000001_` sorts before `20240315_`, which scrambles your intended order.

---

6. Missing down migration

Problem: `tina4python migrate:rollback` fails with an error about a missing file.

Cause: The `.down.sql` file does not exist for the migration being rolled back.

Fix: Create a `.down.sql` file with the exact same base name as the up migration. If your up file is `000001_create_users.sql`, the down file must be `000001_create_users.down.sql`. It should undo exactly what the up migration did. For `CREATE TABLE`, the down is `DROP TABLE IF EXISTS`. For `ALTER TABLE ADD COLUMN`, the down is `ALTER TABLE DROP COLUMN` (though SQLite does not support dropping columns -- in that case, recreate the table).

7. Failed migrations blocking progress

Problem: A migration failed and now `tina4 migrate` keeps retrying it.

Cause: Failed migrations are recorded in the `tina4_migration` table with `passed = 0`. Tina4 retries them on the next `migrate` run.

Fix: Fix the SQL in the migration file, then run `tina4 migrate` again. The failed migration retries. If you need to skip it entirely, update its `passed` column to `1` in the `tina4_migration` table manually -- but fix the root cause first.

8. SQL injection through string formatting

Problem: Your application is vulnerable to SQL injection attacks.

Cause: You used f-strings or string concatenation to build SQL queries with user input: `f"WHERE name = '{name}'"`.

Fix: Use parameterised queries: `"WHERE name = ?", [name]`. This is the single most important security practice for database code. Tina4 handles escaping and quoting for you.

ORM

1. From SQL to Objects

The last chapter was raw SQL. It works. It also gets repetitive. Every insert demands an INSERT statement. Every update demands an UPDATE. Every fetch maps column names to dictionary keys. Over and over.

Tina4's ORM turns database rows into Python objects. Define a model class with fields. The ORM writes the SQL. It stays SQL-first -- you can drop to raw SQL at any moment -- but for the 90% case of CRUD operations, the ORM handles the grunt work.

Picture a blog. Authors, posts, comments. Authors own many posts. Posts own many comments. Comments belong to posts. Modeling these relationships with raw SQL means JOINs and manual foreign key management. The ORM makes this declarative.

ORM at a Glance: Four Languages, One Shape

The ORM does the same job in every Tina4 book. Define a model. Save it. Query it. Each language wears its own clothes — PHP uses typed properties, Python uses field class instances, Ruby uses a DSL, Node uses config objects — but the operations line up. If you know the API in one book, you can read the others.

Defining a Model

The same `Post` model with `id`, `title`, `body`, and `created_at`:

Python — field class instances on the class body:

```
from tina4_python.orm import ORM, IntegerField, StringField, DateTimeField

class Post(ORM):
    table_name = "posts"

    id = IntegerField(primary_key=True, auto_increment=True)
    title = StringField(required=True, max_length=200)
    body = StringField(default="")
    created_at = DateTimeField()
```

PHP — native typed properties:

```
<?php
use Tina4\ORM;

class Post extends ORM
{
    public string $tableName = "posts";

    public int $id;
    public string $title;
    public string $body = "";
    public string $createdAt;
```

```
}
```

Ruby — class-level DSL declarations:

```
class Post < Tina4::ORM
  table_name "posts"

  integer_field :id, primary_key: true, auto_increment: true
  string_field :title, nullable: false, length: 200
  string_field :body, default: ""
  datetime_field :created_at
end
```

Node.js (TypeScript) — config objects in a `static fields` block:

```
import { BaseModel } from "tina4-nodejs/orm";

export default class Post extends BaseModel {
  static tableName = "posts";
  static fields = {
    id:        { type: "integer" as const, primaryKey: true, autoIncrement: true },
    title:     { type: "string"  as const, required: true, maxLength: 200 },
    body:      { type: "string"  as const, default: "" },
    createdAt: { type: "datetime" as const },
  };
}
```

Common Query Operations

Same operation, four shapes:

Operation	Python	PHP	Ruby	Node.js
Find by primary key	`Post.find_by_id(1)`	`Post::findById(1)`	`Post.find_by_id(1)`	`Post.findById(1)`
Filter by attributes	`Post.find({"title": "x"})`	`Post::find(["title" => "x"])`	`Post.find(title: "x")`	`Post.find({ title: "x" })`
Raw SQL where clause	`Post.where("title = ?", ["x"])`	`(new Post())->where("title = ?", ["x"])`	`Post.where("title = ?", ["x"])`	`Post.where("title = ?", ["x"])`
Build and save	`Post.create(title="x")`	`Post::create(["title" => "x"])`	`Post.create(title: "x")`	`Post.create({ title: "x" })`
Save an instance	`post.save()`	`$post->save()`	`post.save`	`post.save()`
Fetch every row	`Post.all()`	`(new Post())->all()`	`Post.all`	`Post.all()`

Delete a record	`post.delete()`	`$post->delete()`	`post.delete`	`post.delete()`
Count rows	`Post.count()`	`(new Post())->count()`	`Post.count`	`Post.count()`

A few details worth noting. `find()` takes attribute names and applies the field map; `where()` takes raw SQL and skips translation. PHP needs `(new Post())` for instance methods like `where()` and `all()` — the rest are static. Ruby methods drop the parentheses by convention.

For full detail on field options, relationships, eager loading, soft delete, validation, and Auto-CRUD, read the rest of this chapter — it shows the API for the language of this book.

---

2. Defining a Model

Create a model file in `src/orm/`. Every `.py` file in that directory is auto-loaded.

Create `src/orm/note.py`:

```
from tina4_python.orm import ORM, IntegerField, StringField, BooleanField, DateTimeField

class Note(ORM):
    table_name = "notes"

    id = IntegerField(primary_key=True, auto_increment=True)
    title = StringField(required=True, max_length=200)
    content = StringField(default="")
    category = StringField(default="general")
    pinned = BooleanField(default=False)
    created_at = DateTimeField()
    updated_at = DateTimeField()
```

A complete model. Here is what each piece does:

- `table_name` -- the database table this model maps to. If omitted, the ORM uses the lowercase class name (e.g. `Contact` -> `contact`).
- `primary_key=True` on a field marks it as the primary key (defaults to `id` if none is specified)
- Each field is a class-level attribute with a field type

Field Types

Field Type	Python Type	SQL Type	Description
`IntegerField`	`int`	`INTEGER`	Whole numbers
`StringField`	`str`	`VARCHAR(255)`	Text strings
`NumericField`	`float`	`REAL`	Decimal numbers
`BooleanField`	`bool`	`INTEGER` (0/1)	True/False
`DateTimeField`	`str`	`DATETIME`	Date and time
`TextField`	`str`	`TEXT`	Long text
`BlobField`	`bytes`	`BLOB`	Binary data
`ForeignKeyField`	`int`	`INTEGER`	Foreign key — auto-wires `belongs_to` and `has_many` (see *Relationships*)

Verbose names (`IntegerField`, `StringField`, `BooleanField`) are the standard. Short aliases (`IntField`, `StrField`, `BoolField`) also work.

Field Options

Option	Type	Description
`primary_key`	`bool`	Marks this field as the primary key
`required`	`bool`	Field must have a value (not None)
`default`	any	Default value when not provided
`max_length`	`int`	Maximum string length
`min_length`	`int`	Minimum string length
`min_value`	number	Minimum numeric value
`max_value`	number	Maximum numeric value
`choices`	list	Allowed values
`auto_increment`	`bool`	Auto-incrementing integer
`regex`	`str`	Pattern the value must match
`validator`	callable	Custom validation function

Field Mapping

When your Python attribute names do not match the database column names, use `field_mapping` to define the translation. `field_mapping` is a dict that maps Python attribute names to DB column names.

```
from tina4_python.orm import ORM, IntegerField, StringField

class User(ORM):
    table_name = "user_accounts"
    field_mapping = {
        "first_name": "fname",      # Python attr -> DB column
        "last_name": "lname",
        "email_address": "email",
    }

    id = IntegerField(primary_key=True, auto_increment=True)
    first_name = StringField(required=True)
    last_name = StringField(required=True)
    email_address = StringField(required=True)
```

With this mapping, `user.first_name` reads from and writes to the `fname` column. The ORM handles the conversion in both directions -- on `find_by_id()`, `save()`, `select()`, and `to_dict()`. This is useful with legacy databases or third-party schemas where you cannot rename the columns.

A common use case is Firebird or Oracle, which store column names in uppercase:

```
from tina4_python.orm import ORM, Field, StringField

class Account(ORM):
    table_name = "ACCOUNTS"
    field_mapping = {
        "account_no":   "ACCOUNTNO",
        "store_name":   "STORENAME",
        "credit_limit": "CREDITLIMIT",
    }
    account_no   = StringField()
    store_name   = StringField()
    credit_limit = Field(float, default=0.0)
```

Python code uses clean snake_case names (`account.account_no`, `account.credit_limit`). The ORM maps them to the uppercase DB columns automatically.

*get*db*column and* get*db*data

Two internal helpers make field_mapping available in custom code:

```
# Get the DB column name for a Python attribute
col = account._get_db_column("account_no")   # "ACCOUNTNO"

# Get a dict of all fields using DB column names as keys
data = account._get_db_data()
# {"ACCOUNTNO": "A001", "STORENAME": "Main Store", "CREDITLIMIT": 5000.0}
```

These are mainly used internally by `save()` and `create_table()`, but are available if you need them in custom queries.

find() vs where() -- naming convention

The two query methods have a deliberate difference in how they handle column names:

- `find(filter_dict)` uses **Python attribute names**. The ORM translates them via `field_mapping`.
- `where(filter_sql)` uses **raw DB column names** in the SQL string. No translation is done.

```
# find() -- use Python attribute names
accounts = Account.find({"account_no": "A001"})   # translates to ACCOUNTNO = ?

# where() -- use DB column names directly in the SQL
accounts = Account.where("ACCOUNTNO = ?", ["A001"])  # raw SQL, no translation
```

This means `find()` is portable across database engines, while `where()` gives you full control of the SQL.

auto_map and Case Conversion Utilities

The `auto_map` flag exists on the ORM base class for cross-language parity with the PHP and Node.js versions. In Python it is a no-op because Python convention already uses `snake_case`, which matches database column names.

For cases where you need to convert between naming conventions (for example, when serialising to a camelCase JSON API), two utility functions are available:

```
from tina4_python.orm.model import snake_to_camel, camel_to_snake

snake_to_camel("first_name")   # "firstName"
camel_to_snake("firstName")    # "first_name"
```

---

3. create_table -- Schema from Models

You can create the database table directly from your model definition:

```
Note.create_table()
```

This generates and runs the CREATE TABLE SQL based on your field definitions. It is good for development and testing. For production, use migrations (Chapter 5) for version-controlled schema changes.

```
tina4 shell
>>> from src.orm.note import Note
>>> Note.create_table()
```

---

4. CRUD Operations

save -- Create or Update

```
from tina4_python.core.router import post, put
from src.orm.note import Note

@post("/api/notes")
async def create_note(request, response):
    note = Note()
    note.title = request.body["title"]
    note.content = request.body.get("content", "")
    note.category = request.body.get("category", "general")
    note.pinned = request.body.get("pinned", False)
    note.save()

    return response({"message": "Note created", "note": note.to_dict()}, 201)
```

`save()` detects whether the record is new (INSERT) or existing (UPDATE) based on whether the primary key has a value. It returns `self` on success, so you can chain calls. It returns `False` on failure.

create -- Build and Save in One Step

When you have a dict of data ready, `create()` builds the model and saves it in one call:

```
note = Note.create({
    "title": "Quick Note",
    "content": "Created in one step",
    "category": "general"
})
```

You can also pass keyword arguments:

```
note = Note.create(title="Quick Note", content="One step", category="general")
```

find*by*id -- Fetch One Record by Primary Key

```
from tina4_python.core.router import get
from src.orm.note import Note

@get("/api/notes/{id:int}")
async def get_note(id, request, response):
    note = Note.find_by_id(id)

    if note is None:
        return response({"error": "Note not found"}, 404)

    return response(note.to_dict())
```

`find_by_id()` takes a primary key value and returns a model instance, or `None` if no row matches. If soft delete is enabled, it excludes soft-deleted records.

Use `find_or_fail()` when you want a `ValueError` raised instead of `None`:

```
note = Note.find_or_fail(id)  # Raises ValueError if not found
```

find -- Query by Filter Dict

The `find()` method accepts a dictionary of column-value pairs and returns a list of matching records:

```
# Find all notes in the "work" category
work_notes = Note.find({"category": "work"})

# Find with pagination and ordering
recent = Note.find({"pinned": True}, limit=10, order_by="created_at DESC")

# Find all records (no filter)
all_notes = Note.find()
```

where -- Query with SQL Conditions

For more complex queries, `where()` takes a SQL WHERE clause with `?` placeholders:

```
notes = Note.where("category = ?", ["work"])
```

delete -- Remove a Record

```
from tina4_python.core.router import delete as delete_route
from src.orm.note import Note

@delete_route("/api/notes/{id:int}")
async def delete_note(id, request, response):
    note = Note.find_by_id(id)

    if note is None:
        return response({"error": "Note not found"}, 404)

    note.delete()

    return response(None, 204)
```

Listing Records

```
@get("/api/notes")
async def list_notes(request, response):
    category = request.params.get("category")

    if category:
        notes = Note.where("category = ?", [category])
    else:
        notes = Note.all()

    return response({
        "notes": [note.to_dict() for note in notes],
        "count": len(notes)
    })
```

`where()` takes a WHERE clause with `?` placeholders and a list of parameters. It returns a list of model instances. `all()` fetches all records. Both support pagination:

```
# With pagination
notes = Note.where("category = ?", ["work"], limit=20, offset=40)

# Fetch all with pagination
notes = Note.all(limit=20, offset=0)
```

```
# SQL-first query -- full control over the SQL
notes = Note.select(
    "SELECT * FROM notes WHERE pinned = ? ORDER BY created_at DESC",
    [1], limit=20, offset=0
)
```

select_one -- Fetch a Single Record by SQL

When you need exactly one record from a custom SQL query:

```
note = Note.select_one("SELECT * FROM notes WHERE slug = ?", ["my-note"])
```

Returns a model instance or `None`.

load -- Populate an Existing Instance

The `load()` method fills an existing model instance from the database:

```
note = Note()
note.id = 42
note.load()  # Loads data for id=42

# Or with a filter string
note = Note()
note.load("slug = ?", ["my-note"])
```

Returns `True` if a record was found, `False` otherwise.

count -- Count Records

```
total = Note.count()
work_count = Note.count("category = ?", ["work"])
```

Respects soft delete -- only counts non-deleted records.

---

5. to*dict, to*json, and Other Serialisation

to_dict

Convert a model instance to a dictionary:

```
note = Note.find_by_id(1)

data = note.to_dict()
# {"id": 1, "title": "Shopping List", "content": "Milk, eggs", "category": "personal", "pinned":
```

The `include` parameter adds relationship data to the output (see Eager Loading below). Pass a list of relationship names:

```
# Include relationships in the dict
data = note.to_dict(include=["comments"])
```

to_json

Convert directly to a JSON string:

```
json_string = note.to_json()
# '{"id": 1, "title": "Shopping List", ...}'
```

Other Serialisation Methods

Method	Returns	Description
`to_dict(include=None)`	`dict`	Primary dict method with optional relationship includes
`to_assoc(include=None)`	`dict`	Alias for `to_dict()`
`to_object()`	`dict`	Alias for `to_dict()`
`to_json(include=None)`	`str`	JSON string
`to_array()`	`list`	Flat list of values (no keys)
`to_list()`	`list`	Alias for `to_array()`

---

6. Relationships

Tina4 ORM supports three relationship types: `has_many`, `has_one`, and `belongs_to`. Each works in two styles:

- **Imperative**: call the method on an instance when you need a one-off lookup
- **Declarative**: define the relationship as a class attribute using descriptor functions — accessed as a simple attribute, lazy-loaded on first access

Both styles support eager loading via `include=["relationship_name"]`.

ForeignKeyField — Auto-Wired Relationships

Declaring a column with `ForeignKeyField(to=OtherModel)` automatically wires both sides of the relationship. The declaring model gets a `belongs_to` accessor (the column name with `_id` stripped), and the referenced model gets a `has_many` accessor (the declaring class name lowercased with `s` appended, or whatever you pass via `related_name=`).

```
from tina4_python.orm import ORM, IntegerField, StringField, ForeignKeyField

class Author(ORM):
    table_name = "authors"
    id = IntegerField(primary_key=True, auto_increment=True)
    name = StringField(required=True)

class BlogPost(ORM):
    table_name = "posts"
    id = IntegerField(primary_key=True, auto_increment=True)
    title = StringField(required=True)
    author_id = ForeignKeyField(to=Author, related_name="posts")
```

With that single `ForeignKeyField` declaration, two accessors are auto-wired:

- `post.author` — returns the `Author` instance (belongs_to)
- `author.posts` — returns a list of `BlogPost` instances (has_many)

No manual `has_many` or `belongs_to` calls required.

```
post = BlogPost.find_by_id(1)
print(post.author.name)         # "Alice"

author = Author.find_by_id(1)
for p in author.posts:
    print(p.title)
```

has_many

An author has many posts:

Create `src/orm/author.py`:

```
from tina4_python.orm import ORM, IntegerField, StringField, DateTimeField

class Author(ORM):
    table_name = "authors"

    id = IntegerField(primary_key=True, auto_increment=True)
    name = StringField(required=True)
    email = StringField(required=True)
    bio = StringField(default="")
    created_at = DateTimeField()
```

Create `src/orm/blog_post.py`:

```
from tina4_python.orm import ORM, IntegerField, StringField, DateTimeField

class BlogPost(ORM):
    table_name = "posts"

    id = IntegerField(primary_key=True, auto_increment=True)
    author_id = IntegerField(required=True)
    title = StringField(required=True, max_length=300)
    slug = StringField(required=True)
    content = StringField(default="")
    status = StringField(default="draft", choices=["draft", "published", "archived"])
    created_at = DateTimeField()
    updated_at = DateTimeField()
```

Now use `has_many` to get an author's posts:

```
@get("/api/authors/{id:int}")
async def get_author(id, request, response):
    author = Author.find_by_id(id)

    if author is None:
        return response({"error": "Author not found"}, 404)

    posts = author.has_many(BlogPost, "author_id")

    data = author.to_dict()
    data["posts"] = [post.to_dict() for post in posts]

    return response(data)
```

```
{
  "id": 1,
  "name": "Alice",
```

```
  "email": "alice@example.com",
  "bio": "Tech writer",
  "posts": [
    {"id": 1, "title": "Getting Started with Tina4", "slug": "getting-started", "status": "publi
    {"id": 2, "title": "Advanced Routing", "slug": "advanced-routing", "status": "draft"}
  ]
}
```

has_one

A user has one profile:

```
profile = user.has_one(Profile, "user_id")
```

Returns a single model instance or `None`.

belongs_to

A post belongs to an author:

```
@get("/api/posts/{id:int}")
async def get_post(id, request, response):
    post = BlogPost.find_by_id(id)

    if post is None:
        return response({"error": "Post not found"}, 404)

    author = post.belongs_to(Author, "author_id")

    data = post.to_dict()
    data["author"] = author.to_dict() if author else None

    return response(data)
```

```
{
  "id": 1,
  "author_id": 1,
  "title": "Getting Started with Tina4",
  "slug": "getting-started",
  "content": "...",
  "status": "published",
  "author": {
    "id": 1,
    "name": "Alice",
    "email": "alice@example.com"
  }
}
```

---

7. Eager Loading

Calling relationship methods inside a loop creates the N+1 problem. Load 10 authors. Call `has_many(BlogPost, "author_id")` for each one. That fires 11 queries -- 1 for authors, 10 for posts. The page drags.

The `include` parameter on `all()`, `where()`, `find_by_id()`, and `select()` solves this. It eager-loads relationships in bulk:

```
@get("/api/authors")
async def list_authors(request, response):
    # Pass a list of relationship names — ORM batch-loads all posts in 2 queries total
    authors = Author.all(include=["posts"])

    data = []
    for author in authors:
        author_dict = author.to_dict(include=["posts"])
        data.append(author_dict)

    return response({"authors": data})
```

Without eager loading, 10 authors and their posts cost 11 queries. With eager loading: 2 queries. That is the difference between a fast page and a slow one.

Declarative Relationships with Descriptors

The imperative `has_many()`, `has_one()`, and `belongs_to()` methods called on instances work for one-off lookups. For models where relationships are always needed, declare them as class attributes using the descriptor functions imported from `tina4_python.orm`:

```
from tina4_python.orm import ORM, IntegerField, StringField, DateTimeField
from tina4_python.orm import has_many, has_one, belongs_to

class Author(ORM):
    table_name = "authors"

    id    = IntegerField(primary_key=True, auto_increment=True)
    name  = StringField(required=True)
    email = StringField(required=True)

    # Declare the relationship once on the class
    posts = has_many("BlogPost", foreign_key="author_id")


class BlogPost(ORM):
    table_name = "posts"

    id        = IntegerField(primary_key=True, auto_increment=True)
    author_id = IntegerField(required=True)
    title     = StringField(required=True)

    # Lazy-load the parent author
    author   = belongs_to("Author", foreign_key="author_id")
    # Lazy-load comments
    comments = has_many("Comment", foreign_key="post_id")
```

With declarative descriptors, accessing the relationship is a simple attribute read:

```
author = Author.find_by_id(1)
for post in author.posts:          # lazy-loads on first access
    print(post.title)

post = BlogPost.find_by_id(10)
print(post.author.name)            # lazy-loads the related Author
```

Eager loading works through the `include` parameter. Pass a list of relationship names:

```
# Eager load posts when fetching all authors
```

```
authors = Author.all(include=["posts"])

# Eager load author and comments when finding a single post
post = BlogPost.find_by_id(1, include=["author", "comments"])
```

Nested Eager Loading

Dot notation loads multiple levels deep:

```
# Load authors, their posts, and each post's comments
authors = Author.all(include=["posts", "posts.comments"])
```

Authors, their posts, and each post's comments. Three queries total instead of hundreds.

to_dict with Nested Includes

When eager loading is active, `to_dict(include=...)` embeds the related data:

```
post = BlogPost.find_by_id(1, include=["author", "comments"])
data = post.to_dict(include=["author", "comments"])
```

```
{
  "id": 1,
  "title": "Getting Started with Tina4",
  "author": {
    "id": 1,
    "name": "Alice",
    "email": "alice@example.com"
  },
  "comments": [
    {"id": 1, "body": "Great post!", "author_name": "Bob"}
  ]
}
```

---

8. Soft Delete

Sometimes a record needs to disappear from queries without leaving the database. Soft delete handles this. The row stays. A flag marks it as deleted. Queries skip it.

```
from tina4_python.orm import ORM, IntegerField, StringField, BooleanField

class Task(ORM):
    table_name = "tasks"
    soft_delete = True  # Enable soft delete

    id = IntegerField(primary_key=True, auto_increment=True)
    title = StringField(required=True)
    completed = BooleanField(default=False)
    is_deleted = IntegerField(default=0)  # Required for soft delete (0 = active, 1 = deleted)
    created_at = StringField()
```

When `soft_delete = True`, the ORM changes its behaviour:

- `task.delete()` sets `is_deleted` to `1` instead of running a DELETE query
- `Task.all()`, `Task.where()`, and `Task.find_by_id()` filter out records where `is_deleted = 1`

- `task.restore()` sets `is_deleted` back to `0` and makes the record visible again
- `task.force_delete()` permanently removes the row from the database
- `Task.with_trashed()` includes soft-deleted records in query results

Deleting and Restoring

```
# Soft delete -- sets is_deleted = 1, row stays in the database
task = Task.find_by_id(1)
task.delete()

# Restore -- sets is_deleted = 0, record is visible again
task.restore()

# Permanently delete -- removes the row, no recovery possible
task.force_delete()
```

`restore()` is the inverse of `delete()`. It sets `is_deleted` back to `0` and commits the change. The record reappears in all standard queries.

Including Soft-Deleted Records

Standard queries (`all()`, `where()`, `find_by_id()`) exclude soft-deleted records. When you need to see everything -- for admin dashboards, audit logs, or data recovery -- use `with_trashed()`:

```
# All tasks, including soft-deleted ones
all_tasks = Task.with_trashed()

# Soft-deleted tasks matching a condition
deleted_tasks = Task.with_trashed("completed = ?", [1])
```

`with_trashed()` accepts the same filter parameters as `where()`. The only difference: it ignores the `is_deleted` filter that standard queries apply.

Counting with Soft Delete

The `count()` class method respects soft delete. It only counts non-deleted records:

```
active_count = Task.count()
active_work = Task.count("category = ?", ["work"])
```

When to Use Soft Delete

Soft delete suits data that users might want to recover -- emails, documents, user accounts. It also serves audit requirements where regulations demand retention. For temporary data (sessions, cache entries, logs), hard delete keeps the table lean.

---

9. Auto-CRUD

Writing the same five REST endpoints for every model gets tedious. Auto-CRUD generates them from your model class. Define the model. Register it. Five routes appear.

The auto_crud Flag

The simplest approach -- set `auto_crud = True` on your model class:

```
class Note(ORM):
    table_name = "notes"
    auto_crud = True  # Generates REST endpoints automatically

    id = IntegerField(primary_key=True, auto_increment=True)
    title = StringField(required=True)
    content = StringField(default="")
```

The moment Python loads this class, the ORM metaclass detects `auto_crud = True` and registers it with `AutoCrud`. Five routes appear at `/api/notes` with no additional code.

Here is a more complete example with a `Product` model:

```
from tina4_python.orm import ORM, Field, IntegerField, StringField

class Product(ORM):
    table_name = "products"
    auto_crud  = True   # registers /api/products routes automatically

    id    = IntegerField(primary_key=True, auto_increment=True)
    name  = StringField(required=True)
    price = Field(float, default=0.0)
```

This registers five endpoints at `/api/products` with no route files needed.

Manual Registration

You can also register models explicitly using `AutoCrud.register()`:

```
from tina4_python.crud import AutoCrud
from src.orm.note import Note

AutoCrud.register(Note)
```

Both approaches produce the same result:

Method	Path	Description
`GET`	`/api/notes`	List all with pagination (`limit`, `offset` params)
`GET`	`/api/notes/{id}`	Get one by primary key
`POST`	`/api/notes`	Create a new record
`PUT`	`/api/notes/{id}`	Update a record
`DELETE`	`/api/notes/{id}`	Delete a record

The endpoint prefix derives from the table name. The `notes` table becomes `/api/notes`. Pass a custom prefix to change it:

```
AutoCrud.register(Note, prefix="/api/v2")
# Routes: /api/v2/notes, /api/v2/notes/{id}, etc.
```

Auto-Discovering Models

Rather than registering each model by hand, point `AutoCrud.discover()` at your models directory. It scans every `.py` file, finds ORM subclasses, and registers them all:

```
from tina4_python.crud import AutoCrud

AutoCrud.discover("src/orm", prefix="/api")
```

Every ORM model in `src/orm/` gets five REST endpoints. No route files needed.

What the Generated Routes Do

GET /api/notes returns paginated results:

```
curl "http://localhost:7146/api/notes?limit=10&offset=0"
```

```
{
  "data": [
    {"id": 1, "title": "Shopping List", "content": "Milk, eggs", "category": "personal", "pinned
    {"id": 2, "title": "Sprint Plan", "content": "Review backlog", "category": "work", "pinned":
  ],
  "total": 2,
  "limit": 10,
  "offset": 0
}
```

POST /api/notes validates input before saving:

```
curl -X POST http://localhost:7146/api/notes \
  -H "Content-Type: application/json" \
  -d '{"title": "New Note", "content": "Created via auto-CRUD"}'
```

If validation fails (for example, a required field is missing), the endpoint returns a 400 with error details:

```
{"error": "Validation failed", "detail": ["title: This field is required"]}
```

DELETE /api/notes/1 respects soft delete. If the model has `soft_delete = True`, the record is marked deleted instead of removed.

Custom Routes Alongside Auto-CRUD

Custom routes defined in `src/routes/` load before auto-CRUD routes. They take precedence. If you need special logic for one endpoint (custom validation, side effects, complex queries), define that route manually. Auto-CRUD handles the rest.

Introspection

Check which models are registered:

```
registered = AutoCrud.models()
# {"notes": <class 'Note'>, "users": <class 'User'>}
```

---

10. Cached Queries

For expensive queries that don't change often, `cached()` caches the results in memory with a TTL:

```
# Cache for 60 seconds
popular = Note.cached(
    "SELECT * FROM notes WHERE pinned = ? ORDER BY created_at DESC",
    [True], ttl=60, limit=20
)
```

Clear the cache when data changes:

```
Note.clear_cache()
```

---

11. Scopes

Scopes are reusable query filters baked into the model:

```
class BlogPost(ORM):
    table_name = "posts"

    id = IntegerField(primary_key=True, auto_increment=True)
    title = StringField(required=True)
    status = StringField(default="draft")
    created_at = DateTimeField()

    @classmethod
    def published(cls):
        return cls.where("status = ?", ["published"])

    @classmethod
    def drafts(cls):
        return cls.where("status = ?", ["draft"])

    @classmethod
    def recent(cls, days=7):
        return cls.where(
            "created_at > datetime('now', ?)",
            [f"-{days} days"]
        )
```

Use them in your routes:

```
@get("/api/posts/published")
async def published_posts(request, response):
    posts = BlogPost.published()
    return response({"posts": [p.to_dict() for p in posts]})

@get("/api/posts/recent")
async def recent_posts(request, response):
    days = int(request.params.get("days", 7))
    posts = BlogPost.recent(days)
    return response({"posts": [p.to_dict() for p in posts]})
```

You can also register scopes dynamically with the `scope()` class method:

```
BlogPost.scope("active", "status != ?", ["archived"])

# Now call it:
active_posts = BlogPost.active()
```

Scopes keep query logic in the model where it belongs. Route handlers stay thin.

---

12. Input Validation

Field definitions carry validation rules. Call `validate()` before `save()` and the ORM checks every constraint:

```
from tina4_python.orm import ORM, IntegerField, StringField, NumericField

class Product(ORM):
    table_name = "products"

    id = IntegerField(primary_key=True, auto_increment=True)
    name = StringField(required=True, min_length=2, max_length=200)
    sku = StringField(required=True, regex=r"^[A-Z]{2}-\d{4}$")  # e.g., EL-1234
    price = NumericField(required=True, min_value=0.01, max_value=999999.99)
    category = StringField(choices=["Electronics", "Kitchen", "Office", "Fitness"])
```

```
@post("/api/products")
async def create_product(request, response):
    product = Product()
    product.name = request.body.get("name")
    product.sku = request.body.get("sku")
    product.price = request.body.get("price")
    product.category = request.body.get("category")

    errors = product.validate()
    if errors:
        return response({"errors": errors}, 400)

    product.save()
    return response({"product": product.to_dict()}, 201)
```

If validation fails, `validate()` returns a list of error messages:

```
{
  "errors": [
    "name: Must be at least 2 characters",
    "sku: Must match pattern ^[A-Z]{2}-\\d{4}$",
    "price: Must be at least 0.01",
    "category: Must be one of: Electronics, Kitchen, Office, Fitness"
  ]
}
```

---

13. Exercise: Build a Blog with Relationships

Build a blog API with authors, posts, and comments.

Requirements

- Create these models:

Author: `id`, `name` (required), `email` (required), `bio`, `created_at`

Post: `id`, `author_id` (integer foreign key), `title` (required, max 300), `slug` (required), `content`, `status` (choices: draft/published/archived, default draft), `created_at`, `updated_at`

Comment: `id`, `post_id` (integer foreign key), `author_name` (required), `author_email` (required), `body` (required, min 5 chars), `created_at`

- Build these endpoints:

Method	Path	Description
`POST`	`/api/authors`	Create an author
`GET`	`/api/authors/{id:int}`	Get author with their posts
`POST`	`/api/posts`	Create a post (requires author_id)
`GET`	`/api/posts`	List published posts with author info
`GET`	`/api/posts/{id:int}`	Get post with author and comments
`POST`	`/api/posts/{id:int}/comments`	Add comment to a post

---

14. Solution

Create `src/orm/author.py`:

```
from tina4_python.orm import ORM, IntegerField, StringField, DateTimeField

class Author(ORM):
    table_name = "authors"

    id = IntegerField(primary_key=True, auto_increment=True)
    name = StringField(required=True, min_length=2)
    email = StringField(required=True)
    bio = StringField(default="")
    created_at = DateTimeField()
```

Create `src/orm/blog_post.py`:

```
from tina4_python.orm import ORM, IntegerField, StringField, DateTimeField

class BlogPost(ORM):
    table_name = "posts"

    id = IntegerField(primary_key=True, auto_increment=True)
```

```
    author_id = IntegerField(required=True)
    title = StringField(required=True, max_length=300)
    slug = StringField(required=True)
    content = StringField(default="")
    status = StringField(default="draft", choices=["draft", "published", "archived"])
    created_at = DateTimeField()
    updated_at = DateTimeField()

    @classmethod
    def published(cls):
        return cls.where("status = ?", ["published"])
```

Create `src/orm/comment.py`:

```
from tina4_python.orm import ORM, IntegerField, StringField, DateTimeField

class Comment(ORM):
    table_name = "comments"

    id = IntegerField(primary_key=True, auto_increment=True)
    post_id = IntegerField(required=True)
    author_name = StringField(required=True)
    author_email = StringField(required=True)
    body = StringField(required=True, min_length=5)
    created_at = DateTimeField()
```

Create `src/routes/blog.py`:

```
from tina4_python.core.router import get, post
from src.orm.author import Author
from src.orm.blog_post import BlogPost
from src.orm.comment import Comment


@post("/api/authors")
async def create_author(request, response):
    author = Author()
    author.name = request.body.get("name")
    author.email = request.body.get("email")
    author.bio = request.body.get("bio", "")

    errors = author.validate()
    if errors:
        return response({"errors": errors}, 400)

    author.save()
    return response({"author": author.to_dict()}, 201)


@get("/api/authors/{id:int}")
async def get_author(id, request, response):
    author = Author.find_by_id(id)

    if author is None:
        return response({"error": "Author not found"}, 404)

    posts = BlogPost.where("author_id = ?", [author.id])

    data = author.to_dict()
    data["posts"] = [p.to_dict() for p in posts]
```

```
    return response(data)


@post("/api/posts")
async def create_post(request, response):
    body = request.body

    # Verify author exists
    author = Author.find_by_id(body.get("author_id"))
    if author is None:
        return response({"error": "Author not found"}, 404)

    blog_post = BlogPost()
    blog_post.author_id = body["author_id"]
    blog_post.title = body.get("title")
    blog_post.slug = body.get("slug")
    blog_post.content = body.get("content", "")
    blog_post.status = body.get("status", "draft")

    errors = blog_post.validate()
    if errors:
        return response({"errors": errors}, 400)

    blog_post.save()
    return response({"post": blog_post.to_dict()}, 201)


@get("/api/posts")
async def list_posts(request, response):
    posts = BlogPost.published()
    data = []

    for p in posts:
        post_dict = p.to_dict()
        author = p.belongs_to(Author, "author_id")
        post_dict["author"] = author.to_dict() if author else None
        data.append(post_dict)

    return response({"posts": data, "count": len(data)})


@get("/api/posts/{id:int}")
async def get_post(id, request, response):
    blog_post = BlogPost.find_by_id(id)

    if blog_post is None:
        return response({"error": "Post not found"}, 404)

    author = blog_post.belongs_to(Author, "author_id")
    comments = blog_post.has_many(Comment, "post_id")

    data = blog_post.to_dict()
    data["author"] = author.to_dict() if author else None
    data["comments"] = [c.to_dict() for c in comments]
    data["comment_count"] = len(comments)

    return response(data)

@post("/api/posts/{id:int}/comments")
```

```
async def add_comment(id, request, response):
    blog_post = BlogPost.find_by_id(id)

    if blog_post is None:
        return response({"error": "Post not found"}, 404)

    comment = Comment()
    comment.post_id = id
    comment.author_name = request.body.get("author_name")
    comment.author_email = request.body.get("author_email")
    comment.body = request.body.get("body")

    errors = comment.validate()
    if errors:
        return response({"errors": errors}, 400)

    comment.save()
    return response({"comment": comment.to_dict()}, 201)
```

---

15. Gotchas

1. Forgetting to call save()

Problem: You set properties on a model but the database does not change.

Cause: Setting `note.title = "New Title"` only changes the Python object. The database remains unchanged until you call `note.save()`.

Fix: Call `save()` after modifying properties. Check the return value -- `save()` returns `self` on success and `False` on failure.

2. find*by*id() returns None

Problem: You call `Note.find_by_id(id)` but get `None` instead of a note object.

Cause: `find_by_id()` returns `None` when no row matches the given primary key. If soft delete is enabled, `find_by_id()` also excludes soft-deleted records.

Fix: Check for `None` after `find_by_id()`: `if note is None: return 404`. Use `find_or_fail()` if you want a `ValueError` raised instead.

3. find() vs find*by*id()

Problem: You call `Note.find(42)` expecting a single record, but get unexpected results.

Cause: `find()` takes a dict filter (`find({"id": 42})`), not a bare primary key value. For single-record lookups by primary key, use `find_by_id(42)`.

Fix: Use `find_by_id(id)` for primary key lookups. Use `find({"column": value})` for filter-based queries.

4. Circular imports with relationships

Problem: `from src.orm.post import BlogPost` in `author.py` and `from src.orm.author import Author` in `post.py` causes an `ImportError`.

Cause: Python cannot handle circular imports at module level.

Fix: Import inside the method that uses the relationship, not at the top of the file. Or pass the model class as a parameter in the route handler where you use both models.

5. to_dict() includes everything

Problem: `user.to_dict()` includes `password_hash` in the API response.

Cause: `to_dict()` includes all fields by default.

Fix: Build the response dict manually, omitting sensitive fields: `{"id": user.id, "name": user.name, "email": user.email}`. Or create a helper method on your model class that returns only safe fields.

6. Validation only runs on validate()

Problem: You call `save()` without calling `validate()` first, and invalid data gets into the database.

Cause: `save()` does not validate. This is by design -- sometimes you need to save partial data or bypass validation for bulk operations.

Fix: Call `errors = model.validate()` before `save()` in your route handlers. Or create a helper method that validates and saves in one step.

7. Foreign key not enforced

Problem: You save a post with `author_id = 999` and it succeeds, even though no author with ID 999 exists.

Cause: SQLite does not enforce foreign key constraints by default. The ORM defines the relationship through `has_many`/`belongs_to` methods, but the database itself may not enforce it.

Fix: Enable SQLite foreign keys with `PRAGMA foreign_keys = ON;` in a migration, or validate the foreign key in your route handler before saving.

8. N+1 query problem

Problem: Listing 100 authors with their posts runs 101 queries (1 for authors + 100 for posts), and the page loads slowly.

Cause: You call `author.has_many(BlogPost, "author_id")` inside a loop for each author.

Fix: Use eager loading with the `include` parameter on `all()`, `where()`, or `select()`. Or fetch all posts in a single query and group them manually:

```
authors = Author.all()
all_posts = BlogPost.select(
    "SELECT * FROM posts WHERE author_id IN (" + ",".join(str(a.id) for a in authors) + ")"
)
posts_by_author = {}
for post in all_posts:
    posts_by_author.setdefault(post.author_id, []).append(post)
```

9. Auto-CRUD endpoint conflicts

Problem: Custom route at `/api/notes/{id}` stops working after registering Auto-CRUD for the Note model.

Cause: Both routes match the same path. The first registered route wins.

Fix: Custom routes in `src/routes/` load before Auto-CRUD routes. They take precedence. If you want different behaviour, use a different path for the custom route.

10. Soft-deleted records appearing in queries

Problem: You soft-deleted a record, but queries still return it.

Cause: Soft delete requires the `soft_delete = True` flag on the model class and an `is_deleted = IntegerField(default=0)` field. Without both, soft delete is inactive.

Fix: Verify both the `soft_delete = True` flag and the `is_deleted = IntegerField(default=0)` field exist on the model. The column stores `0` for active records and `1` for deleted ones.

QueryBuilder Integration

ORM models provide a `query()` class method that returns a `QueryBuilder` pre-configured with the model's table name and database connection. This gives you a fluent API for building complex queries without writing raw SQL:

```
# Fluent query builder from ORM
results = User.query() \
    .select("id", "name", "email") \
    .where("active = ?", [True]) \
    .order_by("name") \
    .limit(50) \
    .get()

# First matching record
user = User.query() \
    .where("email = ?", ["alice@example.com"]) \
    .first()

# Count
total = User.query() \
    .where("role = ?", ["admin"]) \
    .count()

# Check existence
exists = User.query() \
    .where("email = ?", ["test@example.com"]) \
    .exists()
```

See the QueryBuilder chapter for the full fluent API including joins, grouping, having, and MongoDB support.

QueryBuilder

Every database query starts as a string. Small queries stay readable. But the moment you add optional filters, pagination, sorting, and joins, string concatenation turns your code into an unreadable mess of `if` statements and f-strings. One missed space, one misplaced comma, and the query breaks.

QueryBuilder solves this. It gives you a fluent, chainable API that assembles SQL for you. You describe what you want — columns, conditions, joins, ordering — and QueryBuilder produces the correct SQL string with properly separated parameters. No concatenation. No injection risk. No debugging whitespace.

---

1. The Factory: from_table()

Every QueryBuilder chain starts with `from_table()`. It takes a table name and an optional database connection.

```
from tina4_python.query_builder import QueryBuilder

qb = QueryBuilder.from_table("users", db)
```

The first argument is the table name. The second is your database connection — the same `Database` object you use everywhere else. If you omit the database, QueryBuilder will fall back to the global ORM database (set via `orm_bind()`). If neither exists, it raises a `RuntimeError` when you try to execute.

`from_table()` returns a fresh `QueryBuilder` instance. Every method you call on it returns the same instance, so you can chain.

---

2. Choosing Columns: select()

By default, QueryBuilder selects all columns (`*`). Use `select()` to narrow the result.

```
qb = QueryBuilder.from_table("users", db) \
    .select("id", "name", "email")
```

Pass column names as separate arguments — not a list. Each call to `select()` replaces the previous column selection.

```
# This selects only "email", not "id", "name", "email"
qb = QueryBuilder.from_table("users", db) \
    .select("id", "name") \
    .select("email")
```

If you want all columns, skip `select()` entirely. The default is `*`.

---

3. Filtering: where() and or_where()

`where()` adds a condition joined with `AND`. Use `?` placeholders for parameter values.

```
result = QueryBuilder.from_table("users", db) \
    .where("active = ?", [1]) \
    .where("age > ?", [18]) \
    .get()
```

This produces:

```
SELECT * FROM users WHERE active = ? AND age > ?
```

Parameters `[1, 18]` are passed separately to the database driver. No string interpolation. No injection.

OR conditions

Use `or_where()` when you need an OR clause.

```
result = QueryBuilder.from_table("users", db) \
    .where("role = ?", ["admin"]) \
    .or_where("role = ?", ["superadmin"]) \
    .get()
```

Produces:

```
SELECT * FROM users WHERE role = ? OR role = ?
```

The first condition in the chain never gets a connector prefix. Every subsequent `where()` adds `AND`, and every `or_where()` adds `OR`.

Conditions without parameters

Some conditions do not need parameters. Pass the condition string alone.

```
qb.where("deleted_at IS NULL")
```

---

4. Joins: join() and left_join()

`join()` adds an `INNER JOIN`. `left_join()` adds a `LEFT JOIN`. Both take a table name and an ON clause.

```
result = QueryBuilder.from_table("orders", db) \
    .select("orders.id", "users.name", "orders.total") \
    .join("users", "users.id = orders.user_id") \
    .where("orders.total > ?", [100]) \
    .get()
```

Produces:

```
SELECT orders.id, users.name, orders.total
FROM orders
INNER JOIN users ON users.id = orders.user_id
WHERE orders.total > ?
```

For optional relationships — where a matching row might not exist — use `left_join()`.

```
result = QueryBuilder.from_table("users", db) \
    .select("users.name", "profiles.avatar") \
    .left_join("profiles", "profiles.user_id = users.id") \
    .get()
```

You can chain multiple joins. They appear in the SQL in the order you add them.

```
result = QueryBuilder.from_table("orders", db) \
    .join("users", "users.id = orders.user_id") \
    .join("products", "products.id = orders.product_id") \
    .left_join("discounts", "discounts.order_id = orders.id") \
    .get()
```

---

5. Aggregation: group_by() and having()

`group_by()` groups rows by a column. Call it once per column.

```
result = QueryBuilder.from_table("orders", db) \
    .select("user_id", "COUNT(*) as order_count", "SUM(total) as revenue") \
    .group_by("user_id") \
    .get()
```

Produces:

```
SELECT user_id, COUNT(*) as order_count, SUM(total) as revenue
FROM orders
GROUP BY user_id
```

To group by multiple columns, chain multiple calls.

```
qb.group_by("user_id").group_by("status")
```

Filtering groups with having()

`having()` filters after aggregation. It works like `where()` but applies to grouped results.

```
result = QueryBuilder.from_table("orders", db) \
    .select("user_id", "COUNT(*) as order_count") \
    .group_by("user_id") \
    .having("COUNT(*) > ?", [5]) \
    .get()
```

Produces:

```
SELECT user_id, COUNT(*) as order_count
FROM orders
GROUP BY user_id
HAVING COUNT(*) > ?
```

Parameters in `having()` are kept separate from `where()` parameters internally but merged at execution time. The order is always correct.

---

6. Sorting: order_by()

`order_by()` takes a column name and optional direction as a single string.

```
result = QueryBuilder.from_table("users", db) \
    .order_by("name ASC") \
    .get()
```

Chain multiple calls for multi-column sorting. They appear in the SQL in order.

```
result = QueryBuilder.from_table("products", db) \
    .order_by("category ASC") \
    .order_by("price DESC") \
    .get()
```

Produces:

```
SELECT * FROM products ORDER BY category ASC, price DESC
```

If you omit the direction, your database's default applies (usually `ASC`).

---

7. Pagination: limit()

`limit()` sets the maximum number of rows. Pass an optional second argument for the offset.

```
# First page: 10 rows starting at row 0
result = QueryBuilder.from_table("users", db) \
    .limit(10) \
    .get()

# Second page: 10 rows starting at row 10
result = QueryBuilder.from_table("users", db) \
    .limit(10, 10) \
    .get()

# Third page
result = QueryBuilder.from_table("users", db) \
    .limit(10, 20) \
    .get()
```

When you call `get()`, the limit and offset values are passed to `db.fetch()`. If you do not call `limit()`, the default is 100 rows starting at offset 0.

Pagination pattern

A common pattern for API endpoints:

```
@get("/api/users")
async def list_users(request, response):
    page = int(request.params.get("page", 1))
    per_page = int(request.params.get("per_page", 20))
    offset = (page - 1) * per_page

    result = QueryBuilder.from_table("users", db) \
        .select("id", "name", "email") \
        .where("active = ?", [1]) \
        .order_by("name ASC") \
        .limit(per_page, offset) \
        .get()

    total = QueryBuilder.from_table("users", db) \
        .where("active = ?", [1]) \
```

```
        .count()

    return response({
        "users": result.to_list(),
        "total": total,
        "page": page,
        "per_page": per_page,
    })
```

---

8. Inspecting the SQL: to_sql()

Before executing, you can inspect the generated SQL with `to_sql()`. This is useful for debugging.

```
qb = QueryBuilder.from_table("users", db) \
    .select("id", "name") \
    .where("active = ?", [1]) \
    .order_by("name ASC") \
    .limit(10)

print(qb.to_sql())
```

Output:

```
SELECT id, name FROM users WHERE active = ? ORDER BY name ASC
```

Note that `to_sql()` does not include `LIMIT` or `OFFSET` in the string. Those values are passed as arguments to `db.fetch()` at execution time. The SQL string shows everything else — columns, joins, conditions, grouping, having, and ordering.

`to_sql()` does not execute anything. It does not require a database connection. Use it freely for logging and debugging.

---

9. Execution Methods

Four methods execute the query against the database.

get() — Multiple rows

Returns a `DatabaseResult` object. Use `.records` for the list of dicts, `.to_list()` for a plain list, or iterate directly.

```
result = QueryBuilder.from_table("users", db) \
    .where("active = ?", [1]) \
    .get()

for row in result:
    print(row["name"])
```

first() — Single row

Returns a single dict, or `None` if no rows match.

```
user = QueryBuilder.from_table("users", db) \
```

```
    .where("email = ?", ["alice@example.com"]) \
    .first()

if user:
    print(user["name"])
```

count() — Row count

Returns an integer. Internally rewrites the select to `COUNT(*) as cnt` and reads the result.

```
total = QueryBuilder.from_table("orders", db) \
    .where("status = ?", ["pending"]) \
    .count()

print(f"{total} pending orders")
```

exists() — Boolean check

Returns `True` if at least one row matches, `False` otherwise. Calls `count()` under the hood.

```
if QueryBuilder.from_table("users", db) \
    .where("email = ?", ["alice@example.com"]) \
    .exists():
    print("User exists")
```

---

10. Chaining — Building Complex Queries

Every method returns `self`, so you can chain everything into a single expression. Here is a realistic example that combines most features.

```
result = QueryBuilder.from_table("orders", db) \
    .select(
        "orders.id",
        "users.name as customer",
        "products.name as product",
        "orders.quantity",
        "orders.total",
        "orders.created_at",
    ) \
    .join("users", "users.id = orders.user_id") \
    .join("products", "products.id = orders.product_id") \
    .where("orders.status = ?", ["completed"]) \
    .where("orders.created_at > ?", ["2025-01-01"]) \
    .order_by("orders.created_at DESC") \
    .limit(50) \
    .get()
```

You can also build queries conditionally. Since each method returns the same instance, store it in a variable and add clauses as needed.

```
@get("/api/products")
async def search_products(request, response):
    qb = QueryBuilder.from_table("products", db) \
        .select("id", "name", "price", "category")

    # Apply filters only if provided
    category = request.params.get("category")
```

```
    if category:
        qb.where("category = ?", [category])

    min_price = request.params.get("min_price")
    if min_price:
        qb.where("price >= ?", [float(min_price)])

    max_price = request.params.get("max_price")
    if max_price:
        qb.where("price <= ?", [float(max_price)])

    search = request.params.get("q")
    if search:
        qb.where("name LIKE ?", [f"%{search}%"])

    # Always sort and paginate
    qb.order_by("name ASC")
    qb.limit(20)

    return response(qb.get().to_list())
```

This is where QueryBuilder shines. Without it, you would be concatenating SQL fragments with `if` checks and tracking parameter positions manually.

---

11. Using with ORM Models

If your ORM models are bound to a database via `orm_bind()`, QueryBuilder can use that connection automatically. You do not need to pass `db` explicitly.

```
from tina4_python.query_builder import QueryBuilder

# No db argument — uses the global ORM database
result = QueryBuilder.from_table("users") \
    .where("active = ?", [1]) \
    .get()
```

When you call `get()`, `first()`, `count()`, or `exists()` without a database connection, QueryBuilder checks the global ORM database. If it finds one, it uses it. If not, it raises `RuntimeError: QueryBuilder: No database connection provided.`

This means you can use QueryBuilder in the same files as your ORM models without importing or passing the database around.

When to use QueryBuilder vs ORM

Use the ORM when you are working with a single model — loading, saving, deleting records. The ORM gives you objects with attributes.

Use QueryBuilder when you need joins across tables, aggregations, complex filtering, or you want a `DatabaseResult` instead of model instances.

```
# ORM — single model operations
user = User.find(1)
user.name = "Alice"
user.save()
```

```
# QueryBuilder — cross-table query with joins
result = QueryBuilder.from_table("users", db) \
    .select("users.name", "COUNT(orders.id) as order_count") \
    .join("orders", "orders.user_id = users.id") \
    .group_by("users.name") \
    .having("COUNT(orders.id) > ?", [10]) \
    .order_by("order_count DESC") \
    .get()
```

NoSQL: MongoDB Queries

The QueryBuilder can generate MongoDB-compatible query documents with `to_mongo()`. This returns a dict containing the filter, projection, sort, limit, and skip -- ready to pass to PyMongo or any MongoDB driver.

Operator Mapping

SQL Operator	MongoDB Operator
`=`	Exact match
`!=`	`$ne`
`>`	`$gt`
`<`	`$lt`
`>=`	`$gte`
`<=`	`$lte`
`LIKE`	`$regex`
`IN`	`$in`
`IS NULL`	`$exists: false`
`IS NOT NULL`	`$exists: true`

Example

```
from tina4 import QueryBuilder

query = (
    QueryBuilder.from_table("users")
    .select("name", "email")
    .where("age > ?", [25])
    .where("status = ?", ["active"])
    .order_by("name ASC")
    .limit(10)
    .offset(5)
)

mongo = query.to_mongo()
```

The returned dict:

```
{
    "filter": {"age": {"$gt": 25}, "status": "active"},
    "projection": {"name": 1, "email": 1},
    "sort": [("name", 1)],
    "limit": 10,
    "skip": 5,
}
```

Pass it directly to PyMongo:

```
collection = db["users"]
cursor = collection.find(
    mongo["filter"],
    mongo["projection"]
).sort(mongo["sort"]).limit(mongo["limit"]).skip(mongo["skip"])
```

Gotchas

1. "select() replaced my columns"

Cause: Each call to `select()` replaces the column list. It does not append.

Fix: Pass all columns in a single `select()` call.

```
# Wrong — only selects "email"
qb.select("id", "name").select("email")

# Right — selects all three
qb.select("id", "name", "email")
```

2. "My LIMIT is not in to_sql() output"

Cause: `to_sql()` builds the SQL string but does not include `LIMIT` or `OFFSET`. Those values are passed as separate arguments to `db.fetch()` when you call `get()`.

Fix: This is expected behaviour. If you need to see the full query for debugging, print both `to_sql()` and the limit/offset values.

3. "count() returns 0 but get() returns rows"

Cause: `count()` temporarily replaces the select columns with `COUNT(*) as cnt`. If you have a `GROUP BY` clause, the count query counts groups, not total rows. The first group's count is returned, which may be unexpected.

Fix: For simple row counts without grouping, `count()` works correctly. For grouped queries, use `get()` and check the result length instead.

4. "RuntimeError: No database connection provided"

Cause: You called an execution method (`get()`, `first()`, `count()`, `exists()`) without passing a database to `from_table()` and without having called `orm_bind()`.

Fix: Either pass the database explicitly:

```
QueryBuilder.from_table("users", db).get()
```

Or ensure `orm_bind(db)` has been called in your `app.py` before the query runs.

5. "or_where() on the first condition has no effect"

Cause: The first condition in the chain never gets a connector prefix (`AND` or `OR`). Whether you use `where()` or `or_where()` for the first condition, the result is the same.

Fix: This is by design. The connector only matters from the second condition onward.

6. "Parameters are in the wrong order"

Cause: `having()` parameters are stored separately from `where()` parameters. At execution time, they are merged as `where_params + having_params`. If you mix calls in an unusual order, the parameter positions might not match your expectations.

Fix: Add all `where()` conditions before `having()` conditions. This matches the natural SQL order and keeps parameters aligned.

Exercise: Product Search API

Build a product search endpoint that:

- Accepts optional query parameters: `category`, `min_price`, `max_price`, `sort` (column name), `order` (`asc` or `desc`), `page`, `per_page`.
- Returns matching products with their category name (joined from a `categories` table).
- Includes total count and pagination metadata in the response.
- Returns an empty list (not an error) when no products match.

Solution

```
# src/routes/products.py
from tina4_python.core.router import get
from tina4_python.query_builder import QueryBuilder

ALLOWED_SORT_COLUMNS = {"name", "price", "created_at"}

@get("/api/products")
async def search_products(request, response):
    page = max(int(request.params.get("page", 1)), 1)
    per_page = min(int(request.params.get("per_page", 20)), 100)
    offset = (page - 1) * per_page

    # Base query with join
    qb = QueryBuilder.from_table("products", db) \
        .select(
            "products.id",
            "products.name",
            "products.price",
            "categories.name as category",
            "products.created_at",
        ) \
        .left_join("categories", "categories.id = products.category_id")

    # Count query — same filters, no join needed for count
    count_qb = QueryBuilder.from_table("products", db)

    # Apply filters to both queries
    category = request.params.get("category")
    if category:
        qb.where("categories.name = ?", [category])
        count_qb.join("categories", "categories.id = products.category_id")
        count_qb.where("categories.name = ?", [category])

    min_price = request.params.get("min_price")
    if min_price:
        qb.where("products.price >= ?", [float(min_price)])
        count_qb.where("products.price >= ?", [float(min_price)])

    max_price = request.params.get("max_price")
    if max_price:
        qb.where("products.price <= ?", [float(max_price)])
        count_qb.where("products.price <= ?", [float(max_price)])

    # Sorting — validate column name to prevent injection
    sort_col = request.params.get("sort", "name")
    if sort_col not in ALLOWED_SORT_COLUMNS:
        sort_col = "name"

    sort_dir = request.params.get("order", "asc").upper()
    if sort_dir not in ("ASC", "DESC"):
        sort_dir = "ASC"

    qb.order_by(f"products.{sort_col} {sort_dir}")

    # Paginate
    qb.limit(per_page, offset)

    # Execute
```

```
result = qb.get()
total = count_qb.count()

return response({
    "products": result.to_list(),
    "total": total,
    "page": page,
    "per_page": per_page,
    "pages": (total + per_page - 1) // per_page,
})
```

The sort column is validated against a whitelist. The direction is constrained to `ASC` or `DESC`. User input never touches the SQL directly — it either goes through `?` placeholders or gets checked against known-safe values. QueryBuilder handles the assembly. The route handler stays readable.

Authentication

1. Locking the Door

Every endpoint built so far is public. Anyone with the URL can read, create, update, and delete data. Fine for a tutorial. Unacceptable for production.

A real application needs to know two things: who is making the request, and whether they are allowed to make it. This chapter covers Tina4's authentication system. JWT tokens. Password hashing. Middleware-based route protection. CSRF tokens for forms. Session management.

---

2. JWT Tokens

Tina4 uses JSON Web Tokens (JWT) for authentication. A JWT is a signed string carrying a payload -- user ID, role, expiry. The server mints the token at login. The client sends it with every request. The server verifies the signature without touching the database.

Generating a Token

```
from tina4_python.auth import get_token

payload = {
    "user_id": 42,
    "email": "alice@example.com",
    "role": "admin"
}

token = get_token(payload)
```

`get_token()` signs the payload with HS256 (HMAC-SHA256) and returns a JWT string like:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VyX2lkIjo0MiwiZW1haWwiOiJhbGljZUBleGFtcGxlLmNvbSIsInJ
```

The token has three parts separated by dots: header, payload, and signature. The signature ensures the token has not been tampered with.

Token Expiry

By default, tokens expire after 60 minutes. Configure this in `.env`:

```
TINA4_TOKEN_EXPIRES_IN=60
```

The value is in **minutes**. Common settings:

Value	Duration
`15`	15 minutes
`60`	1 hour (default)
`1440`	24 hours
`10080`	7 days

Validating a Token

```
payload = Auth.valid_token(token)
# Returns the payload dict if the token is valid, None if invalid or expired
```

`valid_token()` returns the decoded payload on success, not a boolean. This lets you validate and read the token in one step. Returns `None` if the token is invalid or expired.

Reading the Payload

```
payload = Auth.get_payload(token)
```

Returns the decoded payload dictionary **without validation** -- it just decodes the token:

```
{
    "user_id": 42,
    "email": "alice@example.com",
    "role": "admin",
    "iat": 1711112600,  # issued at (Unix timestamp)
    "exp": 1711116200   # expires at (Unix timestamp)
}
```

If the token cannot be decoded, `get_payload()` returns `None`.

> ***Important:*** `get_payload()` *does not verify the signature or check expiry. Use* `valid_token()` *when you need to confirm the token is trustworthy.*

The Secret Key and Algorithm

Tina4 Python uses **HS256** (HMAC-SHA256) for JWT signing. It uses only the standard library -- zero external dependencies.

Set the secret key in `.env`:

```
TINA4_SECRET=my-super-secret-key-at-least-32-chars
```

If no `TINA4_SECRET` is set, Tina4 falls back to generating a random key at `secrets/jwt.key` on first run. Setting `TINA4_SECRET` explicitly is recommended for production so all server instances share the same key.

Keep this key secret. If someone gets it, they can forge tokens.

---

3. Password Hashing

Plain-text passwords are a liability. Tina4 provides two functions for secure password handling:

Hashing a Password

```
from tina4_python.auth import Auth

hashed = Auth.hash_password("my-secure-password")
# Returns: "$2b$12$abc123...long-hash-string..."
```

Uses PBKDF2 from the standard library -- no external dependencies. Each hash includes a random salt, so hashing the same password twice produces different results.

Checking a Password

```
is_correct = Auth.check_password("my-secure-password", stored_hash)
# Returns True if the password matches the hash
```

Registration Example

```
from tina4_python.core.router import post, noauth
from tina4_python.auth import Auth
from tina4_python.database.connection import Database

@post("/api/register")
@noauth()
async def register(request, response):
    body = request.body

    # Validate input
    if not body.get("name") or not body.get("email") or not body.get("password"):
        return response.json({"error": "Name, email, and password are required"}, 400)

    if len(body["password"]) < 8:
        return response.json({"error": "Password must be at least 8 characters"}, 400)

    db = Database()

    # Check if email already exists
    existing = db.fetch_one("SELECT id FROM users WHERE email = ?", [body["email"]])
    if existing is not None:
        return response.json({"error": "Email already registered"}, 409)

    # Hash the password
    password_hash = Auth.hash_password(body["password"])

    # Create the user
    result = db.execute(
        "INSERT INTO users (name, email, password_hash) VALUES (?, ?, ?)",
        [body["name"], body["email"], password_hash]
    )

    user = db.fetch_one("SELECT id, name, email FROM users WHERE id = ?", [db.get_last_id()])

    return response.json({
        "message": "Registration successful",
        "user": user
    }, 201)
```

```
curl -X POST http://localhost:7146/api/register \
  -H "Content-Type: application/json" \
  -d '{"name": "Alice", "email": "alice@example.com", "password": "securePass123"}'
```

```
{
  "message": "Registration successful",
  "user": {"id": 1, "name": "Alice", "email": "alice@example.com"}
}
```

---

4. The Login Flow

The complete login flow. Client sends credentials. Server validates them. Server returns a JWT token.

```
from tina4_python.core.router import post, noauth
from tina4_python.auth import Auth, get_token
from tina4_python.database.connection import Database

@post("/api/login")
@noauth()
async def login(request, response):
    body = request.body

    if not body.get("email") or not body.get("password"):
        return response.json({"error": "Email and password are required"}, 400)

    db = Database()

    # Find the user
    user = db.fetch_one(
        "SELECT id, name, email, password_hash FROM users WHERE email = ?",
        [body["email"]]
    )

    if user is None:
        return response.json({"error": "Invalid email or password"}, 401)

    # Check the password
    if not Auth.check_password(body["password"], user["password_hash"]):
        return response.json({"error": "Invalid email or password"}, 401)

    # Generate a token
    token = get_token({
        "user_id": user["id"],
        "email": user["email"],
        "name": user["name"]
    })

    return response.json({
        "message": "Login successful",
        "token": token,
        "user": {
            "id": user["id"],
            "name": user["name"],
            "email": user["email"]
        }
    })
```

Notice `@noauth`. The login endpoint must be public. You cannot require a token to get a token.

```
curl -X POST http://localhost:7146/api/login \
  -H "Content-Type: application/json" \
  -d '{"email": "alice@example.com", "password": "securePass123"}'
```

```
{
  "message": "Login successful",
  "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...",
  "user": {
```

```
    "id": 1,
    "name": "Alice",
    "email": "alice@example.com"
  }
}
```

The client stores this token (in localStorage, a cookie, or memory) and sends it with subsequent requests.

---

5. Using Tokens in Requests

The client sends the token in the `Authorization` header with the `Bearer` prefix:

```
curl http://localhost:7146/api/profile \
  -H "Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9..."
```

---

6. Protecting Routes

Auth Middleware

Create a reusable auth middleware:

```
from tina4_python.auth import Auth

async def auth_middleware(request, response, next_handler):
    auth_header = request.headers.get("Authorization", "")

    if not auth_header or not auth_header.startswith("Bearer "):
        return response.json({"error": "Authorization header required"}, 401)

    token = auth_header[7:]  # Remove "Bearer " prefix

    payload = Auth.valid_token(token)
    if payload is None:
        return response.json({"error": "Invalid or expired token"}, 401)

    request.user = payload  # Attach user data to the request

    return await next_handler(request, response)
```

Applying Middleware to Routes

```
from tina4_python.core.router import get, middleware

@get("/api/profile")
@middleware(auth_middleware)
async def profile(request, response):
    return response.json({
        "user_id": request.user["user_id"],
        "email": request.user["email"],
        "name": request.user["name"]
    })

# Without token -- 401
curl http://localhost:7146/api/profile
```

```
{"error":"Authorization header required"}
```

```
# With valid token -- 200
curl http://localhost:7146/api/profile \
  -H "Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9..."
```

```
{
  "user_id": 1,
  "email": "alice@example.com",
  "name": "Alice"
}
```

Applying Middleware to a Group

```
from tina4_python.core.router import get, group, middleware

@group("/api/admin")
@middleware(auth_middleware)
def admin_routes():

    @get("/stats")
    async def admin_stats(request, response):
        return response.json({"active_users": 42})

    @get("/logs")
    async def admin_logs(request, response):
        return response.json({"logs": []})
```

Every route in the group requires a valid token.

---

7. @noauth and @secured Decorators

As introduced in Chapter 2, these decorators control authentication at the route level.

@noauth -- Skip Authentication

Use `@noauth` for public endpoints that should bypass any global or group-level auth:

```
from tina4_python.core.router import get, post, noauth

@get("/api/public/health")
@noauth()
async def health(request, response):
    return response.json({"status": "ok"})

@post("/api/login")
@noauth()
async def login(request, response):
    # Login logic
    pass

@post("/api/register")
@noauth()
async def register(request, response):
    # Registration logic
    pass
```

@secured -- Require Authentication for GET Routes

By default, POST, PUT, PATCH, and DELETE routes are considered secured. GET routes are public. Use `@secured` to explicitly protect a GET route:

```
from tina4_python.core.router import get, secured

@get("/api/me")
@secured()
async def me(request, response):
    # This GET route requires authentication
    return response.json(request.user)
```

Role-Based Authorization

Combine auth middleware with role checks:

```
from tina4_python.auth import Auth

def require_role(role):
    async def role_middleware(request, response, next_handler):
        auth_header = request.headers.get("Authorization", "")
        if not auth_header or not auth_header.startswith("Bearer "):
            return response.json({"error": "Authorization required"}, 401)

        token = auth_header[7:]
        payload = Auth.valid_token(token)
        if payload is None:
            return response.json({"error": "Invalid or expired token"}, 401)

        if payload.get("role") != role:
            return response.json({"error": f"Forbidden. Required role: {role}"}, 403)

        request.user = payload
        return await next_handler(request, response)

    return role_middleware
```

Use it as middleware:

```
from tina4_python.core.router import delete as delete_route, middleware

@delete_route("/api/users/{id:int}")
@middleware(require_role("admin"))
async def delete_user(request, response):
    # Only admins can delete users
    return response.json({"deleted": True})
```

---

8. CSRF Protection

Traditional form-based applications (not SPAs) need CSRF protection. Tina4 provides it through form tokens.

Generating a Token

In your template, include the CSRF token in every form:

```
<form method="POST" action="/profile/update">
    {{ form_token() }}

    <div class="form-group">
        <label for="name">Name</label>
        <input type="text" name="name" id="name" value="{{ user.name }}">
    </div>

    <button type="submit">Update Profile</button>
</form>
```

`{{ form_token() }}` renders a hidden input field:

```
<input type="hidden" name="_token" value="abc123randomtoken456">
```

Validating the Token

In your route handler, check the token:

```
from tina4_python.core.router import post
from tina4_python.auth import Auth

@post("/profile/update")
async def update_profile(request, response):
    # Validate CSRF token
    if not Auth.validate_form_token(request.body.get("_token", "")):
        return response.json({"error": "Invalid form token. Please refresh and try again."}, 403

    # Process the form...
    return response.redirect("/profile")
```

The CSRF token is tied to the session and expires after a single use. A malicious site cannot forge a form submission because it cannot guess the token.

> ***Note (3.10.9):*** *Form token validation internally uses* `Auth.valid_token_static()`*, a classmethod that does not require an* `Auth` *instance. Earlier versions incorrectly called the instance method, which could fail when no request context was available. If you validate form tokens manually, prefer* `Auth.valid_token_static(token)` *for reliability.*

When to Use CSRF Tokens

Use CSRF tokens for:

- HTML forms submitted by browsers
- Any POST/PUT/DELETE from server-rendered pages

You do not need CSRF tokens for:

- API endpoints that use JWT (the Bearer token already proves the request is intentional)
- Single-page applications that use `fetch()` with custom headers

---

9. Sessions

Tina4 supports server-side sessions for storing per-user state between requests. JWTs handle API authentication. Sessions handle stateful web pages. Both work side by side.

Session Configuration

Set the session backend in `.env`:

```
# File-based sessions (default)
TINA4_SESSION_BACKEND=file

# Redis
TINA4_SESSION_BACKEND=redis
TINA4_SESSION_REDIS_HOST=localhost
TINA4_SESSION_REDIS_PORT=6379

# MongoDB
TINA4_SESSION_BACKEND=mongodb
TINA4_SESSION_REDIS_HOST=localhost
TINA4_SESSION_REDIS_PORT=27017

# Valkey
TINA4_SESSION_BACKEND=valkey
TINA4_SESSION_REDIS_HOST=localhost
TINA4_SESSION_REDIS_PORT=6379
```

File-based sessions work out of the box. No extra dependencies. For production deployments with multiple servers, use Redis or Valkey so sessions are shared across instances.

Using Sessions

Access session data via `request.session`:

```
from tina4_python.core.router import get, post

@post("/login-form")
async def login_form(request, response):
    # After validating credentials...
    request.session["user_id"] = 42
    request.session["user_name"] = "Alice"
    request.session["logged_in"] = True

    return response.redirect("/dashboard")

@get("/dashboard")
async def dashboard(request, response):
    if not request.session.get("logged_in"):
        return response.redirect("/login")

    return response.render("dashboard.html", {
        "user_name": request.session["user_name"]
    })

@post("/logout")
async def logout(request, response):
    # Clear all session data
    request.session.clear()
```

```
    return response.redirect("/login")
```

Session Options

```
TINA4_SESSION_TTL=3600       # Session lifetime in seconds (default: 3600)
TINA4_SESSION_NAME=tina4_session  # Cookie name for the session ID
```

---

10. Exercise: Build Login, Register, and Profile

Build a complete authentication system with registration, login, profile viewing, and password changing.

Requirements

- Create a `users` table migration with: `id`, `name`, `email` (unique), `password_hash`, `role` (default "user"), `created_at`
- Build these endpoints:

Method	Path	Auth	Description
`POST`	`/api/register`	@noauth	Create an account. Validate name, email, password (min 8 chars).
`POST`	`/api/login`	@noauth	Login. Return JWT token.
`GET`	`/api/profile`	secured	Get current user's profile from token.
`PUT`	`/api/profile`	secured	Update name and email.
`PUT`	`/api/profile/password`	secured	Change password. Require current password.

- Create auth middleware that extracts the user from the JWT and attaches it to `request.user`.

Test with:

```
# Register
curl -X POST http://localhost:7146/api/register \
  -H "Content-Type: application/json" \
  -d '{"name": "Alice", "email": "alice@example.com", "password": "securePass123"}'

# Login
curl -X POST http://localhost:7146/api/login \
  -H "Content-Type: application/json" \
  -d '{"email": "alice@example.com", "password": "securePass123"}'

# Save the token from login response, then:

# Get profile
curl http://localhost:7146/api/profile \
  -H "Authorization: Bearer YOUR_TOKEN_HERE"

# Update profile
curl -X PUT http://localhost:7146/api/profile \
  -H "Authorization: Bearer YOUR_TOKEN_HERE" \
  -H "Content-Type: application/json" \
  -d '{"name": "Alice Smith"}'

# Change password
curl -X PUT http://localhost:7146/api/profile/password \
  -H "Authorization: Bearer YOUR_TOKEN_HERE" \
  -H "Content-Type: application/json" \
  -d '{"current_password": "securePass123", "new_password": "evenMoreSecure456"}'

# Try with no token (should fail)
curl http://localhost:7146/api/profile
```

---

11. Solution

Migration

Create `migrations/20260322160000_create_users_table.sql`:

```
-- UP
CREATE TABLE users (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    name TEXT NOT NULL,
    email TEXT NOT NULL UNIQUE,
    password_hash TEXT NOT NULL,
    role TEXT NOT NULL DEFAULT 'user',
    created_at TEXT DEFAULT CURRENT_TIMESTAMP
);

-- DOWN
DROP TABLE IF EXISTS users;
```

```
tina4 migrate
```

Routes

Create `src/routes/auth.py`: _____

```
from tina4_python.core.router import get, post, put, noauth, middleware
from tina4_python.auth import Auth, get_token
from tina4_python.database.connection import Database


async def auth_middleware(request, response, next_handler):
    auth_header = request.headers.get("Authorization", "")

    if not auth_header or not auth_header.startswith("Bearer "):
        return response.json({"error": "Authorization required. Send: Authorization: Bearer <tok

    token = auth_header[7:]

    payload = Auth.valid_token(token)
    if payload is None:
        return response.json({"error": "Invalid or expired token. Please login again."}, 401)

    request.user = payload

    return await next_handler(request, response)


@post("/api/register")
@noauth()
async def register(request, response):
    body = request.body

    errors = []
    if not body.get("name"):
        errors.append("Name is required")
    if not body.get("email"):
        errors.append("Email is required")
    if not body.get("password"):
        errors.append("Password is required")
    elif len(body["password"]) < 8:
        errors.append("Password must be at least 8 characters")

    if errors:
        return response.json({"errors": errors}, 400)

    db = Database()

    existing = db.fetch_one("SELECT id FROM users WHERE email = ?", [body["email"]])
    if existing is not None:
        return response.json({"error": "Email already registered"}, 409)

    password_hash = Auth.hash_password(body["password"])

    result = db.execute(
        "INSERT INTO users (name, email, password_hash) VALUES (?, ?, ?)",
        [body["name"], body["email"], password_hash]
    )

    user = db.fetch_one("SELECT id, name, email, role, created_at FROM users WHERE id = ?", [db.

    return response.json({"message": "Registration successful", "user": user}, 201)


@post("/api/login")
```

```
@noauth()
async def login(request, response):
    body = request.body

    if not body.get("email") or not body.get("password"):
        return response.json({"error": "Email and password are required"}, 400)

    db = Database()

    user = db.fetch_one(
        "SELECT id, name, email, password_hash, role FROM users WHERE email = ?",
        [body["email"]]
    )

    if user is None or not Auth.check_password(body["password"], user["password_hash"]):
        return response.json({"error": "Invalid email or password"}, 401)

    token = get_token({
        "user_id": user["id"],
        "email": user["email"],
        "name": user["name"],
        "role": user["role"]
    })

    return response.json({
        "message": "Login successful",
        "token": token,
        "user": {
            "id": user["id"],
            "name": user["name"],
            "email": user["email"],
            "role": user["role"]
        }
    })


@get("/api/profile")
@middleware(auth_middleware)
async def get_profile(request, response):
    db = Database()

    user = db.fetch_one(
        "SELECT id, name, email, role, created_at FROM users WHERE id = ?",
        [request.user["user_id"]]
    )

    if user is None:
        return response.json({"error": "User not found"}, 404)

    return response.json(user)


@put("/api/profile")
@middleware(auth_middleware)
async def update_profile(request, response):
    db = Database()
    body = request.body
    user_id = request.user["user_id"]
```

```
    if body.get("email"):
        existing = db.fetch_one(
            "SELECT id FROM users WHERE email = ? AND id != ?",
            [body["email"], user_id]
        )
        if existing is not None:
            return response.json({"error": "Email already in use by another account"}, 409)

    current = db.fetch_one("SELECT * FROM users WHERE id = ?", [user_id])

    db.execute(
        "UPDATE users SET name = ?, email = ? WHERE id = ?",
        [body.get("name", current["name"]), body.get("email", current["email"]), user_id]
    )

    updated = db.fetch_one(
        "SELECT id, name, email, role, created_at FROM users WHERE id = ?",
        [user_id]
    )

    return response.json({"message": "Profile updated", "user": updated})


@put("/api/profile/password")
@middleware(auth_middleware)
async def change_password(request, response):
    db = Database()
    body = request.body
    user_id = request.user["user_id"]

    if not body.get("current_password") or not body.get("new_password"):
        return response.json({"error": "Current password and new password are required"}, 400)

    if len(body["new_password"]) < 8:
        return response.json({"error": "New password must be at least 8 characters"}, 400)

    user = db.fetch_one("SELECT password_hash FROM users WHERE id = ?", [user_id])

    if not Auth.check_password(body["current_password"], user["password_hash"]):
        return response.json({"error": "Current password is incorrect"}, 401)

    new_hash = Auth.hash_password(body["new_password"])

    db.execute(
        "UPDATE users SET password_hash = ? WHERE id = ?",
        [new_hash, user_id]
    )

    return response.json({"message": "Password changed successfully"})
```

Expected output for register:

```
{
  "message": "Registration successful",
  "user": {
    "id": 1,
    "name": "Alice",
    "email": "alice@example.com",
    "role": "user",
```

```
    "created_at": "2026-03-22 16:00:00"
  }
}
```

(Status: `201 Created`)

Expected output for profile without token:

```
{"error":"Authorization required. Send: Authorization: Bearer <token>"}
```

(Status: `401 Unauthorized`)

---

12. Gotchas

1. Token expiry confusion

Problem: Tokens that worked yesterday now return 401.

Cause: The default token lifetime is 60 minutes (`TINA4_TOKEN_EXPIRES_IN=60`). After that, the token is invalid even if the signature is correct.

Fix: Issue a new token at login. If your application needs long-lived sessions, use refresh tokens: a short-lived access token (15 minutes) paired with a long-lived refresh token (7 days) that can be used to get a new access token without re-entering credentials.

2. Secret key management

Problem: Tokens generated on one server are invalid on another, or tokens stop working after a deployment.

Cause: Each server generated its own random `secrets/jwt.key` file. Or the key file was deleted/regenerated during deployment.

Fix: Set `TINA4_SECRET` in `.env` explicitly and use the same value across all servers. Store it in your deployment secrets manager (not in version control). If the key changes, all existing tokens become invalid and users must log in again.

3. CORS with authentication

Problem: Frontend requests with the `Authorization` header fail with a CORS error, even though `CORS_ORIGINS=*` is set.

Cause: When the browser sends an `Authorization` header, it first sends a preflight `OPTIONS` request. The server must respond to the OPTIONS request with the correct CORS headers, including `Access-Control-Allow-Headers: Authorization`.

Fix: Tina4 handles preflight requests automatically. Make sure `CORS_ORIGINS` is set correctly. If it is still failing, check that you are not overriding CORS headers in middleware.

4. Storing tokens in localStorage

Problem: Your token is stolen via an XSS attack because it was stored in `localStorage`.

Cause: Any JavaScript on the page can read `localStorage`, including injected scripts from an XSS vulnerability.

Fix: Store tokens in `httpOnly` cookies when possible -- they cannot be accessed by JavaScript. For SPAs that must use `localStorage`, implement strict Content Security Policy headers and sanitize all user input.

5. Forgetting @noauth on login

Problem: Your login endpoint returns 401 -- you cannot log in because the endpoint requires authentication.

Cause: If you have global auth middleware, the login endpoint needs the `@noauth` decorator to bypass it.

Fix: Add `@noauth` to your login and register routes. Without it, users cannot authenticate because authentication is required to authenticate -- a catch-22.

6. Password hash column too short

Problem: Registration fails with a database error about the password hash being too long.

Cause: PBKDF2 hashes can be long. If your `password_hash` column is defined as `VARCHAR(50)`, it gets truncated.

Fix: Use `TEXT` for the password hash column, or at minimum `VARCHAR(255)`. Never constrain the hash length.

7. Token in URL query parameters

Problem: Tokens in URLs like `/api/profile?token=eyJ...` leak through browser history, server logs, and the Referer header.

Cause: Query parameters are visible in many places where headers are not.

Fix: Always send tokens in the `Authorization` header, never in the URL. The only exception is WebSocket connections, where the initial HTTP upgrade request cannot carry custom headers -- use a short-lived token for that case.

Sessions and Cookies

1. Remembering Your Users

JWT tokens handle APIs. Traditional web applications need more. A shopping cart that persists across pages. A flash message that appears once after a redirect. A "remember me" checkbox on the login form. These features run on sessions and cookies -- server-side state tied to a browser.

Picture an e-commerce site. A customer adds three items to their cart. Navigates to a product page. Comes back to the cart. Without sessions, the cart is empty every time. Sessions give the server memory. Each browser gets its own state, preserved across requests.

Sessions are auto-started. Every route handler receives `request.session` ready to use. No manual setup. No configuration required for the default file backend.

---

2. Session Configuration

The default backend is file-based sessions. They work out of the box with no additional dependencies.

To change the backend, set `TINA4_SESSION_BACKEND` in `.env`:

```
TINA4_SESSION_BACKEND=file
```

Available Backends

Backend	Value	Package Required	Best For
File	`file`	None	Development, single server
Redis	`redis`	`redis`	Production, multi-server
MongoDB	`mongodb`	`pymongo`	Production, document stores
Valkey	`valkey`	`valkey`	Production, Redis alternative
Database	`database`	None	Production, using existing DB

Redis Configuration

```
TINA4_SESSION_BACKEND=redis
TINA4_SESSION_REDIS_HOST=localhost
TINA4_SESSION_REDIS_PORT=6379
TINA4_SESSION_REDIS_PASSWORD=
```

Install the Redis driver:

```
uv add redis
```

MongoDB Configuration

```
TINA4_SESSION_BACKEND=mongodb
TINA4_SESSION_REDIS_HOST=localhost
TINA4_SESSION_REDIS_PORT=27017
TINA4_SESSION_MONGO_DB=tina4_sessions
```

Valkey Configuration

```
TINA4_SESSION_BACKEND=valkey
TINA4_SESSION_REDIS_HOST=localhost
TINA4_SESSION_REDIS_PORT=6379
```

Database Sessions

```
TINA4_SESSION_BACKEND=database
```

Stores sessions in the `tina4_session` table using your existing database connection (`TINA4_DATABASE_URL`). The table is auto-created on first use.

Session Lifetime

```
TINA4_SESSION_TTL=3600  # 1 hour in seconds (default)
```

Session Cookie

The session cookie is named `tina4_session`. It is `HttpOnly` and `SameSite=Lax` by default. Tina4 manages it automatically -- you never need to set or read it yourself.

---

3. The Session API

Access session data through `request.session`. It is available in every route handler with zero configuration.

Full API Reference

Method	Description
`request.session.set(key, value)`	Store a value
`request.session.get(key, default)`	Retrieve a value (with optional default)
`request.session.delete(key)`	Remove a key
`request.session.has(key)`	Check if a key exists
`request.session.clear()`	Remove all session data
`request.session.destroy()`	Destroy the session entirely
`request.session.save()`	Persist session data (auto-called after response)
`request.session.regenerate()`	Generate a new session ID, preserve data
`request.session.flash(key, value)`	Set flash data (one-time read)
`request.session.flash(key)`	Read and remove flash data
`request.session.all()`	Get all session data as a dictionary

---

4. Reading and Writing Session Data

Writing to the Session

```
from tina4_python.core.router import post

@post("/login-form")
async def login_form(request, response):
    # After validating credentials...
    request.session.set("user_id", 42)
    request.session.set("user_name", "Alice")
    request.session.set("role", "admin")
    request.session.set("logged_in", True)

    return response.redirect("/dashboard")
```

Reading from the Session

```
from tina4_python.core.router import get

@get("/dashboard")
async def dashboard(request, response):
    if not request.session.get("logged_in"):
        return response.redirect("/login")

    return response.render("dashboard.html", {
        "user_name": request.session.get("user_name"),
        "role": request.session.get("role")
    })
```

Deleting Session Data

```
# Delete a single key
request.session.delete("temp_data")

# Clear all session data (logout)
@post("/logout")
async def logout(request, response):
    request.session.clear()
    return response.redirect("/login")
```

Checking if a Key Exists

```
if request.session.has("user_id"):
    user_id = request.session.get("user_id")
else:
    user_id = None

# Or use .get() with a default
user_id = request.session.get("user_id", None)
```

Getting All Session Data

```
all_data = request.session.all()
```

---

5. Flash Messages

Flash messages are session values that exist for exactly one read. Set them before a redirect. Read them on the next request. They vanish automatically after being read.

Setting a Flash Message

```
@post("/api/contact")
async def submit_contact(request, response):
    # Process the form...

    request.session.flash("message", "Your message has been sent!")
    request.session.flash("message_type", "success")

    return response.redirect("/contact")
```

Reading a Flash Message

```
@get("/contact")
async def contact_page(request, response):
    flash_message = request.session.flash("message")
    flash_type = request.session.flash("message_type") or "info"

    return response.render("contact.html", {
        "flash_message": flash_message,
        "flash_type": flash_type
    })
```

Calling `request.session.flash(key)` with only a key reads the value and removes it in one step. The next time the user visits `/contact`, the flash message will be gone.

Flash Messages in Templates

```
{% if flash_message %}
    <div class="alert alert-{{ flash_type }}">
        {{ flash_message }}
    </div>
{% endif %}
```

---

6. Cookies

Cookies are small data fragments stored in the browser. Unlike sessions, cookies travel with every request and are visible to JavaScript (unless `httpOnly` is set).

Setting a Cookie

```
from tina4_python.core.router import post

@post("/preferences")
async def save_preferences(request, response):
    theme = request.body.get("theme", "light")
    language = request.body.get("language", "en")

    return response.cookie("theme", theme, {
        "max_age": 365 * 24 * 60 * 60,  # 1 year
        "path": "/",
        "samesite": "Lax"
    }).cookie("language", language, {
        "max_age": 365 * 24 * 60 * 60,
        "path": "/",
        "samesite": "Lax"
    }).json({"message": "Preferences saved"})
```

Reading a Cookie

```
from tina4_python.core.router import get

@get("/")
async def home(request, response):
    theme = request.cookies.get("theme", "light")
    language = request.cookies.get("language", "en")

    return response.render("home.html", {
        "theme": theme,
        "language": language
    })
```

Deleting a Cookie

Set `max_age` to 0:

```
@post("/clear-preferences")
async def clear_preferences(request, response):
    return response.cookie("theme", "", {"max_age": 0, "path": "/"}) \
                   .cookie("language", "", {"max_age": 0, "path": "/"}) \
                   .json({"message": "Preferences cleared"})
```

Cookie Options

Option	Type	Description
`max_age`	int	Lifetime in seconds. 0 = delete. Omit = session cookie (deleted when browser closes).
`path`	str	URL path scope. `/` means the whole site. `/admin` means only under `/admin`.
`domain`	str	Domain scope. `.example.com` includes subdomains.
`secure`	bool	Only send over HTTPS. Always `True` in production.
`httponly`	bool	Not accessible via JavaScript. Use for session cookies.
`samesite`	str	`"Strict"`, `"Lax"`, or `"None"`. Controls cross-site sending.

When to Use Cookies vs Sessions

Use Case	Use Cookies	Use Sessions
User preferences (theme, language)	Yes	No
Shopping cart	No	Yes
Authentication state	No (use JWT in header)	Yes (for form-based auth)
Flash messages	No	Yes
Tracking consent	Yes	No
Sensitive data (user ID, role)	No	Yes

The rule: sensitive data or data the client should not see goes in sessions. Non-sensitive data that should persist beyond session expiry goes in cookies.

---

7. Remember Me

The "remember me" pattern extends the session lifetime when the user checks a box on the login form:

```
from tina4_python.core.router import post
from tina4_python.auth import Auth, get_token
from tina4_python.database.connection import Database

@post("/login-form")
async def login_form(request, response):
    body = request.body
    db = Database()

    user = db.fetch_one(
        "SELECT id, name, email, password_hash FROM users WHERE email = ?",
        [body.get("email")]
    )

    if user is None or not Auth.check_password(body.get("password", ""), user["password_hash"]):
        request.session.flash("message", "Invalid email or password")
        request.session.flash("message_type", "error")
        return response.redirect("/login")

    # Set session data
    request.session.set("user_id", user["id"])
    request.session.set("user_name", user["name"])
    request.session.set("logged_in", True)

    # Handle "remember me"
    remember = body.get("remember_me") == "on"

    if remember:
        # Generate a long-lived token
        remember_token = get_token({"user_id": user["id"]})

        # Store it in a cookie that lasts 30 days
        return response.cookie("remember_token", remember_token, {
            "max_age": 30 * 24 * 60 * 60,
            "httponly": True,
            "secure": True,
            "path": "/",
            "samesite": "Lax"
        }).redirect("/dashboard")

    return response.redirect("/dashboard")
```

Then, on every page load, check for the remember token if the session has expired:

```
async def remember_me_middleware(request, response, next_handler):
    if not request.session.get("logged_in"):
        remember_token = request.cookies.get("remember_token")

        if remember_token and Auth.valid_token(remember_token):
            payload = Auth.get_payload(remember_token)
            db = Database()
            user = db.fetch_one(
                "SELECT id, name FROM users WHERE id = ?",
                [payload["user_id"]]
```

```
            )

            if user:
                request.session.set("user_id", user["id"])
                request.session.set("user_name", user["name"])
                request.session.set("logged_in", True)

    return await next_handler(request, response)
```

---

8. Session Security

Regenerate Session ID After Login

To prevent session fixation attacks, regenerate the session ID after login:

```
@post("/login-form")
async def login_form(request, response):
    # After validating credentials...

    # Regenerate session ID (prevents session fixation)
    request.session.regenerate()

    request.session.set("user_id", user["id"])
    request.session.set("logged_in", True)

    return response.redirect("/dashboard")
```

Destroy a Session

To completely destroy a session (not just clear its data):

```
@post("/logout")
async def logout(request, response):
    request.session.destroy()
    return response.redirect("/login")
```

Secure Cookie Settings

The session cookie (`tina4_session`) is `HttpOnly` and `SameSite=Lax` by default. For production, ensure HTTPS:

```
TINA4_SESSION_SECURE=true     # Only send over HTTPS
TINA4_SESSION_HTTPONLY=true   # Not accessible via JavaScript (default)
TINA4_SESSION_SAMESITE=Lax   # Prevent CSRF via cross-site requests (default)
```

`TINA4_SESSION_SAMESITE` controls cross-site cookie behavior:

Value	Behavior
`Strict`	Never sent with cross-site requests. Safest. Breaks some flows (clicking links from email).
`Lax`	Sent with top-level navigations (clicking links) but not cross-site API calls. Good default.

`None`	Always sent. Requires `TINA4_SESSION_SECURE=true`. Only for cross-site cookie access.

---

9. Exercise: Build a Shopping Cart

Build a shopping cart using sessions.

Requirements

- Create these endpoints:

Method	Path	Description
`GET`	`/cart`	View cart contents (rendered HTML page)
`POST`	`/cart/add`	Add an item to the cart
`POST`	`/cart/update`	Update item quantity
`POST`	`/cart/remove`	Remove an item from the cart
`POST`	`/cart/clear`	Clear the entire cart
`GET`	`/cart/api`	Get cart as JSON (for AJAX)

- The cart is stored in the session via `request.session.set("cart", [...])`
- Each cart item has: `product_id`, `name`, `price`, `quantity`
- Adding an existing product increments its quantity
- The cart page shows items, quantities, line totals, and a grand total
- Use flash messages for feedback ("Item added", "Cart cleared", etc.)

Use this product data for validation:

```
PRODUCTS = {
    1: {"name": "Wireless Keyboard", "price": 79.99},
    2: {"name": "USB-C Hub", "price": 49.99},
    3: {"name": "Monitor Stand", "price": 129.99},
    4: {"name": "Mechanical Mouse", "price": 59.99},
    5: {"name": "Desk Lamp", "price": 39.99}
}
```

Test with:

```
# Add items
curl -X POST http://localhost:7146/cart/add \
  -H "Content-Type: application/json" \
  -d '{"product_id": 1, "quantity": 2}' \
  -c cookies.txt -b cookies.txt

curl -X POST http://localhost:7146/cart/add \
  -H "Content-Type: application/json" \
  -d '{"product_id": 3, "quantity": 1}' \
  -c cookies.txt -b cookies.txt

# View cart
curl http://localhost:7146/cart/api -b cookies.txt

# Update quantity
curl -X POST http://localhost:7146/cart/update \
  -H "Content-Type: application/json" \
  -d '{"product_id": 1, "quantity": 3}' \
  -c cookies.txt -b cookies.txt

# Remove item
curl -X POST http://localhost:7146/cart/remove \
  -H "Content-Type: application/json" \
  -d '{"product_id": 3}' \
  -c cookies.txt -b cookies.txt

# Clear cart
curl -X POST http://localhost:7146/cart/clear -c cookies.txt -b cookies.txt
```

---

10. Solution

Create `src/routes/cart.py`:

```
from tina4_python.core.router import get, post

PRODUCTS = {
    1: {"name": "Wireless Keyboard", "price": 79.99},
    2: {"name": "USB-C Hub", "price": 49.99},
    3: {"name": "Monitor Stand", "price": 129.99},
    4: {"name": "Mechanical Mouse", "price": 59.99},
    5: {"name": "Desk Lamp", "price": 39.99}
}


def get_cart(session):
    return session.get("cart", [])


def save_cart(session, cart):
    session.set("cart", cart)


def calculate_totals(cart):
    for item in cart:
        item["line_total"] = round(item["price"] * item["quantity"], 2)
```

```
    grand_total = round(sum(item["line_total"] for item in cart), 2)
    return cart, grand_total


@get("/cart")
async def view_cart(request, response):
    cart = get_cart(request.session)
    cart, grand_total = calculate_totals(cart)
    flash_message = request.session.flash("message")
    flash_type = request.session.flash("message_type") or "info"

    return response.render("cart.html", {
        "cart": cart,
        "grand_total": grand_total,
        "item_count": sum(item["quantity"] for item in cart),
        "flash_message": flash_message,
        "flash_type": flash_type
    })


@get("/cart/api")
async def cart_api(request, response):
    cart = get_cart(request.session)
    cart, grand_total = calculate_totals(cart)

    return response.json({
        "cart": cart,
        "grand_total": grand_total,
        "item_count": sum(item["quantity"] for item in cart)
    })


@post("/cart/add")
async def add_to_cart(request, response):
    body = request.body
    product_id = body.get("product_id")
    quantity = int(body.get("quantity", 1))

    if product_id not in PRODUCTS:
        return response.json({"error": "Product not found"}, 404)

    if quantity < 1:
        return response.json({"error": "Quantity must be at least 1"}, 400)

    product = PRODUCTS[product_id]
    cart = get_cart(request.session)

    # Check if product already in cart
    for item in cart:
        if item["product_id"] == product_id:
            item["quantity"] += quantity
            save_cart(request.session, cart)
            request.session.flash("message", f"Updated {product['name']} quantity")
            request.session.flash("message_type", "success")
            return response.json({"message": f"Updated {product['name']} quantity", "cart": cart

    # Add new item
    cart.append({
        "product_id": product_id,
```

```
        "name": product["name"],
        "price": product["price"],
        "quantity": quantity
    })
    save_cart(request.session, cart)

    request.session.flash("message", f"Added {product['name']} to cart")
    request.session.flash("message_type", "success")

    return response.json({"message": f"Added {product['name']}", "cart": cart}, 201)


@post("/cart/update")
async def update_cart(request, response):
    body = request.body
    product_id = body.get("product_id")
    quantity = int(body.get("quantity", 1))

    cart = get_cart(request.session)

    for item in cart:
        if item["product_id"] == product_id:
            if quantity < 1:
                cart.remove(item)
                request.session.flash("message", f"Removed {item['name']} from cart")
            else:
                item["quantity"] = quantity
                request.session.flash("message", f"Updated {item['name']} quantity to {quantity}
            request.session.flash("message_type", "success")
            save_cart(request.session, cart)
            return response.json({"message": "Cart updated", "cart": cart})

    return response.json({"error": "Product not in cart"}, 404)


@post("/cart/remove")
async def remove_from_cart(request, response):
    product_id = request.body.get("product_id")
    cart = get_cart(request.session)

    for item in cart:
        if item["product_id"] == product_id:
            cart.remove(item)
            save_cart(request.session, cart)
            request.session.flash("message", f"Removed {item['name']} from cart")
            request.session.flash("message_type", "success")
            return response.json({"message": f"Removed {item['name']}", "cart": cart})

    return response.json({"error": "Product not in cart"}, 404)


@post("/cart/clear")
async def clear_cart(request, response):
    save_cart(request.session, [])
    request.session.flash("message", "Cart cleared")
    request.session.flash("message_type", "info")
    return response.json({"message": "Cart cleared", "cart": []})
```

Expected output for cart API after adding items:

```
{
  "cart": [
    {"product_id": 1, "name": "Wireless Keyboard", "price": 79.99, "quantity": 2, "line_total":
    {"product_id": 3, "name": "Monitor Stand", "price": 129.99, "quantity": 1, "line_total": 129
  ],
  "grand_total": 289.97,
  "item_count": 3
}
```


---

11. Gotchas

1. Session lost between requests

Problem: Data you stored in the session is gone on the next request.

Cause: The session cookie is not being sent back by the client. This happens when using curl without `-c cookies.txt -b cookies.txt`, or when the browser blocks cookies.

Fix: For curl testing, use `-c cookies.txt -b cookies.txt` to save and send cookies. In the browser, make sure cookies are not blocked for your site.

2. Session data is not JSON-serializable

Problem: Storing a Python object (like a datetime or a custom class) in the session causes an error.

Cause: Sessions are serialized to JSON (or pickled). Complex Python objects cannot be serialized directly.

Fix: Convert objects to strings or dictionaries before storing: `request.session.set("login_time", datetime.now().isoformat())`.

3. Cookie not sent on cross-origin requests

Problem: Your frontend at `http://localhost:3000` calls your API at `http://localhost:7146`, but cookies are not included.

Cause: Browsers do not send cookies cross-origin by default. You need both CORS configuration and explicit `credentials: "include"` in fetch.

Fix: Set `CORS_CREDENTIALS=true` in `.env` and use `fetch(url, { credentials: "include" })` in JavaScript. Also set `CORS_ORIGINS` to the specific frontend origin (not `*` -- wildcard does not work with credentials).

4. File-based sessions on multi-server deployments

Problem: A user is logged in on server A but logged out on server B.

Cause: File-based sessions are stored on the local filesystem. Each server has its own session files.

Fix: Switch to Redis, Valkey, MongoDB, or database-backed sessions so all servers share the same session store.

5. Session cookie overwritten by another app

Problem: Your session keeps getting reset even though you set it correctly.

Cause: Another application on the same domain uses the same session cookie name.

Fix: The session cookie name (`tina4_session`) is managed by Tina4. If you need to change it, set `TINA4_SESSION_NAME=myapp_session` in `.env`.

6. Secure cookies on localhost

Problem: Setting `"secure": True` on a cookie means it never gets sent during development.

Cause: Secure cookies are only sent over HTTPS. `http://localhost` is not HTTPS.

Fix: Do not set `"secure": True` during development. Use environment-based configuration: `"secure": os.getenv("TINA4_DEBUG") != "true"`.

7. Flash messages shown twice

Problem: A flash message appears on the page, but when you refresh, it shows again.

Cause: You read the flash message but did not remove it. Using `request.session.get()` reads without deleting.

Fix: Use `request.session.flash("message")` to read flash data. It reads and deletes in one step. Do not use `request.session.get()` for flash messages.

Middleware

1. The Pipeline Pattern

Every HTTP request passes through a series of gates before reaching your route handler. Rate limiter. Body parser. Auth check. Logger. These gates are middleware -- code that wraps your route handler and runs before, after, or both.

Picture a public API. Every request hits a rate limit check. Some endpoints require an API key. All responses need CORS headers. Errors need logging. Without middleware, that logic lives in every handler. Duplicated. Scattered. Fragile. With middleware, you write it once and attach it where it belongs.

Tina4 Python ships with built-in middleware (CORS, rate limiting) and lets you write your own. This chapter covers both.

---

2. Built-In Middleware

Tina4 Python ships with two built-in middleware classes in `tina4_python.core.middleware`: `CorsMiddleware` and `RateLimiter`. Both are configured via environment variables.

CorsMiddleware

CORS (Cross-Origin Resource Sharing) controls which websites can call your API from a browser. When React at `http://localhost:3000` calls your Tina4 API at `http://localhost:7146`, the browser sends a preflight `OPTIONS` request first. Wrong headers and the browser blocks everything.

Configure via `.env`:

```
TINA4_CORS_ORIGINS=https://app.example.com,https://admin.example.com
TINA4_CORS_METHODS=GET,POST,PUT,PATCH,DELETE,OPTIONS
TINA4_CORS_HEADERS=Content-Type,Authorization,X-Request-ID
TINA4_CORS_MAX_AGE=86400
TINA4_CORS_CREDENTIALS=true
```

With these settings, only `app.example.com` and `admin.example.com` can make cross-origin requests to your API. The browser automatically handles preflight `OPTIONS` requests.

For development, you can allow all origins:

```
TINA4_CORS_ORIGINS=*
```

The CORS middleware is active by default. You do not need to register it manually.

You can also use `CorsMiddleware` programmatically in your own middleware:

```
from tina4_python.core.middleware import CorsMiddleware

cors = CorsMiddleware()

# Check if an origin is allowed
```

```
allowed = cors.allowed_origin("https://app.example.com")

# Apply CORS headers to a response
cors.apply(request, response)

# Check if this is a preflight request
if cors.is_preflight(request):
    # Handle OPTIONS request
    pass
```

RateLimiter

The rate limiter prevents abuse by limiting how many requests a single IP can make in a sliding time window. It uses an in-memory store that is thread-safe.

Configure via `.env`:

```
TINA4_RATE_LIMIT=60
TINA4_RATE_WINDOW=60
```

This allows 60 requests per 60-second window per IP address. When a client exceeds the limit, they receive a `429 Too Many Requests` response with a `Retry-After` header.

Rate limit headers are included in every response:

```
X-RateLimit-Limit: 60
X-RateLimit-Remaining: 57
X-RateLimit-Reset: 60
```

You can use the `RateLimiter` class directly for custom rate limiting logic:

```
from tina4_python.core.middleware import RateLimiter

limiter = RateLimiter()

allowed, info = limiter.check(request.ip)
if not allowed:
    return response.json({
        "error": "Too many requests",
        "retry_after": info["reset"]
    }, 429)

# Add rate limit headers to the response
limiter.apply_headers(response, info)
```

Like CORS, the rate limiter is active by default based on your `.env` configuration.

---

3. Writing Custom Middleware

Tina4 Python supports two styles of middleware: function-based and class-based.

Function-Based Middleware

A middleware function takes three arguments: `request`, `response`, and `next_handler`. It must return a response.

```
async def my_middleware(request, response, next_handler):
```

```
    # Code that runs BEFORE the route handler
    print("Before handler")

    # Call the next middleware or the route handler
    result = await next_handler(request, response)

    # Code that runs AFTER the route handler
    print("After handler")

    return result
```

Class-Based Middleware

Class-based middleware uses a naming convention: static methods prefixed with `before_` run before the route handler, and methods prefixed with `after_` run after it. Each method receives `(request, response)` and returns `(request, response)`.

```
class MyMiddleware:
    @staticmethod
    def before_check(request, response):
        """Runs before the route handler."""
        print("Before handler")
        return request, response

    @staticmethod
    def after_cleanup(request, response):
        """Runs after the route handler."""
        print("After handler")
        return request, response
```

If a `before_*` method returns a response with an error status (>= 400), the route handler is skipped entirely. This is short-circuiting.

Example: Request Timer (Class-Based)

```
import time

class TimingMiddleware:
    @staticmethod
    def before_start_timer(request, response):
        request._start_time = time.time()
        return request, response

    @staticmethod
    def after_add_timing(request, response):
        elapsed = time.time() - getattr(request, "_start_time", time.time())
        from tina4_python.core.response import Response
        Response.add_header("X-Response-Time", f"{elapsed:.3f}s")
        return request, response
```

Example: Request Logger (Function-Based)

```
from datetime import datetime

async def log_middleware(request, response, next_handler):
    timestamp = datetime.now().strftime("%Y-%m-%d %H:%M:%S")
    print(f"[{timestamp}] {request.method} {request.path}")
    print(f"  Headers: {dict(request.headers)}")

    if request.body:
        print(f"  Body: {request.body}")

    result = await next_handler(request, response)

    print(f"  Status: {result.status_code if hasattr(result, 'status_code') else 'unknown'}")

    return result
```

Example: Security Headers (Class-Based)

```
from tina4_python.core.response import Response

class SecurityHeaders:
    @staticmethod
    def after_security(request, response):
        Response.add_header("X-Content-Type-Options", "nosniff")
        Response.add_header("X-Frame-Options", "DENY")
        Response.add_header("X-XSS-Protection", "1; mode=block")
        Response.add_header("Strict-Transport-Security", "max-age=31536000")
        return request, response
```

Example: JSON Content-Type Enforcer (Function-Based)

```
async def require_json(request, response, next_handler):
    if request.method in ("POST", "PUT", "PATCH"):
        content_type = request.headers.get("Content-Type", "")
        if "application/json" not in content_type:
            return response.json({
                "error": "Content-Type must be application/json"
            }, 415)

    return await next_handler(request, response)
```

Example: Request ID (Class-Based)

```
import secrets

class RequestIdMiddleware:
    @staticmethod
    def before_inject_id(request, response):
        request.request_id = secrets.token_hex(8)
        return request, response

    @staticmethod
    def after_add_header(request, response):
        from tina4_python.core.response import Response
        Response.add_header("X-Request-ID", getattr(request, "request_id", ""))
        return request, response
```

Example: Input Sanitization (Class-Based)

```
import html

class InputSanitizer:
    @staticmethod
    def before_sanitize(request, response):
        if request.body and isinstance(request.body, dict):
            request.body = InputSanitizer._sanitize_dict(request.body)
        return request, response

    @staticmethod
    def _sanitize_dict(data):
        sanitized = {}
        for key, value in data.items():
            if isinstance(value, str):
                sanitized[key] = html.escape(value)
            elif isinstance(value, dict):
                sanitized[key] = InputSanitizer._sanitize_dict(value)
            else:
                sanitized[key] = value
        return sanitized
```

---

4. The @middleware Decorator

Apply middleware to a single route with the `@middleware` decorator. Both function-based and class-based middleware work:

```
from tina4_python.core.router import get, post, middleware

# Function-based middleware
@middleware(timer_middleware)
@get("/api/data")
async def get_data(request, response):
    return response.json({"data": [1, 2, 3]})

# Class-based middleware
@middleware(TimingMiddleware)
@get("/api/stats")
async def get_stats(request, response):
    return response.json({"stats": {}})
```

Apply multiple middleware by passing them as separate arguments:

```
@middleware(RequestIdMiddleware, SecurityHeaders, require_json)
@post("/api/items")
async def create_item(request, response):
    return response.json({"item": request.body}, 201)
```

You can mix function-based and class-based middleware freely. They run in the order you list them. In the example above: `RequestIdMiddleware` runs first (before and after hooks), then `SecurityHeaders`, then `require_json`, then the route handler.

---

5. Middleware on Route Groups

Apply middleware to all routes in a group:

```
from tina4_python.core.router import Router, get, post, middleware

def api_v1(group):

    @group.get("/users")
    async def list_users(request, response):
        return response.json({"users": []})

    @group.post("/users")
    async def create_user(request, response):
        return response.json({"created": True}, 201)

    @group.get("/products")
    async def list_products(request, response):
        return response.json({"products": []})

Router.group("/api/v1", api_v1, middleware=[TimingMiddleware, SecurityHeaders])
```

Every route inside the group now has `TimingMiddleware` and `SecurityHeaders` applied. You can still add route-specific middleware on top:

```
def api_v1(group):

    @group.get("/public")
    async def public_endpoint(request, response):
        # Only group middleware runs
        return response.json({"public": True})

    @middleware(AuthMiddleware)
    @group.post("/admin")
    async def admin_endpoint(request, response):
        # Group middleware + AuthMiddleware both run
        return response.json({"admin": True})

Router.group("/api/v1", api_v1, middleware=[RequestIdMiddleware])
```

Group middleware always runs before route-specific middleware. This means authentication checks at the group level cannot be bypassed by individual routes.

---

6. Execution Order

Stacked middleware forms a nested pipeline. Requests travel inward. Responses travel outward:

```
Request arrives
  → log_middleware (before)
    → auth_middleware (before)
      → timer_middleware (before)
        → route handler
      → timer_middleware (after)
    → auth_middleware (after)
  → log_middleware (after)
Response sent
```

Each middleware wraps around the next one. The outermost middleware runs first on the way in and last on the way out.

Here is a concrete example showing the order:

```
async def middleware_a(request, response, next_handler):
    print("A: before")
    result = await next_handler(request, response)
    print("A: after")
    return result

async def middleware_b(request, response, next_handler):
    print("B: before")
    result = await next_handler(request, response)
    print("B: after")
    return result

@get("/test")
@middleware(middleware_a, middleware_b)
async def test(request, response):
    print("Handler")
    return response.json({"ok": True})
```

When you request `/test`, the console shows:

```
A: before
B: before
Handler
B: after
A: after
```

---

7. Short-Circuiting

Skip `next_handler` and the chain stops cold. The route handler never runs. This is how blocking middleware works:

```
async def maintenance_mode(request, response, next_handler):
    import os
    if os.getenv("MAINTENANCE_MODE") == "true":
        return response.json({
            "error": "Service is under maintenance. Please try again later."
        }, 503)

    return await next_handler(request, response)
```

When `MAINTENANCE_MODE=true`, every request gets a 503 response without reaching any route handler.

Conditional Short-Circuit

```
async def require_api_key(request, response, next_handler):
    api_key = request.headers.get("X-API-Key", "")

    if not api_key:
        return response.json({"error": "API key required"}, 401)

    # Validate the key against a database or config
    valid_keys = ["key-abc-123", "key-def-456", "key-ghi-789"]
    if api_key not in valid_keys:
        return response.json({"error": "Invalid API key"}, 403)

    # Attach the key info to the request for the handler to use
    request.api_key = api_key

    return await next_handler(request, response)
```

```
# No key -- 401
curl http://localhost:7146/api/data
```

```
{"error":"API key required"}
```

```
# Invalid key -- 403
curl http://localhost:7146/api/data -H "X-API-Key: wrong-key"
```

```
{"error":"Invalid API key"}
```

```
# Valid key -- 200
curl http://localhost:7146/api/data -H "X-API-Key: key-abc-123"
```

```
{"data":[1,2,3]}
```

---

8. Modifying Request and Response

Middleware can modify the request before it reaches the handler, and the response before it reaches the client.

Adding Data to the Request

```
async def inject_user_agent(request, response, next_handler):
    ua = request.headers.get("User-Agent", "")

    request.is_mobile = "Mobile" in ua or "Android" in ua or "iPhone" in ua
    request.is_bot = "bot" in ua.lower() or "spider" in ua.lower()

    return await next_handler(request, response)
```

Now the route handler can access `request.is_mobile` and `request.is_bot`.

Modifying the Response

```
async def add_security_headers(request, response, next_handler):
    result = await next_handler(request, response)

    # Add security headers to every response
    return response.header("X-Content-Type-Options", "nosniff") \
                   .header("X-Frame-Options", "DENY") \
                   .header("X-XSS-Protection", "1; mode=block") \
                   .header("Strict-Transport-Security", "max-age=31536000; includeSubDomains")
```

---

9. Real-World Example: JWT Authentication Middleware

This class-based middleware verifies JWT tokens on protected routes. It uses the `before_*` / `after_*` convention.

```
from tina4_python.auth import Auth

class JwtAuthMiddleware:
    @staticmethod
    def before_verify_token(request, response):
        auth_header = request.headers.get("authorization", "")

        if not auth_header or not auth_header.startswith("Bearer "):
            return request, response({"error": "Authorization header required"}, 401)

        token = auth_header[7:]
        payload = Auth.valid_token(token)

        if payload is None:
            return request, response({"error": "Invalid or expired token"}, 401)

        # Attach the decoded payload to the request
        request.user = payload
        return request, response
```

Apply it to a group of protected routes:

```
from tina4_python.core.router import Router, get, post, middleware

def protected_routes():

    @get("/profile")
    async def get_profile(request, response):
        return response({"user": request.user})

    @post("/settings")
    async def update_settings(request, response):
        user_id = request.user["sub"]
        return response({"updated": True, "user_id": user_id})

Router.group("/api/protected", protected_routes, middleware=[JwtAuthMiddleware])
```

The middleware short-circuits with 401 if the token is missing or invalid. The route handler never runs. If the token is valid, the decoded payload is available as `request.user`.

---

10. Real-World Example: API Key Middleware with Database Lookup

```
from tina4_python.database.connection import Database
from datetime import datetime

async def api_key_middleware(request, response, next_handler):
    api_key = request.headers.get("X-API-Key", "")

    if not api_key:
        return response.json({
            "error": "API key required. Send it in the X-API-Key header."
        }, 401)

    db = Database()
    key_record = db.fetch_one(
        "SELECT id, name, rate_limit, is_active FROM api_keys WHERE key_value = ?",
        [api_key]
    )

    if key_record is None:
        return response.json({"error": "Invalid API key"}, 403)

    if not key_record["is_active"]:
        return response.json({"error": "API key has been deactivated"}, 403)

    # Update last used timestamp
    db.execute(
        "UPDATE api_keys SET last_used_at = ?, request_count = request_count + 1 WHERE id = ?",
        [datetime.now().isoformat(), key_record["id"]]
    )

    # Attach key info to request
    request.api_key_id = key_record["id"]
    request.api_key_name = key_record["name"]

    return await next_handler(request, response)
```

---

11. Exercise: Build an API Key Middleware System

Build a complete API key system with key management and usage tracking.

Requirements

- Create a migration for an `api_keys` table: `id`, `name`, `key_value` (unique), `is_active` (default true), `rate_limit` (default 100), `request_count` (default 0), `last_used_at`, `created_at`
- Build these endpoints:

Method	Path	Middleware	Description

`POST`	`/admin/api-keys`	Auth	Create a new API key (generate random key)
`GET`	`/admin/api-keys`	Auth	List all API keys with usage stats
`DELETE`	`/admin/api-keys/{id:int}`	Auth	Deactivate an API key
`GET`	`/api/data`	API Key	Protected endpoint -- requires valid API key
`GET`	`/api/status`	API Key	Another protected endpoint

- The API key middleware should:
- Check `X-API-Key` header
- Validate against the database
- Reject deactivated keys
- Track usage (increment count, update last*used*at)
- Attach key info to the request

Test with:

```
# Create an API key (requires auth token from Chapter 8)
curl -X POST http://localhost:7146/admin/api-keys \
  -H "Authorization: Bearer YOUR_AUTH_TOKEN" \
  -H "Content-Type: application/json" \
  -d '{"name": "Mobile App"}'

# Use the API key
curl http://localhost:7146/api/data \
  -H "X-API-Key: THE_GENERATED_KEY"

# List keys with stats
curl http://localhost:7146/admin/api-keys \
  -H "Authorization: Bearer YOUR_AUTH_TOKEN"
```

---

12. Solution

Migration

Create `migrations/20260322170000_create_api_keys_table.sql`:

```
-- UP
CREATE TABLE api_keys (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    name TEXT NOT NULL,
```

```
    key_value TEXT NOT NULL UNIQUE,
    is_active INTEGER NOT NULL DEFAULT 1,
    rate_limit INTEGER NOT NULL DEFAULT 100,
    request_count INTEGER NOT NULL DEFAULT 0,
    last_used_at TEXT,
    created_at TEXT DEFAULT CURRENT_TIMESTAMP
);

CREATE UNIQUE INDEX idx_api_keys_key ON api_keys (key_value);

-- DOWN
DROP INDEX IF EXISTS idx_api_keys_key;
DROP TABLE IF EXISTS api_keys;
```

Routes

Create `src/routes/api_keys.py`:

```
from tina4_python.core.router import get, post, delete as delete_route, middleware
from tina4_python.auth import Auth
from tina4_python.database.connection import Database
from datetime import datetime
import secrets


async def auth_middleware(request, response, next_handler):
    auth_header = request.headers.get("Authorization", "")
    if not auth_header or not auth_header.startswith("Bearer "):
        return response.json({"error": "Authorization required"}, 401)

    token = auth_header[7:]
    if not Auth.valid_token(token):
        return response.json({"error": "Invalid or expired token"}, 401)

    request.user = Auth.get_payload(token)
    return await next_handler(request, response)


async def api_key_middleware(request, response, next_handler):
    api_key = request.headers.get("X-API-Key", "")

    if not api_key:
        return response.json({"error": "API key required. Send in X-API-Key header."}, 401)

    db = Database()
    key_record = db.fetch_one(
        "SELECT id, name, is_active FROM api_keys WHERE key_value = ?",
        [api_key]
    )

    if key_record is None:
        return response.json({"error": "Invalid API key"}, 403)

    if not key_record["is_active"]:
        return response.json({"error": "API key has been deactivated"}, 403)

    db.execute(
        "UPDATE api_keys SET last_used_at = ?, request_count = request_count + 1 WHERE id = ?",
        [datetime.now().isoformat(), key_record["id"]]
```

```
    )

    request.api_key_id = key_record["id"]
    request.api_key_name = key_record["name"]

    return await next_handler(request, response)


@post("/admin/api-keys")
@middleware(auth_middleware)
async def create_api_key(request, response):
    db = Database()
    name = request.body.get("name", "Unnamed Key")
    key_value = f"tk_{secrets.token_hex(24)}"

    db.execute(
        "INSERT INTO api_keys (name, key_value) VALUES (?, ?)",
        [name, key_value]
    )

    key = db.fetch_one("SELECT * FROM api_keys WHERE id = last_insert_rowid()")

    return response.json({"message": "API key created", "key": key}, 201)


@get("/admin/api-keys")
@middleware(auth_middleware)
async def list_api_keys(request, response):
    db = Database()
    keys = db.fetch("SELECT id, name, key_value, is_active, rate_limit, request_count, last_used
    return response.json({"keys": keys, "count": len(keys)})


@delete_route("/admin/api-keys/{id:int}")
@middleware(auth_middleware)
async def deactivate_api_key(request, response):
    db = Database()
    key_id = request.params["id"]

    existing = db.fetch_one("SELECT id FROM api_keys WHERE id = ?", [key_id])
    if existing is None:
        return response.json({"error": "API key not found"}, 404)

    db.execute("UPDATE api_keys SET is_active = 0 WHERE id = ?", [key_id])
    return response.json({"message": "API key deactivated"})


@get("/api/data")
@middleware(api_key_middleware)
async def api_data(request, response):
    return response.json({
        "data": [1, 2, 3, 4, 5],
        "api_key": request.api_key_name
    })


@get("/api/status")
@middleware(api_key_middleware)
async def api_status(request, response):
```

```
    return response.json({
        "status": "operational",
        "api_key": request.api_key_name
    })
```

---

13. Gotchas

1. Forgetting to await next_handler

Problem: The route handler never runs, or you get an error about a coroutine object.

Cause: You called `next_handler(request, response)` without `await`.

Fix: Always use `await next_handler(request, response)`. Since Tina4 Python is async, every middleware and handler must be awaited.

2. Middleware modifies response after it is sent

Problem: Headers or cookies you add in the "after" phase of middleware do not appear in the response.

Cause: The response was already finalized by the route handler.

Fix: In the "after" phase, modify the result returned by `next_handler`, not the original `response` object. Some modifications may need to happen in the "before" phase instead.

3. Middleware applied to wrong routes

Problem: Your API key middleware runs on public routes that should not require a key.

Cause: The middleware is applied to the group, and the public route is inside that group.

Fix: Move public routes outside the group, or use `@noauth` on specific routes to bypass authentication middleware. For custom middleware like API key checks, you need to handle exemptions manually by checking the path in the middleware.

4. Middleware execution order surprises

Problem: Your auth check runs after your logging middleware, but you wanted it to run first.

Cause: Middleware in `@middleware(a, b, c)` runs in left-to-right order: `a` wraps `b` wraps `c` wraps handler.

Fix: Put the middleware you want to run first at the leftmost position: `@middleware(auth_middleware, log_middleware)`.

5. Error in middleware breaks the chain

Problem: An unhandled exception in middleware causes a 500 error without reaching the route handler.

Cause: If middleware throws an exception before calling `next_handler`, no subsequent middleware or the handler runs.

Fix: Wrap risky middleware code in try/except and return an appropriate error response instead of letting the exception propagate.

6. Database connections in middleware

Problem: Opening a database connection in middleware that runs on every request causes connection pool exhaustion.

Cause: Each middleware call creates a new `Database()` instance.

Fix: Tina4's `Database()` uses connection pooling internally, so this is usually safe. But if you are seeing issues, cache your database lookups (like API keys) in memory with a TTL instead of querying on every request.

7. Modifying request in middleware does not persist

Problem: You set `request.custom_field = "value"` in middleware, but the route handler does not see it.

Cause: In some edge cases, the request object may be copied between middleware stages.

Fix: Use `request.custom_field` consistently. If it is not persisting, check that you are modifying the same request object that is passed to `next_handler`. Do not create a new request object.

Security

Every route you write is a door. Chapter 8 gave you locks. Chapter 10 gave you guards. Chapter 9 gave you session keys. This chapter ties them together into a defence that works without thinking about it.

Tina4 ships secure by default. POST routes require authentication. CSRF tokens protect forms. Security headers harden every response. The framework does the boring security work so you focus on building features. But you need to understand what it does — and why — so you don't accidentally undo it.

---

1. Secure-by-Default Routing

Every POST, PUT, PATCH, and DELETE route requires a valid `Authorization: Bearer` token. No configuration needed. No decorator to remember. The framework enforces this before your handler runs.

```
from tina4_python.core.router import post

@post("/api/orders")
async def create_order(request, response):
    # This handler ONLY runs if the request carries a valid Bearer token.
    # Without one, the framework returns 401 before your code executes.
    return response({"created": True}, 201)
```

Test it without a token:

```
curl -X POST http://localhost:7146/api/orders \
  -H "Content-Type: application/json" \
  -d '{"product": "widget"}'
# 401 Unauthorized
```

Test it with a valid token:

```
curl -X POST http://localhost:7146/api/orders \
  -H "Content-Type: application/json" \
  -H "Authorization: Bearer eyJhbGciOiJIUzI1NiJ9..." \
  -d '{"product": "widget"}'
# 201 Created
```

GET routes are public by default. Anyone can read. Writing requires proof of identity.

Making a Write Route Public

Some endpoints need to accept unauthenticated writes — webhooks, registration forms, public contact forms. Use `@noauth()`:

```
from tina4_python.core.router import post, noauth

@noauth()
@post("/api/webhooks/stripe")
async def stripe_webhook(request, response):
    # No token required. Stripe can POST here freely.
    return response({"received": True})
```

Protecting a GET Route

Admin dashboards, user profiles, account settings — some pages need protection even though they only read data. Use `@secured()`:

```
from tina4_python.core.router import get, secured

@secured()
@get("/api/admin/users")
async def admin_users(request, response):
    # Requires a valid Bearer token, even though it's a GET.
    return response({"users": []})
```

The Rule

Method	Default	Override
GET, HEAD, OPTIONS	Public	`@secured()` to protect
POST, PUT, PATCH, DELETE	Auth required	`@noauth()` to open

Two decorators. One rule. No surprises.

---

2. CSRF Protection

Cross-Site Request Forgery tricks a user's browser into submitting a form to your server. The browser sends cookies automatically — including session cookies. Without CSRF protection, an attacker's page can submit forms as your logged-in user.

Tina4 blocks this with form tokens.

How It Works

- Your template renders a hidden token using `{{ form_token() }}`.
- The browser submits the token with the form data.
- The `CsrfMiddleware` validates the token before the route handler runs.
- Invalid or missing tokens receive a `403 Forbidden` response.

The Template

```
<form method="POST" action="/api/profile">
    {{ form_token() }}
    <input type="text" name="name" placeholder="Your name">
    <button type="submit">Save</button>
</form>
```

The `{{ form_token() }}` call generates a hidden input field containing a signed JWT. The token is bound to the current session — a token from one session cannot be used in another.

The Middleware

CSRF protection is on by default. Every POST, PUT, PATCH, and DELETE request must include a valid form token. The middleware checks two places:

- **Request body** — `request.body["formToken"]`
- **Request header** — `X-Form-Token`

If the token is missing or invalid, the middleware returns 403 before your handler runs.

AJAX Requests

For JavaScript-driven forms, send the token as a header:

```
// frond.min.js handles this automatically via saveForm()
// For manual AJAX, extract the token from the hidden field:
const token = document.querySelector('input[name="formToken"]').value;

fetch("/api/profile", {
    method: "POST",
    headers: {
        "Content-Type": "application/json",
        "X-Form-Token": token
    },
    body: JSON.stringify({ name: "Alice" })
});
```

Tokens in Query Strings — Blocked

Tokens must never appear in URLs. Query strings are logged in server access logs, browser history, and referrer headers. A token in the URL is a token anyone can steal.

Tina4 rejects any request that carries `formToken` in the query string and logs a warning:

```
CSRF token found in query string — rejected for security.
Use POST body or X-Form-Token header instead.
```

Skipping CSRF Validation

Three scenarios skip CSRF checks automatically:

- **GET, HEAD, OPTIONS** — Safe methods don't modify state.
- **Routes with** `@noauth()` — Public write endpoints don't need CSRF (they have no session to protect).
- **Requests with a valid Bearer token** — API clients authenticate with tokens, not cookies. CSRF only matters for cookie-based sessions.

Disabling CSRF Globally

For internal microservices behind a firewall — where no browser ever touches the API — you can disable CSRF entirely:

```
TINA4_CSRF=false
```

Leave it enabled for anything a browser can reach. The cost is one hidden field per form. The protection is worth it.

---

3. Session-Bound Tokens

A form token alone prevents cross-site forgery. But what if someone steals a token from a form? Session binding stops them.

When `{{ form_token() }}` generates a token, it embeds the current session ID in the JWT payload. The CSRF middleware checks that the session ID in the token matches the session ID of the request. A token stolen from one session cannot be replayed in another.

This happens automatically. No configuration. No extra code.

How Stolen Tokens Fail

- Attacker visits your site, gets a form token for session `abc-123`.
- Attacker sends that token from their own session `xyz-789`.
- The middleware compares: `abc-123 != xyz-789` — rejected with 403.

The token is cryptographically valid. But it belongs to the wrong session. The binding catches it.

---

4. Security Headers

Every response from Tina4 carries security headers. The `SecurityHeadersMiddleware` injects them before the response reaches the browser. No opt-in required.

Header	Default Value	Purpose
`X-Frame-Options`	`SAMEORIGIN`	Prevents clickjacking — your pages cannot be embedded in iframes on other domains.
`X-Content-Type-Options`	`nosniff`	Stops browsers from guessing content types. A script is a script, not HTML.
`Content-Security-Policy`	`default-src 'self'`	Controls which resources the browser loads. Blocks inline scripts from injected HTML.
`Referrer-Policy`	`strict-origin-when-cross-origin`	Limits referrer data sent to external sites. Protects internal URLs from leaking.
`X-XSS-Protection`	`0`	Disabled. Modern CSP replaces this legacy header. Keeping it on can introduce vulnerabilities.
`Permissions-Policy`	`camera=(), microphone=(), geolocation=()`	Disables browser APIs your app does not need.

HSTS — Enforcing HTTPS

Strict Transport Security tells the browser to always use HTTPS. Disabled by default (it breaks local development on HTTP). Enable it in production:

```
TINA4_HSTS=31536000
```

This sets a one-year HSTS policy with `includeSubDomains`. Once a browser sees this header, it refuses to connect over HTTP — even if the user types `http://`.

Customising Headers

Override any header via environment variables:

```
TINA4_FRAME_OPTIONS=DENY
TINA4_CSP=default-src 'self'; script-src 'self' https://cdn.example.com
TINA4_REFERRER_POLICY=no-referrer
TINA4_PERMISSIONS_POLICY=camera=(), microphone=(), geolocation=(), payment=()
```

---

5. SameSite Cookies

Session cookies control who can send them. The `SameSite` attribute tells the browser when to include the cookie in requests.

Value	Behaviour
`Strict`	Cookie sent only on same-site requests. Even clicking a link from email to your site won't include the cookie. The user must navigate directly.
`Lax`	Cookie sent on same-site requests and top-level navigations (clicking a link). Not sent on cross-site AJAX or form POSTs from other domains.
`None`	Cookie sent on all requests, including cross-site. Requires `Secure` flag (HTTPS only).

Tina4 defaults to `Lax`. This blocks cross-site form submissions (CSRF) while allowing normal link navigation. Users click a link to your site from an email — they stay logged in. An attacker's page submits a hidden form — the cookie is withheld.

For most applications, `Lax` is the right choice. Change it only if you understand the trade-offs.

---

6. Login Flow — Complete Example

Authentication, sessions, tokens, and security converge in the login flow. Here is a complete implementation.

The Login Route

```
from tina4_python.core.router import post, noauth
from tina4_python.auth import Auth, get_token

@noauth()
@post("/api/login")
async def login(request, response):
    email = request.body.get("email", "")
    password = request.body.get("password", "")

    if not email or not password:
        return response({"error": "Email and password required"}, 400)

    # Look up user (replace with your database query)
    user = db.fetch_one(
        "SELECT id, email, password_hash, role FROM users WHERE email = ?",
        [email]
    )

    if not user:
        return response({"error": "Invalid credentials"}, 401)

    # Verify password
    if not Auth.check_password(password, user["password_hash"]):
        return response({"error": "Invalid credentials"}, 401)

    # Generate token with user claims
    token = get_token(
        {"sub": user["id"], "email": user["email"], "role": user["role"]}
    )

    # Store user in session
    request.session.set("user_id", user["id"])
    request.session.set("email", user["email"])
    request.session.set("role", user["role"])
    request.session.save()

    return response({"token": token, "user": {"id": user["id"], "email": user["email"]}})
```

The `@noauth()` decorator opens this route to unauthenticated requests. The handler validates credentials and issues a token. The session stores the user identity for server-side lookups.

The Login Form

```
{% extends "base.twig" %}
{% block content %}
<div class="container mt-5" style="max-width: 400px;">
    <h2>Login</h2>
    <form id="loginForm">
        {{ form_token() }}
        <div class="mb-3">
            <label for="email">Email</label>
            <input type="email" name="email" id="email" class="form-control"
                   placeholder="you@example.com" required>
        </div>
        <div class="mb-3">
            <label for="password">Password</label>
            <input type="password" name="password" id="password" class="form-control"
                   placeholder="Your password" required>
        </div>
        <button type="button" class="btn btn-primary w-100"
                onclick="saveForm('loginForm', '/api/login', 'loginMsg', handleLogin)">
            Sign In
        </button>
        <div id="loginMsg" class="mt-3"></div>
    </form>
</div>

<script>
function handleLogin(result) {
    if (result.token) {
        localStorage.setItem("token", result.token);
        window.location.href = "/dashboard";
    }
}
</script>
{% endblock %}
```

Protected Pages — Checking the Session

```
from tina4_python.core.router import get

@get("/dashboard")
async def dashboard(request, response):
    user_id = request.session.get("user_id")

    if not user_id:
        return response.redirect("/login")

    return response.render("dashboard.twig", {
        "email": request.session.get("email"),
        "role": request.session.get("role"),
    })
```

Logout — Destroying the Session

```
from tina4_python.core.router import post, noauth

@noauth()
@post("/api/logout")
async def logout(request, response):
    request.session.destroy()
    return response({"logged_out": True})
```

---

7. Handling Expired Sessions

Sessions expire. Tokens expire. The user clicks a link and finds themselves staring at a broken page or a cryptic error. A good security implementation handles expiry gracefully.

The Pattern: Redirect to Login, Then Back

When a session expires mid-use, the user should:

- See a login page — not an error.
- Log in again.
- Land on the page they were trying to reach — not the home page.

```
from urllib.parse import quote

@get("/account/settings")
async def account_settings(request, response):
    user_id = request.session.get("user_id")

    if not user_id:
        # Remember where they wanted to go
        return_url = quote(request.url)
        return response.redirect(f"/login?redirect={return_url}")

    return response.render("settings.twig", {"user_id": user_id})
```

The login handler reads the `redirect` parameter after successful authentication:

```
@noauth()
@post("/api/login")
async def login(request, response):
    # ... validate credentials ...

    redirect_url = request.params.get("redirect", "/dashboard")

    return response({
        "token": token,
        "redirect": redirect_url
    })
```

The login form JavaScript redirects to the saved URL:

```
function handleLogin(result) {
    if (result.token) {
        localStorage.setItem("token", result.token);
        window.location.href = result.redirect || "/dashboard";
```

```
    }
}
```

Token Refresh

Tokens expire based on `TINA4_TOKEN_LIMIT` (default: 60 minutes). The `frond.min.js` frontend library handles token refresh automatically — every response includes a `FreshToken` header with a new token. The client stores it and uses it for the next request.

For custom AJAX code, read the header yourself:

```
const res = await fetch("/api/data", {
    headers: { "Authorization": "Bearer " + localStorage.getItem("token") }
});

const freshToken = res.headers.get("FreshToken");
if (freshToken) {
    localStorage.setItem("token", freshToken);
}
```

---

8. Rate Limiting

Brute-force login attempts. Credential stuffing. API abuse. Rate limiting stops all of them.

Tina4 includes a sliding-window rate limiter that tracks requests per IP address. It activates automatically.

```
TINA4_RATE_LIMIT=100
TINA4_RATE_WINDOW=60
```

One hundred requests per sixty seconds per IP. Exceed the limit and the server returns `429 Too Many Requests` with headers telling the client when to retry:

```
X-RateLimit-Limit: 100
X-RateLimit-Remaining: 0
X-RateLimit-Reset: 45
```

For login routes, consider a stricter limit:

```
from tina4_python.core.router import post, noauth, middleware

class LoginRateLimit:
    @staticmethod
    def before_rate_check(request, response):
        # Custom rate limiter: 5 attempts per 60 seconds for login
        ip = request.ip
        # ... implement per-route rate limiting ...
        return request, response

@noauth()
@middleware(LoginRateLimit)
@post("/api/login")
async def login(request, response):
    # ... login logic ...
    pass
```

---

9. CORS and Credentials

When your frontend runs on a different origin than your API (common in development), CORS controls whether the browser sends cookies and auth headers.

Tina4 handles CORS automatically. The relevant security settings:

```
TINA4_CORS_ORIGINS=*
TINA4_CORS_CREDENTIALS=true
```

Two rules to remember:

- `TINA4_CORS_ORIGINS=*` **with** `TINA4_CORS_CREDENTIALS=true` is invalid per the CORS spec. Tina4 handles this — when origin is `*`, the credentials header is not sent. But in production, list your actual origins.
- **Cookies need** `SameSite=None; Secure` for true cross-origin requests. If your API is on `api.example.com` and your frontend is on `app.example.com`, the default `Lax` cookie works because they share the same registrable domain. Different domains need `SameSite=None`.

Production CORS:

```
TINA4_CORS_ORIGINS=https://app.example.com,https://admin.example.com
TINA4_CORS_CREDENTIALS=true
```

---

10. Security Checklist

Before you deploy, verify:

- [ ] `TINA4_SECRET` is set to a long, random string — not the default.
- [ ] `TINA4_DEBUG=false` in production.
- [ ] `TINA4_HSTS=31536000` if serving over HTTPS.
- [ ] `TINA4_CORS_ORIGINS` lists your actual domains — not `*`.
- [ ] `TINA4_CSRF=true` (the default) for any browser-facing application.
- [ ] Login route uses `@noauth()` and validates credentials before issuing tokens.
- [ ] Session is regenerated after login (prevents session fixation).
- [ ] Passwords are hashed with `Auth.hash_password()` — never stored in plain text.
- [ ] File uploads are validated and size-limited (`TINA4_MAX_UPLOAD_SIZE`).
- [ ] Rate limiting is active on login and registration routes.
- [ ] Expired sessions redirect to login with a return URL.

Gotchas

1. "My POST route returns 401 but I didn't add auth"

Cause: Tina4 requires authentication on all write routes by default.

Fix: Add `@noauth()` above the route decorator if the endpoint should be public. Otherwise, send a valid Bearer token with the request.

2. "CSRF validation fails on AJAX requests"

Cause: The form token is not included in the request.

Fix: Send the token as an `X-Form-Token` header. If using `frond.min.js`, call `saveForm()` — it handles tokens automatically.

3. "I disabled CSRF but forms still fail"

Cause: The route still requires Bearer auth (separate from CSRF). CSRF and auth are independent checks.

Fix: Either send a Bearer token or add `@noauth()` to the route.

4. "My Content-Security-Policy blocks inline scripts"

Cause: The default CSP is `default-src 'self'`, which blocks inline `<script>` tags and `onclick` handlers.

Fix: Move scripts to external `.js` files (the right approach) or relax the CSP:

```
TINA4_CSP=default-src 'self'; script-src 'self' 'unsafe-inline'
```

Prefer external scripts. Inline scripts are an XSS vector.

5. "User stays logged in after session expires"

Cause: The frontend stores a JWT in localStorage. The token is still valid even after the session is destroyed server-side.

Fix: Check the session on every page load. If the session is gone, redirect to login regardless of the token. Tokens authenticate API calls; sessions track server-side state. Both must be valid.

Exercise: Secure Contact Form

Build a public contact form that:

- Does not require login (`@noauth()`).
- Validates CSRF tokens (form includes `{{ form_token() }}`).
- Rate-limits submissions to 3 per minute per IP.
- Stores messages in the database.
- Returns a success message.

Solution

```
# src/routes/contact.py
from tina4_python.core.router import post, get, noauth, template

@get("/contact")
@template("contact.twig")
async def contact_page(request, response):
    return {"title": "Contact Us"}


@noauth()
@post("/api/contact")
async def submit_contact(request, response):
    name = request.body.get("name", "").strip()
    email = request.body.get("email", "").strip()
    message = request.body.get("message", "").strip()

    if not name or not email or not message:
        return response({"error": "All fields are required"}, 400)

    db.insert("contact_messages", {
        "name": name,
        "email": email,
        "message": message,
    })
    db.commit()

    return response({"success": True, "message": "Thank you for your message"})
```

```
{# src/templates/contact.twig #}
{% extends "base.twig" %}
{% block title %}Contact Us{% endblock %}
{% block content %}
<div class="container mt-5" style="max-width: 500px;">
    <h2>{{ title }}</h2>
    <form id="contactForm">
        {{ form_token() }}
        <div class="mb-3">
            <label for="name">Name</label>
            <input type="text" name="name" id="name" class="form-control"
                   placeholder="Jane Smith" required>
        </div>
        <div class="mb-3">
            <label for="email">Email</label>
            <input type="email" name="email" id="email" class="form-control"
                   placeholder="jane@example.com" required>
        </div>
        <div class="mb-3">
            <label for="message">Message</label>
            <textarea name="message" id="message" class="form-control" rows="4"
                      placeholder="How can we help?" required></textarea>
        </div>
        <button type="button" class="btn btn-primary"
                onclick="saveForm('contactForm', '/api/contact', 'contactMsg')">
            Send Message
        </button>
        <div id="contactMsg" class="mt-3"></div>
    </form>
</div>
```

```
{% endblock %}
```

The form is public. The CSRF token is present. The `@noauth()` decorator opens the route. The middleware validates the token. The database stores the message. The user sees confirmation.

Five moving parts. Zero security holes. The framework handles the rest.

Caching

1. From 800ms to 3ms

Your product catalog page runs 12 database queries. 800 milliseconds to render. Every visitor triggers the same queries, the same template rendering, the same JSON serialization -- for data that changes once a day.

Add caching. The first request takes 800ms. The next 10,000 take 3ms each. A 266x improvement from one line of configuration.

Caching stores the result of expensive operations for reuse. Tina4 provides three levels: response caching (entire HTTP responses), database query caching, and a direct cache API for custom use cases.

---

2. Response Caching with ResponseCache Middleware

The fastest way to cache is at the HTTP response level. The `ResponseCache` middleware stores the complete response (headers and body) and serves it directly on subsequent requests without calling your route handler at all.

```
from tina4_python.core.router import get

@get("/api/products", middleware=["ResponseCache:300"])
async def list_products(request, response):
    # This handler runs 12 database queries and takes 800ms
    # With ResponseCache, it only runs once every 5 minutes

    print("Handler called -- this should only appear once every 5 minutes")

    products = [
        {"id": 1, "name": "Wireless Keyboard", "price": 79.99},
        {"id": 2, "name": "USB-C Hub", "price": 49.99},
        {"id": 3, "name": "Monitor Stand", "price": 129.99}
    ]

    return response({"products": products, "generated_at": datetime.now(timezone.utc).isoformat(
```

The `"ResponseCache:300"` middleware caches the response for 300 seconds (5 minutes). During those 5 minutes:

- The first request runs the handler (800ms)
- The next 10,000 requests serve the cached response (3ms each)
- After 300 seconds, the cache expires and the next request runs the handler again

```
curl http://localhost:7146/api/products
```

```
{
  "products": [
    {"id": 1, "name": "Wireless Keyboard", "price": 79.99},
    {"id": 2, "name": "USB-C Hub", "price": 49.99},
    {"id": 3, "name": "Monitor Stand", "price": 129.99}
```

```
  ],
  "generated_at": "2026-03-22T14:30:00+00:00"
}
```

Call it again within 5 minutes:

```
curl http://localhost:7146/api/products
```

```
{
  "products": [
    {"id": 1, "name": "Wireless Keyboard", "price": 79.99},
    {"id": 2, "name": "USB-C Hub", "price": 49.99},
    {"id": 3, "name": "Monitor Stand", "price": 129.99}
  ],
  "generated_at": "2026-03-22T14:30:00+00:00"
}
```

Notice the `generated_at` timestamp is the same. The handler did not run -- the response came from cache.

Cache Headers

The `ResponseCache` middleware automatically sets cache-related headers:

```
X-Cache: HIT
X-Cache-TTL: 247
Cache-Control: public, max-age=300
```

- `X-Cache: HIT` or `X-Cache: MISS` tells you whether the response came from cache
- `X-Cache-TTL` shows the remaining time-to-live in seconds
- `Cache-Control` enables browser and CDN caching

Caching with Query Parameters

By default, the cache key includes the full URL with query parameters. `/api/products?page=1` and `/api/products?page=2` are cached separately:

```
@get("/api/products", middleware=["ResponseCache:300"])
async def list_products(request, response):
    page = int(request.params.get("page", 1))
    limit = 20
    offset = (page - 1) * limit

    return response({
        "page": page,
        "products": [],
        "generated_at": datetime.now(timezone.utc).isoformat()
    })
```

```
curl "http://localhost:7146/api/products?page=1"  # Cache MISS, stores for page=1
curl "http://localhost:7146/api/products?page=2"  # Cache MISS, stores for page=2
curl "http://localhost:7146/api/products?page=1"  # Cache HIT
```

What Not to Cache

Do not use `ResponseCache` on:

- **POST, PUT, PATCH, DELETE routes**: Only GET responses should be cached

- **User-specific endpoints**: `/api/profile` returns different data for each user
- **Real-time data**: Stock prices, live scores, chat messages
- **Authenticated endpoints**: Unless the cache is scoped per user

```
# GOOD: Public, rarely changing data
@get("/api/categories", middleware=["ResponseCache:3600"])
async def list_categories(request, response):
    return response({"categories": []})

# BAD: User-specific data -- do NOT cache
@get("/api/profile", middleware=["auth_middleware"])
async def get_profile(request, response):
    return response(request.user)
```

---

3. Memory Cache (Default)

Tina4's cache system stores data in memory by default. No configuration needed.

```
# This is the default -- you do not need to set it explicitly
TINA4_CACHE_BACKEND=memory
```

Memory cache is the fastest option (no disk I/O, no network calls) but it resets when the server restarts. It is ideal for development and single-server deployments where losing the cache on restart is acceptable.

---

4. Redis Cache

For production deployments where you want cache persistence across server restarts and shared cache across multiple server instances, use Redis:

```
TINA4_CACHE_BACKEND=redis
TINA4_CACHE_URL=localhost
TINA4_CACHE_URL=6379
TINA4_CACHE_URL=your-redis-password
```

Your code does not change. The `cache_get`, `cache_set`, and `ResponseCache` middleware all work identically. Only the storage backend changes.

Why Redis?

- Cache survives server restarts
- Shared across multiple server instances (behind a load balancer)
- Sub-millisecond reads and writes
- Built-in key expiry (TTL cleanup is automatic)
- Same Redis instance can serve sessions, cache, and queues

---

5. File Cache

If you want cache persistence but do not have Redis, use file-based caching:

```
TINA4_CACHE_BACKEND=file
TINA4_CACHE_DIR=/path/to/cache/directory
```

File cache stores each cache entry as a file on disk. It is slower than memory or Redis but survives server restarts without extra infrastructure.

When to Use File Cache

- You need cache persistence but cannot run Redis
- Your hosting environment is limited (shared hosting, no external services)
- Cache entries are large and you do not want them in memory

---

6. Direct Cache API

For custom caching logic, use the `cache_get`, `cache_set`, and `cache_delete` functions directly:

cache_set

```
from tina4_python.cache import cache_set

# Cache a value for 300 seconds
cache_set("product:42", {
    "id": 42,
    "name": "Wireless Keyboard",
    "price": 79.99,
    "in_stock": True
}, ttl=300)

# Cache a string
cache_set("exchange_rate:USD_EUR", "0.92", ttl=3600)

# Cache indefinitely (no TTL)
cache_set("app:config", {"theme": "dark", "lang": "en"})
```

cache_get

```
from tina4_python.cache import cache_get, cache_set

product = cache_get("product:42")
# Returns the cached value, or None if not found or expired

if product is None:
    # Cache miss -- fetch from database
    product = fetch_product_from_database(42)
    cache_set("product:42", product, ttl=300)

return response(product)
```

cache_delete

```
from tina4_python.cache import cache_delete

# Delete a specific key
cache_delete("product:42")

# Delete multiple keys
cache_delete("product:42")
cache_delete("product:43")
cache_delete("product:44")
```

Real-World Pattern: Cache-Aside

The most common caching pattern is cache-aside (also called lazy loading):

```
from tina4_python.core.router import get
from tina4_python.cache import cache_get, cache_set

@get("/api/products/{product_id}")
async def get_product(request, response):
    product_id = request.params["product_id"]
    cache_key = f"product:{product_id}"

    # 1. Try the cache first
    product = cache_get(cache_key)

    if product is not None:
        # Cache hit -- return immediately
        return response({**product, "source": "cache"})

    # 2. Cache miss -- fetch from database
    db = Database.get_connection()
    product = db.fetch_one(
        "SELECT id, name, category, price, in_stock FROM products WHERE id = ?",
        [product_id]
    )

    if product is None:
        return response({"error": "Product not found"}, 404)

    # 3. Store in cache for next time
    cache_set(cache_key, product, ttl=600)  # Cache for 10 minutes

    return response({**product, "source": "database"})
```

```
curl http://localhost:7146/api/products/42
```

First call (cache miss):

```
{
  "id": 42,
  "name": "Wireless Keyboard",
  "category": "Electronics",
  "price": 79.99,
  "in_stock": true,
  "source": "database"
}
```

Second call (cache hit):

```
{
  "id": 42,
  "name": "Wireless Keyboard",
  "category": "Electronics",
  "price": 79.99,
  "in_stock": true,
  "source": "cache"
}
```

---

7. Database Query Caching

Tina4 can cache database query results automatically. Enable it in `.env`:

```
TINA4_DB_CACHE=true
TINA4_DB_CACHE_TTL=300
```

With database caching enabled, identical queries return cached results instead of hitting the database:

```
from tina4_python.core.router import get

@get("/api/categories")
async def list_categories(request, response):
    db = Database.get_connection()

    # First call: executes the query (20ms)
    # Subsequent calls within 300 seconds: returns cached result (0.1ms)
    categories = db.fetch_all("SELECT * FROM categories ORDER BY name")

    return response({"categories": categories})
```

The cache key is derived from the SQL query and its parameters. Different queries or different parameters produce different cache keys:

```
# These are cached separately:
db.fetch_all("SELECT * FROM products WHERE category = ?", ["Electronics"])
db.fetch_all("SELECT * FROM products WHERE category = ?", ["Fitness"])
```

When to Use DB Cache

- Read-heavy applications where the same queries run repeatedly
- Reference data that changes infrequently (categories, countries, settings)
- Dashboard queries that aggregate large datasets

When Not to Use DB Cache

- Write-heavy applications where data changes constantly
- Queries with real-time requirements (inventory counts, live prices)
- Queries that must always return the latest data

Skipping Cache for Specific Queries

```
# Force a fresh query, bypassing the cache
fresh_data = db.fetch_all("SELECT * FROM products", no_cache=True)
```

---

8. Cache Invalidation Strategies

Cache invalidation is the hard problem. Stale cache serves outdated data. Premature invalidation throws away performance gains. Three strategies handle this.

Strategy 1: Time-Based Expiry (TTL)

The simplest strategy. Set a TTL and let the cache expire naturally:

```
cache_set("products:featured", featured_products, ttl=600)  # Expires in 10 minutes
```

Good for data where near-real-time accuracy is acceptable. A 10-minute delay in updating the featured products list is usually fine.

Strategy 2: Event-Based Invalidation

Clear the cache when the underlying data changes:

```
from tina4_python.core.router import put
from tina4_python.cache import cache_set, cache_delete

@put("/api/products/{product_id}")
async def update_product(request, response):
    product_id = request.params["product_id"]
    body = request.body
    db = Database.get_connection()

    db.execute(
        "UPDATE products SET name = ?, price = ? WHERE id = ?",
        [body["name"], body["price"], product_id]
    )

    # Invalidate the cache for this product
    cache_delete(f"product:{product_id}")

    # Also invalidate any list caches that might include this product
    cache_delete("products:all")
    cache_delete("products:featured")

    updated = db.fetch_one("SELECT * FROM products WHERE id = ?", [product_id])

    return response(updated)
```

This is the most accurate strategy -- the cache is always fresh after a write. The downside is you must remember to invalidate everywhere the data could be cached.

Strategy 3: Write-Through Cache

Update the cache at the same time as the database:

```
@put("/api/products/{product_id}")
async def update_product(request, response):
    product_id = request.params["product_id"]
    body = request.body
    db = Database.get_connection()

    db.execute(
```

```
        "UPDATE products SET name = ?, price = ? WHERE id = ?",
        [body["name"], body["price"], product_id]
    )

    updated = db.fetch_one("SELECT * FROM products WHERE id = ?", [product_id])

    # Write the new data directly to cache (instead of deleting)
    cache_set(f"product:{product_id}", updated, ttl=600)

    return response(updated)
```

This ensures the cache always has the latest data. No cache miss after an update -- the next read comes from the already-warm cache.

---

9. TTL Management

Choosing the right TTL depends on how often the data changes and how acceptable stale data is:

Data Type	Suggested TTL	Reasoning
Static config (categories, countries)	3600 (1 hour)	Changes rarely, stale data is harmless
Product catalog	300 (5 min)	Updates several times per day
User profile	60 (1 min)	Users expect changes to appear quickly
Search results	120 (2 min)	Balance between freshness and performance
Dashboard stats	30 (30 sec)	Near-real-time but expensive to compute
Exchange rates	60 (1 min)	Updates frequently, slight delay is acceptable
Shopping cart	0 (no cache)	Must always reflect current state

Dynamic TTL

Adjust TTL based on data characteristics:

```
from tina4_python.cache import cache_get, cache_set

def get_cached_product(product_id):
    cache_key = f"product:{product_id}"
    product = cache_get(cache_key)

    if product is not None:
        return product
```

```
    product = fetch_product_from_database(product_id)

    # Popular products: shorter TTL (more likely to change)
    # Inactive products: longer TTL (rarely change)
    ttl = 60 if product["view_count"] > 1000 else 3600

    cache_set(cache_key, product, ttl=ttl)

    return product
```

---

10. Cache Statistics

Monitor cache performance to verify that caching is actually helping:

```
from tina4_python.core.router import get
from tina4_python.cache import cache_stats

@get("/api/cache/stats")
async def get_cache_stats(request, response):
    stats = cache_stats()
    return response(stats)
```

```
curl http://localhost:7146/api/cache/stats
```

```
{
  "backend": "memory",
  "entries": 42,
  "hits": 15234,
  "misses": 891,
  "hit_rate": "94.5%",
  "memory_bytes": 524288,
  "oldest_entry_age": 3542
}
```

Hit rate above 90%: your caching strategy works. Below 80%: TTLs are too short, cache is too small, or you are caching data that is not accessed often enough to benefit.

The dev dashboard at `/__dev` shows cache statistics too -- per-key hit counts and miss counts. You see which keys earn their keep.

---

11. Combining Cache Layers

For maximum performance, layer multiple cache strategies:

```
from datetime import datetime, timezone
from tina4_python.core.router import get
from tina4_python.cache import cache_get, cache_set
import math

@get("/api/catalog", middleware=["ResponseCache:60"])
async def get_catalog(request, response):
    page = int(request.params.get("page", 1))
    cache_key = f"catalog:page:{page}"

    # Layer 1: Check application cache
```

```
    catalog = cache_get(cache_key)

    if catalog is not None:
        return response({**catalog, "cache": "application"})

    # Layer 2: Database query (with DB-level caching if TINA4_DB_CACHE=true)
    db = Database.get_connection()
    limit = 20
    offset = (page - 1) * limit

    products = db.fetch_all(
        """SELECT p.*, c.name as category_name
           FROM products p
           JOIN categories c ON p.category_id = c.id
           WHERE p.active = 1
           ORDER BY p.created_at DESC
           LIMIT ? OFFSET ?""",
        [limit, offset]
    )

    total = db.fetch_one("SELECT COUNT(*) as count FROM products WHERE active = 1")

    catalog = {
        "products": products,
        "page": page,
        "total": total["count"],
        "pages": math.ceil(total["count"] / limit),
        "generated_at": datetime.now(timezone.utc).isoformat()
    }

    # Store in application cache
    cache_set(cache_key, catalog, ttl=300)

    return response({**catalog, "cache": "none"})
```

This creates three cache layers:

- **ResponseCache** (60 seconds): The entire HTTP response is cached. No Python code runs at all.
- **Application cache** (300 seconds): If the response cache expired but the app cache is still fresh, skip the database queries.
- **DB query cache** (if enabled): Individual query results are cached even if the application cache missed.

The first visitor after a full cache expiry waits 800ms. Everyone else gets the response in under 5ms.

---

12. Exercise: Cache an Expensive Product Listing Endpoint

Build a product listing endpoint that uses caching at multiple levels.

Requirements

- Create a `GET /api/store/products` endpoint that:

- Accepts query parameters: `category`, `page`, `limit`
- Returns a list of products with pagination metadata
- Uses the direct cache API (`cache_get`/`cache_set`) with a 5-minute TTL
- Includes a `source` field in the response (`"cache"` or `"database"`)
- Create a `POST /api/store/products` endpoint that:
- Creates a new product
- Invalidates the relevant cache entries
- Create a `GET /api/store/cache-stats` endpoint that shows cache statistics

Test with:

```
# First call -- cache miss, slow
curl "http://localhost:7146/api/store/products?category=Electronics&page=1"

# Second call -- cache hit, fast
curl "http://localhost:7146/api/store/products?category=Electronics&page=1"

# Different category -- cache miss
curl "http://localhost:7146/api/store/products?category=Fitness&page=1"

# Create a product -- should invalidate cache
curl -X POST http://localhost:7146/api/store/products \
  -H "Content-Type: application/json" \
  -d '{"name": "Smart Watch", "category": "Electronics", "price": 299.99}'

# Same query again -- cache miss (invalidated by the POST)
curl "http://localhost:7146/api/store/products?category=Electronics&page=1"

# Check cache stats
curl http://localhost:7146/api/store/cache-stats
```

---

13. Solution

Create `src/routes/store_cached.py`:

```
import json
import time
import hashlib
from datetime import datetime, timezone
from tina4_python.core.router import get, post
from tina4_python.cache import cache_get, cache_set, cache_delete, cache_stats


def get_product_store():
    return [
        {"id": 1, "name": "Wireless Keyboard", "category": "Electronics", "price": 79.99, "in_st
        {"id": 2, "name": "Yoga Mat", "category": "Fitness", "price": 29.99, "in_stock": True},
        {"id": 3, "name": "Coffee Grinder", "category": "Kitchen", "price": 49.99, "in_stock": F
        {"id": 4, "name": "Standing Desk", "category": "Electronics", "price": 549.99, "in_stock
        {"id": 5, "name": "Running Shoes", "category": "Fitness", "price": 119.99, "in_stock": T
        {"id": 6, "name": "Bluetooth Speaker", "category": "Electronics", "price": 39.99, "in_st
```

```
        {"id": 7, "name": "Resistance Bands", "category": "Fitness", "price": 14.99, "in_stock":
        {"id": 8, "name": "French Press", "category": "Kitchen", "price": 34.99, "in_stock": Tru
    ]


@get("/api/store/products")
async def list_store_products(request, response):
    category = request.params.get("category")
    page = int(request.params.get("page", 1))
    limit = int(request.params.get("limit", 20))

    # Build cache key from query parameters
    key_data = json.dumps({"category": category, "page": page, "limit": limit})
    cache_key = f"store:products:{hashlib.md5(key_data.encode()).hexdigest()}"

    # Try cache first
    cached = cache_get(cache_key)

    if cached is not None:
        return response({**cached, "source": "cache"})

    # Simulate expensive database query
    time.sleep(0.1)  # 100ms delay

    products = get_product_store()

    # Filter by category
    if category is not None:
        products = [p for p in products if p["category"].lower() == category.lower()]

    total = len(products)
    offset = (page - 1) * limit
    products = products[offset:offset + limit]

    import math
    result = {
        "products": products,
        "page": page,
        "limit": limit,
        "total": total,
        "pages": math.ceil(total / limit),
        "generated_at": datetime.now(timezone.utc).isoformat()
    }

    # Cache for 5 minutes
    cache_set(cache_key, result, ttl=300)

    return response({**result, "source": "database"})


@post("/api/store/products")
async def create_store_product(request, response):
    body = request.body

    if not body.get("name"):
        return response({"error": "Name is required"}, 400)

    import random
    product = {
```

```
        "id": random.randint(100, 9999),
        "name": body["name"],
        "category": body.get("category", "General"),
        "price": float(body.get("price", 0)),
        "in_stock": True
    }

    # Invalidate all product list caches
    categories = ["Electronics", "Fitness", "Kitchen", "General"]
    for cat in categories:
        for p in range(1, 6):
            key_data = json.dumps({"category": cat, "page": p, "limit": 20})
            cache_delete(f"store:products:{hashlib.md5(key_data.encode()).hexdigest()}")

    # Also invalidate the unfiltered list
    for p in range(1, 6):
        key_data = json.dumps({"category": None, "page": p, "limit": 20})
        cache_delete(f"store:products:{hashlib.md5(key_data.encode()).hexdigest()}")

    return response({
        "message": "Product created",
        "product": product,
        "cache_invalidated": True
    }, 201)


@get("/api/store/cache-stats")
async def store_cache_stats(request, response):
    return response(cache_stats())
```

Expected output -- first call (cache miss):

```
curl "http://localhost:7146/api/store/products?category=Electronics&page=1"
```


```
{
  "products": [
    {"id": 1, "name": "Wireless Keyboard", "category": "Electronics", "price": 79.99, "in_stock"
    {"id": 4, "name": "Standing Desk", "category": "Electronics", "price": 549.99, "in_stock": t
    {"id": 6, "name": "Bluetooth Speaker", "category": "Electronics", "price": 39.99, "in_stock"
  ],
  "page": 1,
  "limit": 20,
  "total": 3,
  "pages": 1,
  "generated_at": "2026-03-22T14:30:00+00:00",
  "source": "database"
}
```


Expected output -- second call (cache hit):


```
{
  "products": [
    {"id": 1, "name": "Wireless Keyboard", "category": "Electronics", "price": 79.99, "in_stock"
    {"id": 4, "name": "Standing Desk", "category": "Electronics", "price": 549.99, "in_stock": t
    {"id": 6, "name": "Bluetooth Speaker", "category": "Electronics", "price": 39.99, "in_stock"
  ],
  "page": 1,
  "limit": 20,
  "total": 3,
  "pages": 1,
```

```
  "generated_at": "2026-03-22T14:30:00+00:00",
  "source": "cache"
}
```

Notice: same `generated_at`, but `source` changed from `"database"` to `"cache"`. The handler did not run.

---

14. Gotchas

1. Caching Authenticated Responses

Problem: User A's profile is served to User B because the response was cached.

Cause: `ResponseCache` caches by URL only. If `/api/profile` returns different data per user but all requests hit the same URL, the first user's response is served to everyone.

Fix: Do not use `ResponseCache` on user-specific endpoints. Use the direct cache API with user-specific keys instead: `cache_set(f"profile:{user_id}", data, ttl=300)`.

2. Cache Stampede

Problem: When a popular cache key expires, hundreds of requests hit the database simultaneously (all experiencing a cache miss at the same moment).

Cause: All requests see the cache miss and all try to rebuild the cache independently.

Fix: Use cache locking or "stale-while-revalidate". One request rebuilds the cache while others serve the stale value. Tina4's `ResponseCache` handles this automatically by serving the expired response to concurrent requests while one request refreshes it.

3. Memory Cache Lost on Restart

Problem: After restarting the server, performance drops until the cache warms up.

Cause: Memory cache is lost when the process restarts. Every request is a cache miss until data is cached again.

Fix: For production, use Redis cache (`TINA4_CACHE_BACKEND=redis`). It persists across server restarts. Alternatively, implement a cache warmup script that pre-populates the cache with frequently accessed data.

4. Stale Data After Database Update

Problem: You updated a product's price in the database, but the API still returns the old price.

Cause: The cache still has the old data and has not expired yet.

Fix: Always invalidate (or update) the cache when you modify the underlying data. Use `cache_delete(f"product:{product_id}")` after an update, or use write-through caching with `cache_set()` to update the cache with the new value.

5. Cache Key Collisions

Problem: Two different queries return the same cached data.

Cause: Your cache keys are not specific enough. Using `"products"` as a key for both the full list and a filtered list causes collisions.

Fix: Include all relevant parameters in the cache key: `"products:category:Electronics:page:1:limit:20"`. Or use an MD5 hash of the parameters: `f"products:{hashlib.md5(json.dumps(params).encode()).hexdigest()}"`.

6. Serialization Overhead

Problem: Caching makes certain requests slower, not faster.

Cause: The cached object is very large. Serializing and deserializing it takes more time than re-computing it.

Fix: Only cache data that is expensive to compute. If the original operation takes 5ms and cache serialization takes 10ms, caching is counterproductive. Profile before and after caching to verify the improvement.

7. Forgetting to Set TTL

Problem: Cache entries never expire and the server's memory grows until it crashes.

Cause: You called `cache_set("key", value)` without a TTL. The entry lives forever (or until the server restarts).

Fix: Always set a TTL: `cache_set("key", value, ttl=300)`. Even for data that "never changes," set a long TTL like 86400 (24 hours). This provides a safety net against stale data and prevents unbounded memory growth.

Queues

1. Not Everything Should Happen Right Now

Some tasks are too slow for an HTTP request. Sending an email: 2 seconds. Generating a PDF report: 10 seconds. Processing a large CSV upload: a minute. Run these inside a route handler and the user stares at a spinner. Or the request times out.

Queues solve this. Push a message describing the work. A separate worker picks it up and does the job in the background. The user gets an instant response: "Your report is being generated."

Picture a store that sends order confirmations, generates invoices, and syncs inventory with a warehouse. None of these should block checkout. Each one becomes a queue message. A worker processes it on its own schedule.

---

2. Queue Configuration

Tina4 Python includes a built-in queue that requires zero additional setup. The default backend is file-based.

Default (File Queue)

The file queue works out of the box with no extra configuration:

```
# No TINA4_QUEUE_BACKEND needed -- file is the default
```

The first time you push a message, Tina4 automatically creates the queue storage.

Switching to RabbitMQ

When your application outgrows the file queue (millions of messages, multiple services, pub/sub patterns), switch to RabbitMQ:

```
TINA4_QUEUE_BACKEND=rabbitmq
TINA4_QUEUE_URL=amqp://user:pass@localhost:5672
```

Install the client library:

```
uv add pika
```

Switching to Kafka

For stream processing, event sourcing, or very high throughput:

```
TINA4_QUEUE_BACKEND=kafka
TINA4_QUEUE_URL=localhost:9092
```

Install the client library:

```
uv add confluent-kafka
```

Switching to MongoDB

```
TINA4_QUEUE_BACKEND=mongodb
TINA4_QUEUE_URL=mongodb://user:pass@localhost:27017/tina4
```

Install the driver:

```
uv add pymongo
```

The key point: your code stays the same. The `Queue` class, `push`, `pop`, and `consume` work identically whether the backend is file, RabbitMQ, Kafka, or MongoDB. The backend is configured via environment variables.

---

3. Creating a Queue and Pushing Messages

```
from tina4_python.queue import Queue

queue = Queue(topic="emails")

# Push a message
message_id = queue.push({
    "to": "alice@example.com",
    "subject": "Order Confirmation",
    "body": "Your order #1234 has been confirmed."
})
```

The `topic` argument names the queue. The payload is any dictionary that can be serialized to JSON.

> ***Backend configuration:*** *The queue backend is selected via environment variables, not constructor parameters. Set* `TINA4_QUEUE_BACKEND` *to* `file` *(default),* `rabbitmq`*,* `kafka`*, or* `mongodb`*. For the file backend, the* `TINA4_QUEUE_PATH` *environment variable controls the storage directory (default: data/queue). See Section 2 and Section 8 for full details.*

Convenience Method: produce

The `produce` method pushes to a specific topic without creating a separate Queue instance:

```
queue = Queue(topic="emails")
queue.produce("invoices", {"order_id": 101, "format": "pdf"})
```

Queue Size

Check how many pending messages are in the queue:

```
count = queue.size()
```

---

4. Consuming Messages

The consume Pattern

The `consume` method is a generator that yields jobs one at a time. Each job must be explicitly completed or failed:

```
from tina4_python.queue import Queue

queue = Queue(topic="emails")

for job in queue.consume("emails"):
    try:
        send_email(job.payload["to"], job.payload["subject"], job.payload["body"])
        job.complete()
    except Exception as e:
        job.fail(str(e))
```

You can also filter by job ID:

```
for job in queue.consume("emails", id="specific-job-id"):
    # Process only the job with that ID
    job.complete()
```

Manual Pop

For more control, pop a single message:

```
job = queue.pop()

if job is not None:
    try:
        send_email(job.payload["to"], job.payload["subject"])
        job.complete()
    except Exception as e:
        job.fail(str(e))
```

---

5. Job Lifecycle

A job moves through these statuses:

```
push() -> PENDING -> pop()/consume() -> RESERVED -> job.complete() -> COMPLETED
                                                  -> job.fail()    -> FAILED
                                                                        |
                                                                  retry (manual)
                                                                        |
                                                                    PENDING
                                                                        |
                                                              max retries exceeded
                                                                        |
                                                                   DEAD LETTER
```

Job Methods

When you receive a job from `consume` or `pop`, you have three methods:

- `job.complete()` -- mark the job as done

- `job.fail(reason)` -- mark the job as failed with a reason string
- `job.reject(reason)` -- alias for `fail`

Always call one of these. If you do not, the job stays reserved.

---

6. Retry and Dead Letters

Max Retries

The `Queue` constructor accepts a `max_retries` parameter (default: 3). When a job's attempt count reaches `max_retries`, `retry_failed()` skips it.

```
queue = Queue(topic="emails", max_retries=5)
```

Retrying Failed Jobs

```
# Retry a specific job by ID
queue.retry(job_id)

# Retry all failed jobs (skips those that exceeded max_retries)
queue.retry_failed()
```

Dead Letters

Jobs that have exceeded `max_retries` are dead letters. There is no magic dead letter queue -- you retrieve and handle them yourself:

```
dead_jobs = queue.dead_letters()

for job in dead_jobs:
    print(f"Dead job: {job.id}")
    print(f"  Payload: {job.data}")
    print(f"  Error: {job.error}")
```

Purging Jobs

Remove jobs by status:

```
queue.purge("completed")
queue.purge("failed")
```

---

7. Queue in Route Handlers

The most common pattern is pushing messages from route handlers:

```
from tina4_python.router import get, post
from tina4_python.queue import Queue

queue = Queue(topic="emails")

@post("/api/orders")
async def create_order(request, response):
    body = request.body
```

```
    # Create the order in the database
    order_id = 101  # Simulated

    # Send confirmation email
    queue.push({
        "type": "order_confirmation",
        "to": body["email"],
        "order_id": order_id,
        "total": body["total"]
    })

    # Generate invoice on a different topic
    queue.produce("invoices", {
        "order_id": order_id,
        "format": "pdf"
    })

    # Sync with warehouse on a different topic
    queue.produce("warehouse_sync", {
        "order_id": order_id,
        "items": body["items"]
    })

    return response.json({
        "message": "Order created",
        "order_id": order_id
    }, 201)
```

The user gets an instant response. The email, invoice, and warehouse sync happen in the background.

---

8. Switching Backends via .env

Switching backends is a config change, not a code change.

Development: File (default)

```
# No config needed -- file is the default
```

Production: RabbitMQ

```
TINA4_QUEUE_BACKEND=rabbitmq
TINA4_QUEUE_URL=amqp://user:pass@rabbitmq.internal:5672
```

High-Scale Production: Kafka

```
TINA4_QUEUE_BACKEND=kafka
TINA4_QUEUE_URL=kafka-1:9092,kafka-2:9092,kafka-3:9092
```

Production: MongoDB

```
TINA4_QUEUE_BACKEND=mongodb
TINA4_QUEUE_URL=mongodb://user:pass@mongo.internal:27017/tina4
```

Your queue code does not change at all. The same `queue.push()` and `queue.consume()` calls work with every backend.

---

9. Produce and Consume Across Topics

The `Queue` class provides `produce()` and `consume()` methods for cross-topic messaging:

```
from tina4_python.queue import Queue

queue = Queue(topic="default")

# Produce onto a specific topic
queue.produce("emails", {"to": "alice@example.com", "subject": "Hello"})

# Consume from a specific topic
for job in queue.consume("emails"):
    process(job)
    job.complete()
```

The `produce()` method pushes a job onto any named topic. The `consume()` method yields all available jobs from a topic as a generator.

---

10. Exercise: Build an Email Queue

Build an email queue system that sends emails in the background, including failure handling.

Requirements

- Create these endpoints:

Method	Path	Description
`POST`	`/api/emails/send`	Queue an email for sending
`GET`	`/api/emails/queue`	List pending email count
`GET`	`/api/emails/dead`	List dead letter jobs
`POST`	`/api/emails/retry`	Retry all failed jobs

- The email payload should include: `to` (required), `subject` (required), `body` (required)
- Create a consumer that processes the queue, simulating occasional failures
- When an email to a specific address fails repeatedly, it should end up in dead letters

Test with:

```
# Queue an email
curl -X POST http://localhost:7146/api/emails/send \
  -H "Content-Type: application/json" \
  -d '{"to": "alice@example.com", "subject": "Welcome!", "body": "Thanks for signing up."}'

# Check queue size
curl http://localhost:7146/api/emails/queue

# Check dead letters
curl http://localhost:7146/api/emails/dead

# Retry failed
curl -X POST http://localhost:7146/api/emails/retry
```

---

11. Solution

Create `src/routes/email_queue.py`:

```
from tina4_python.router import get, post
from tina4_python.queue import Queue

queue = Queue(topic="emails", max_retries=3)


@post("/api/emails/send")
async def queue_email(request, response):
    body = request.body

    errors = []
    if not body.get("to"):
        errors.append("'to' is required")
    if not body.get("subject"):
        errors.append("'subject' is required")
    if not body.get("body"):
        errors.append("'body' is required")

    if errors:
        return response.json({"errors": errors}, 400)

    message_id = queue.push({
        "to": body["to"],
        "subject": body["subject"],
        "body": body["body"]
    })

    return response.json({
        "message": "Email queued for sending",
        "message_id": message_id
    }, 201)


@get("/api/emails/queue")
async def email_queue_size(request, response):
    count = queue.size()
    return response.json({"pending": count})
```

```
@get("/api/emails/dead")
async def email_dead_letters(request, response):
    dead = queue.dead_letters()
    items = []
    for job in dead:
        items.append({
            "id": job.id,
            "payload": job.data,
            "error": job.error
        })
    return response.json({"dead_letters": items, "count": len(items)})


@post("/api/emails/retry")
async def retry_failed_emails(request, response):
    queue.retry_failed()
    return response.json({"message": "Failed emails re-queued for retry"})
```

Create a separate consumer file `src/workers/email_worker.py`:

```
from tina4_python.queue import Queue
import time

queue = Queue(topic="emails", max_retries=3)

for job in queue.consume("emails"):
    payload = job.data

    print(f"Sending email to {payload['to']}...")
    print(f"  Subject: {payload['subject']}")
    print(f"  Body: {payload['body'][:50]}...")

    try:
        # Simulate sending (replace with real email logic)
        time.sleep(1)

        # Simulate failure for a specific address
        if payload["to"] == "bad@example.com":
            raise Exception("SMTP connection refused")

        print(f"  Email sent to {payload['to']} successfully!")
        job.complete()

    except Exception as e:
        print(f"  Failed: {e}")
        job.fail(str(e))
```

After the consumer has retried a job to `bad@example.com` three times, `queue.dead_letters()` returns that job. The `/api/emails/dead` endpoint shows it. You investigate, fix the address, and call `/api/emails/retry` to re-queue.

---

12. Gotchas

1. Always call complete or fail

Problem: Jobs stay in reserved status forever.

Cause: Your consumer does not call `job.complete()` or `job.fail()`. The job stays reserved and is never released.

Fix: Always call one of `job.complete()`, `job.fail(reason)`, or `job.reject(reason)` in your consumer loop.

2. Worker not picking up messages

Problem: Messages are pushed but nothing happens.

Cause: No consumer process is running, or the consumer is listening on a different topic.

Fix: Make sure the consumer is running. Check that the topic name in `queue.push()` matches the topic name in `queue.consume()`.

3. Payload too large

Problem: Pushing a message with a large payload is slow.

Cause: The payload is serialized to JSON and stored in the backend. Very large payloads slow down the queue.

Fix: Keep payloads small. Store files on disk or in object storage and put the file path in the payload. Payloads should be metadata, not data.

4. Dead letters pile up

Problem: Dead letters accumulate and nobody notices.

Cause: Failed jobs that exceed `max_retries` become dead letters but are never cleaned up.

Fix: Monitor dead letters with `queue.dead_letters()`. Set up an alert when the count exceeds a threshold. Investigate the root cause, fix it, then call `queue.retry_failed()` or `queue.purge("failed")`.

5. File backend for production

Problem: Multiple workers cause contention on file-based queue storage.

Cause: File-based storage supports one writer at a time.

Fix: For production with multiple workers, switch to RabbitMQ, Kafka, or MongoDB via the `TINA4_QUEUE_BACKEND` environment variable. The file backend is fine for development and single-worker setups.

6. Environment-specific topic collision

Problem: Development and staging environments process each other's messages.

Cause: Both environments use the same backend and the same topic names.

Fix: Use separate backends per environment, or prefix topic names with the environment: `Queue(topic="dev_emails")`.

Events

1. Decouple Everything with Events

Your order route creates an order, sends a confirmation email, deducts inventory, and notifies the warehouse. Six hundred lines of intertwined logic. Add a new requirement — loyalty points — and you are touching the order route again.

Events flip this around. The order route fires `order.placed` and walks away. Separate handlers pick it up — one sends the email, another deducts inventory, a third notifies the warehouse. None of them know about each other.

Tina4's event system is synchronous by default and supports async via `emit_async`. No broker required.

---

2. Registering Handlers with @on

```
from tina4_python.core.events import on, emit

@on("user.created")
def welcome_email(payload):
    print(f"Sending welcome email to {payload['email']}")

@on("user.created")
def create_profile(payload):
    print(f"Creating profile for {payload['user_id']}")
```

Register as many handlers as you need for the same event. They all run when the event fires.

---

3. Firing Events with emit

```
from tina4_python.core.router import post
from tina4_python.core.events import emit

@post("/api/users")
async def register_user(request, response):
    body = request.body

    user = {
        "user_id": 42,
        "email": body["email"],
        "name": body["name"]
    }

    # Fire the event -- all registered handlers run now
    emit("user.created", user)

    return response({"message": "User created", "user_id": user["user_id"]}, 201)
```

```
curl -X POST http://localhost:7146/api/users \
  -H "Content-Type: application/json" \
```

```
  -d '{"email": "alice@example.com", "name": "Alice"}'
```

Output in the server log:

```
Sending welcome email to alice@example.com
Creating profile for 42
```

---

4. One-Time Handlers with @once

A handler registered with `once` fires exactly once and then automatically unregisters itself:

```
from tina4_python.core.events import once, emit

@once("app.started")
def run_migrations(payload):
    print("Running database migrations...")

emit("app.started", {})   # Runs run_migrations
emit("app.started", {})   # Does nothing -- handler was removed
```

Useful for initialisation tasks, first-visit tracking, or features that should only trigger once per session.

---

5. Removing Handlers with off

Remove a handler by function reference:

```
from tina4_python.core.events import on, off, emit

def log_purchase(payload):
    print(f"Purchase logged: {payload['order_id']}")

on("order.placed", log_purchase)

emit("order.placed", {"order_id": 101})  # Handler runs

off("order.placed", log_purchase)

emit("order.placed", {"order_id": 102})  # Handler does not run
```

---

6. Priority Ordering

When multiple handlers listen to the same event, you control which one runs first using the `priority` parameter. Lower numbers run first. Default priority is `10`.

```
from tina4_python.core.events import on, emit

@on("order.placed", priority=1)
def validate_stock(payload):
    print(f"[priority 1] Validating stock for order {payload['order_id']}")

@on("order.placed", priority=5)
```

```
def charge_payment(payload):
    print(f"[priority 5] Charging payment for order {payload['order_id']}")

@on("order.placed", priority=10)
def send_confirmation(payload):
    print(f"[priority 10] Sending confirmation for order {payload['order_id']}")

emit("order.placed", {"order_id": 200})
```

Output:

```
[priority 1] Validating stock for order 200
[priority 5] Charging payment for order 200
[priority 10] Sending confirmation for order 200
```

Stock is validated before payment is charged, payment is charged before the confirmation email goes out.

---

7. Async Event Support with emit_async

For handlers that do I/O — sending HTTP requests, writing to a database, publishing to a message broker — use `emit_async` with async handlers:

```
import asyncio
from tina4_python.core.events import on, emit_async

@on("order.placed")
async def notify_warehouse(payload):
    # Simulate async HTTP call to warehouse API
    await asyncio.sleep(0.1)
    print(f"Warehouse notified for order {payload['order_id']}")

@on("order.placed")
async def update_analytics(payload):
    await asyncio.sleep(0.05)
    print(f"Analytics updated for order {payload['order_id']}")
```

Call `emit_async` from an async route handler:

```
from tina4_python.core.router import post
from tina4_python.core.events import emit_async

@post("/api/orders")
async def create_order(request, response):
    body = request.body

    order = {
        "order_id": 300,
        "customer_id": body["customer_id"],
        "total": body["total"],
        "items": body["items"]
    }

    # All async handlers run concurrently
    await emit_async("order.placed", order)

    return response({"message": "Order placed", "order_id": order["order_id"]}, 201)
```

`emit_async` gathers all async handlers and runs them concurrently using `asyncio.gather`. Synchronous handlers registered on the same event run first (in priority order), then all async handlers run in parallel.

---

8. Real-World Pattern: Order Pipeline

A realistic e-commerce order with five independent concerns:

```
from tina4_python.core.events import on, emit_async

@on("order.placed", priority=1)
def reserve_stock(payload):
    print(f"Reserving {len(payload['items'])} items")

@on("order.placed", priority=2)
def charge_customer(payload):
    print(f"Charging ${payload['total']} to customer {payload['customer_id']}")

@on("order.placed", priority=5)
async def send_confirmation_email(payload):
    import asyncio
    await asyncio.sleep(0)  # Yield to event loop
    print(f"Confirmation sent to {payload['email']}")

@on("order.placed", priority=5)
async def sync_warehouse(payload):
    import asyncio
    await asyncio.sleep(0)
    print(f"Warehouse sync queued for order {payload['order_id']}")

@on("order.placed", priority=10)
def award_loyalty_points(payload):
    points = int(payload['total'] * 10)
    print(f"Awarding {points} loyalty points to customer {payload['customer_id']}")
```

The order route:

```
from tina4_python.core.router import post
from tina4_python.core.events import emit_async

@post("/api/orders")
async def place_order(request, response):
    body = request.body

    order = {
        "order_id": 301,
        "customer_id": body["customer_id"],
        "email": body["email"],
        "total": body["total"],
        "items": body["items"]
    }

    await emit_async("order.placed", order)

    return response({"order_id": order["order_id"], "status": "placed"}, 201)
```

Adding loyalty points or a fraud check means adding a new `@on` handler. The route file never changes.

---

9. Exercise: User Lifecycle Events

Build a user registration flow using events.

Requirements

- Create a `POST /api/register` endpoint that:
- Accepts `email`, `name`, `password`
- Emits `user.created` with `{user_id, email, name}`
- Returns `{"user_id": ..., "status": "created"}`
- Register three handlers for `user.created`:
- Priority 1: log the event with timestamp
- Priority 2: simulate sending a welcome email
- Priority 5: simulate creating a default profile
- Create a `POST /api/users/{user_id}/deactivate` endpoint that emits `user.deactivated`
- Register a one-time handler for `app.ready` that prints a startup message

Test with:

```
# Register a user
curl -X POST http://localhost:7146/api/register \
  -H "Content-Type: application/json" \
  -d '{"email": "bob@example.com", "name": "Bob", "password": "secret"}'

# Deactivate a user
curl -X POST http://localhost:7146/api/users/42/deactivate
```

---

10. Solution

Create `src/routes/user_events.py`:

```
from datetime import datetime, timezone
from tina4_python.core.router import post
from tina4_python.core.events import on, once, emit, emit_async


@once("app.ready")
def on_app_ready(payload):
    print(f"[{datetime.now(timezone.utc).isoformat()}] Application ready.")


@on("user.created", priority=1)
def log_user_created(payload):
```

```
    print(f"[{datetime.now(timezone.utc).isoformat()}] user.created: {payload['email']}")


@on("user.created", priority=2)
def send_welcome_email(payload):
    print(f"Welcome email dispatched to {payload['email']}")


@on("user.created", priority=5)
def create_default_profile(payload):
    print(f"Default profile created for user {payload['user_id']}")


@on("user.deactivated")
def log_deactivation(payload):
    print(f"User {payload['user_id']} deactivated at {datetime.now(timezone.utc).isoformat()}")


@post("/api/register")
async def register_user(request, response):
    body = request.body

    if not body.get("email") or not body.get("name"):
        return response({"error": "email and name are required"}, 400)

    user = {
        "user_id": 42,
        "email": body["email"],
        "name": body["name"]
    }

    emit("user.created", user)

    return response({"user_id": user["user_id"], "status": "created"}, 201)


@post("/api/users/{user_id}/deactivate")
async def deactivate_user(request, response):
    user_id = request.params["user_id"]

    emit("user.deactivated", {"user_id": user_id})

    return response({"user_id": user_id, "status": "deactivated"})
```

Fire the app ready event once on startup in `src/app.py`:

```
from tina4_python.core.events import emit
emit("app.ready", {})
```

---

11. Gotchas

1. Emitting from a sync context with async handlers

Problem: `emit()` is called from a synchronous function but some handlers are `async def`. They will not run.

Fix: Use `emit_async()` from an async context for events that have async handlers. Synchronous handlers always run regardless of which emit variant you use.

2. Handler order is non-deterministic at equal priority

Problem: Two handlers at `priority=5` run in a different order than expected.

Fix: Give them distinct priority values. Priority uniqueness within an event gives you guaranteed ordering.

3. Forgetting once fires exactly once

Problem: An initialisation handler registered with `@once` does not run on the second app start during hot reload.

Fix: `@once` handlers remove themselves after firing. For handlers that must run every startup, use `@on` instead.

4. Mutating the payload in a handler

Problem: A high-priority handler modifies the payload dict and downstream handlers see the mutated version.

Fix: This is intentional and useful (e.g., adding `payment_id` to the payload before the confirmation handler runs). If you want to prevent mutation, pass `payload.copy()` or use a frozen data structure.

Localization

1. One App, Many Languages

Your application is live. A customer in Berlin arrives and sees English. A customer in Tokyo sees English. A customer in São Paulo sees English.

Tina4 localization lets you write translations once and switch the active locale at runtime. Store JSON files in `src/locales/`, call `t()` at the point of use, and set the active language from a query parameter, a cookie, or an environment variable. No third-party libraries required.

---

2. Locale Files

Create one JSON file per locale in `src/locales/`. The filename is the locale code.

```
src/
  locales/
    en.json
    de.json
    ja.json
    pt.json
```

`src/locales/en.json`:

```
{
  "welcome": "Welcome",
  "greeting": "Hello, {name}!",
  "order": {
    "placed": "Your order has been placed.",
    "total": "Order total: {currency}{amount}",
    "item_count": "{count} item in your order",
    "item_count_plural": "{count} items in your order"
  },
  "errors": {
    "not_found": "The requested resource was not found.",
    "unauthorized": "You are not authorized to access this resource."
  }
}
```

`src/locales/de.json`:

```
{
  "welcome": "Willkommen",
  "greeting": "Hallo, {name}!",
  "order": {
    "placed": "Ihre Bestellung wurde aufgegeben.",
    "total": "Bestellsumme: {currency}{amount}",
    "item_count": "{count} Artikel in Ihrer Bestellung",
    "item_count_plural": "{count} Artikel in Ihrer Bestellung"
  },
  "errors": {
    "not_found": "Die angeforderte Ressource wurde nicht gefunden.",
    "unauthorized": "Sie sind nicht berechtigt, auf diese Ressource zuzugreifen."
  }
```

```
}
```

---

3. The t() Function

```
from tina4_python.i18n import t

# Simple key
message = t("welcome")
# "Welcome"

# Nested key with dot notation
error = t("errors.not_found")
# "The requested resource was not found."

# Interpolation with {placeholder}
greeting = t("greeting", name="Alice")
# "Hello, Alice!"

order_total = t("order.total", currency="$", amount="99.99")
# "Order total: $99.99"
```

Keys use dot notation for nested structures. Placeholders in curly braces are replaced with keyword arguments.

---

4. The I18n Class

For more control — setting the locale per request, loading custom locale paths — use the `I18n` class directly:

```
from tina4_python.i18n import I18n

i18n = I18n(locale="de", path="src/locales")

print(i18n.t("welcome"))
# "Willkommen"

print(i18n.t("greeting", name="Max"))
# "Hallo, Max!"

# Switch locale at runtime
i18n.set_locale("en")
print(i18n.t("welcome"))
# "Welcome"
```

The `I18n` class loads locale files lazily. Setting the locale to `"de"` loads `src/locales/de.json` the first time a translation is requested. Subsequent requests for the same locale use the in-memory cache.

---

5. Setting the Locale at Runtime

From a Query Parameter

The simplest approach: read `?lang=` from the request and set the locale.

```
from tina4_python.core.router import get
from tina4_python.i18n import I18n

@get("/api/welcome")
async def welcome(request, response):
    lang = request.params.get("lang", "en")
    i18n = I18n(locale=lang, path="src/locales")

    return response({
        "message": i18n.t("greeting", name="World"),
        "locale": lang
    })
```

```
curl "http://localhost:7146/api/welcome?lang=en"
# {"message": "Hello, World!", "locale": "en"}

curl "http://localhost:7146/api/welcome?lang=de"
# {"message": "Hallo, World!", "locale": "de"}
```

From the Accept-Language Header

A more standard approach uses the browser's language preference:

```
from tina4_python.core.router import get
from tina4_python.i18n import I18n

def detect_locale(request, default="en"):
    accept = request.headers.get("Accept-Language", default)
    # "de-DE,de;q=0.9,en;q=0.8" -> "de"
    primary = accept.split(",")[0].split("-")[0].strip()
    supported = ["en", "de", "ja", "pt"]
    return primary if primary in supported else default

@get("/api/dashboard")
async def dashboard(request, response):
    locale = detect_locale(request)
    i18n = I18n(locale=locale, path="src/locales")

    return response({
        "welcome": i18n.t("welcome"),
        "locale": locale
    })
```

From a Session or Cookie

```
@get("/api/home")
async def home(request, response):
    locale = request.session.get("locale", "en")
    i18n = I18n(locale=locale, path="src/locales")

    return response({"message": i18n.t("welcome")})

@post("/api/locale")
async def set_locale(request, response):
    locale = request.body.get("locale", "en")
    request.session["locale"] = locale
    return response({"locale": locale})
```

---

6. The TINA4_LOCALE Environment Variable

Set the application's default locale in `.env`:

```
TINA4_LOCALE=de
```

When `TINA4_LOCALE` is set, calls to `t()` without an explicit locale use German as the default. This is the locale loaded at application startup.

```
from tina4_python.i18n import t

# Uses TINA4_LOCALE from .env (e.g., "de")
message = t("welcome")
# "Willkommen"
```

---

7. Interpolation with {placeholder}

Placeholders are written as `{name}` in the JSON file and filled via keyword arguments to `t()`:

```
from tina4_python.i18n import I18n

i18n = I18n(locale="en", path="src/locales")

# Single placeholder
greeting = i18n.t("greeting", name="Alice")
# "Hello, Alice!"

# Multiple placeholders
total = i18n.t("order.total", currency="€", amount="149.99")
# "Order total: €149.99"
```

Missing placeholders are left as-is:

```
i18n.t("order.total", currency="$")
# "Order total: ${amount}"  -- {amount} not supplied
```

Extra keyword arguments are silently ignored.

---

8. Fallback Behaviour

When a key is missing from the active locale, Tina4 falls back through a chain:

- Active locale (`de`)
- Base language (`de` if locale was `de_AT`)
- Default locale (`en` — or whatever you passed as `default_locale=` when creating the `I18n` instance)
- The raw key string itself

```
TINA4_LOCALE=de
```

```
// en.json
{
  "beta_feature": "This feature is in beta."
}

// de.json
{
  "welcome": "Willkommen"
  // beta_feature not yet translated
}
```

```
i18n = I18n(locale="de", path="src/locales")
i18n.t("beta_feature")
# Falls back to en.json: "This feature is in beta."

i18n.t("completely.missing.key")
# Returns "completely.missing.key" -- the key itself
```

Your app never crashes on a missing translation. It returns the raw key string, which your translation team can search for to find gaps.

---

9. Translating Templates

In Frond templates, call `t()` directly:

```
<h1>{{ t("welcome") }}</h1>
<p>{{ t("greeting", name=user.name) }}</p>

<div class="order-summary">
  <p>{{ t("order.placed") }}</p>
  <p>{{ t("order.total", currency="$", amount=order.total) }}</p>
</div>
```

Pass the locale through the template context:

```
@get("/dashboard")
async def dashboard(request, response):
    locale = request.params.get("lang", "en")
    i18n = I18n(locale=locale, path="src/locales")

    return response.render("dashboard.html", {
        "t": i18n.t,
        "user": {"name": "Alice"},
```

```
        "locale": locale
    })
```

---

10. Exercise: Multi-Language API Responses

Build an API that returns translated error messages and UI strings based on the request locale.

Requirements

- Create locale files for `en`, `fr`, and `es` with these keys:
- `errors.not_found`
- `errors.validation`
- `user.created`
- `user.greeting`
- Create `GET /api/users/{user_id}` that:
- Detects locale from `?lang=` param (default: `en`)
- Returns a user or a translated 404 message
- Create `POST /api/users` that:
- Validates required fields and returns translated validation errors
- Returns a translated success message on creation

Test with:

```
# English (default)
curl "http://localhost:7146/api/users/999"
# {"error": "The requested resource was not found."}

# French
curl "http://localhost:7146/api/users/999?lang=fr"
# {"error": "La ressource demandée est introuvable."}

# Spanish
curl "http://localhost:7146/api/users/999?lang=es"
# {"error": "El recurso solicitado no fue encontrado."}

# Create user in French
curl -X POST "http://localhost:7146/api/users?lang=fr" \
  -H "Content-Type: application/json" \
  -d '{"email": "alice@example.com", "name": "Alice"}'
```

---

11. Solution

`src/locales/en.json`:

```
{
  "errors": {
    "not_found": "The requested resource was not found.",
```

```
    "validation": "Validation failed: {fields}"
  },
  "user": {
    "created": "User account created successfully.",
    "greeting": "Hello, {name}!"
  }
}
```

src/locales/fr.json:

```
{
  "errors": {
    "not_found": "La ressource demandée est introuvable.",
    "validation": "Échec de validation : {fields}"
  },
  "user": {
    "created": "Compte utilisateur créé avec succès.",
    "greeting": "Bonjour, {name} !"
  }
}
```

src/locales/es.json:

```
{
  "errors": {
    "not_found": "El recurso solicitado no fue encontrado.",
    "validation": "Error de validación: {fields}"
  },
  "user": {
    "created": "Cuenta de usuario creada con éxito.",
    "greeting": "¡Hola, {name}!"
  }
}
```

src/routes/users_i18n.py:

```
from tina4_python.core.router import get, post
from tina4_python.i18n import I18n

USERS = {
    1: {"id": 1, "name": "Alice", "email": "alice@example.com"},
    2: {"id": 2, "name": "Bob", "email": "bob@example.com"}
}

SUPPORTED_LOCALES = ["en", "fr", "es"]

def get_i18n(request):
    lang = request.params.get("lang", "en")
    if lang not in SUPPORTED_LOCALES:
        lang = "en"
    return I18n(locale=lang, path="src/locales")


@get("/api/users/{user_id}")
async def get_user(request, response):
    i18n = get_i18n(request)
    user_id = int(request.params["user_id"])

    user = USERS.get(user_id)
    if user is None:
```

```
        return response({"error": i18n.t("errors.not_found")}, 404)

    return response({
        "user": user,
        "greeting": i18n.t("user.greeting", name=user["name"])
    })


@post("/api/users")
async def create_user(request, response):
    i18n = get_i18n(request)
    body = request.body

    missing = [f for f in ["email", "name"] if not body.get(f)]
    if missing:
        return response({
            "error": i18n.t("errors.validation", fields=", ".join(missing))
        }, 400)

    new_user = {
        "id": max(USERS.keys()) + 1,
        "name": body["name"],
        "email": body["email"]
    }
    USERS[new_user["id"]] = new_user

    return response({
        "message": i18n.t("user.created"),
        "user": new_user
    }, 201)
```

---

12. Gotchas

1. Missing locale file raises an error

Problem: `I18n(locale="zh", ...)` crashes because `src/locales/zh.json` does not exist.

Fix: Pass `default_locale="en"` when creating the `I18n` instance. Always validate user-supplied locale values against a supported list before passing them to `I18n.set_locale()`.

2. Placeholder name mismatch

Problem: JSON has `{firstName}` but code calls `t("greeting", first_name="Alice")`. The placeholder is not replaced.

Fix: Placeholder names in JSON and keyword argument names in Python must match exactly. Use consistent naming conventions — either camelCase or snake_case throughout your locale files.

3. TINA4_LOCALE not set, t() uses "en"

Problem: `t()` returns English strings even though your app targets a different language.

Fix: Set `TINA4_LOCALE=de` (or your target locale) in `.env`. Without it, `t()` defaults to `"en"`.

4. Locale files not reloading during development

Problem: You updated a locale file but the app still serves the old translation.

Fix: The `I18n` class caches locale files in memory. Restart the dev server or set a shorter cache duration during development. In production this is the correct behaviour.

Structured Logging

1. Stop Using print()

`print("user logged in")` gets the job done in development. In production, it produces an undated, unlabelled stream of text with no severity, no context, and no way to filter. A crash at 3am gives you nothing useful.

Structured logging adds timestamps, levels, and machine-readable output to every message. Set the level in `.env`. Rotate logs by size. Write to a file. Filter by severity. Tina4's `Log` class does all of this without configuration beyond a single environment variable.

---

2. Basic Usage

```
from tina4_python.debug import Log

Log.info("Application started")
Log.debug("Request received", path="/api/users", method="GET")
Log.warning("Rate limit approaching", remaining=10)
Log.error("Database connection failed", host="db.internal", port=5432)
```

Output:

```
2026-04-02 09:14:01 [INFO]    Application started
2026-04-02 09:14:01 [DEBUG]   Request received path=/api/users method=GET
2026-04-02 09:14:01 [WARNING] Rate limit approaching remaining=10
2026-04-02 09:14:01 [ERROR]   Database connection failed host=db.internal port=5432
```

Every message includes a UTC timestamp and level. Keyword arguments become structured key-value pairs appended to the message.

---

3. Log Levels

There are five levels, in ascending severity:

Level	Method	When to Use
`ALL`	—	All levels (used only as a filter setting)
`DEBUG`	`Log.debug()`	Verbose diagnostic data; development only
`INFO`	`Log.info()`	Normal operational events
`WARNING`	`Log.warning()`	Unexpected conditions that are not errors

`ERROR`	`Log.error()`	Failures that need investigation

Setting `TINA4_LOG_LEVEL=WARNING` suppresses `DEBUG` and `INFO` messages. Only warnings and errors reach the log file.

---

4. The TINA4*LOG*LEVEL Environment Variable

```
# Development: see everything
TINA4_LOG_LEVEL=ALL

# Staging: see info and above
TINA4_LOG_LEVEL=INFO

# Production: warnings and errors only
TINA4_LOG_LEVEL=WARNING
```

The level is read at startup. Changing it requires a server restart (or, if you implement a reload endpoint, a signal to the process).

---

5. Logging in Route Handlers

```
from tina4_python.core.router import get, post
from tina4_python.debug import Log

@post("/api/orders")
async def create_order(request, response):
    body = request.body

    Log.info("Order creation started", customer_id=body.get("customer_id"))

    if not body.get("items"):
        Log.warning("Order rejected: no items", customer_id=body.get("customer_id"))
        return response({"error": "At least one item is required"}, 400)

    try:
        # Simulate order processing
        order_id = 1001
        total = sum(item.get("price", 0) for item in body["items"])

        Log.info("Order created", order_id=order_id, total=total, item_count=len(body["items"]))

        return response({"order_id": order_id, "total": total}, 201)

    except Exception as exc:
        Log.error("Order creation failed", error=str(exc), customer_id=body.get("customer_id"))
        return response({"error": "Internal server error"}, 500)
```

Log structured data alongside the message. When something goes wrong you can grep or filter by `order_id`, `customer_id`, or `error` without parsing free-form text.

---

6. File Output and Log Rotation

Write logs to a file:

```
TINA4_LOG_FILE=logs/app.log
```

Tina4 creates the file (and the `logs/` directory) if it does not exist. Logs are written to both the file and stdout.

Enable rotation by size:

```
TINA4_LOG_FILE=logs/app.log
TINA4_LOG_MAX_SIZE=10485760
TINA4_LOG_KEEP=5
```

`TINA4_LOG_MAX_SIZE=10485760` rotates the log file when it reaches 10 MB. `TINA4_LOG_KEEP=5` keeps the five most recent rotated files before deleting the oldest. These settings cap total log storage at roughly 50 MB.

Rotated files are named `app.log.1`, `app.log.2`, and so on.

---

7. Logging Exceptions

Log a full exception with stack trace using `Log.error` and Python's `traceback` module:

```
import traceback
from tina4_python.debug import Log

try:
    result = 1 / 0
except Exception as exc:
    Log.error(
        "Unhandled exception",
        error=str(exc),
        traceback=traceback.format_exc()
    )
```

Output:

```
2026-04-02 09:14:01 [ERROR] Unhandled exception error=division by zero traceback=Traceback (most
  File "src/routes/orders.py", line 34, in create_order
    result = 1 / 0
ZeroDivisionError: division by zero
```

The full stack trace is attached as a structured field. Log aggregators can extract, index, and alert on it.

---

8. Request Logging Middleware

Log every incoming request automatically:

```
from tina4_python.debug import Log

async def request_logger(request, response, next_handler):
```

```
    import time
    start = time.monotonic()
    result = await next_handler(request, response)
    elapsed = round((time.monotonic() - start) * 1000, 2)

    Log.info(
        "Request completed",
        method=request.method,
        path=request.url,
        status=getattr(result, "status_code", 200),
        duration_ms=elapsed
    )

    return result
```

Register it globally in your app:

```
from tina4_python.core.router import use
use(request_logger)
```

Every request now produces a structured log line:

```
2026-04-02 09:14:01 [INFO] Request completed method=POST path=/api/orders status=201 duration_ms
2026-04-02 09:14:02 [INFO] Request completed method=GET path=/api/users status=200 duration_ms=2
```

Filter by path, method, or status code in any log aggregation tool.

---

9. Log Levels by Environment

A typical multi-environment setup:

```
# .env.development
TINA4_LOG_LEVEL=ALL
TINA4_LOG_FILE=

# .env.staging
TINA4_LOG_LEVEL=INFO
TINA4_LOG_FILE=logs/staging.log
TINA4_LOG_MAX_SIZE=10485760
TINA4_LOG_KEEP=3

# .env.production
TINA4_LOG_LEVEL=WARNING
TINA4_LOG_FILE=logs/app.log
TINA4_LOG_MAX_SIZE=52428800
TINA4_LOG_KEEP=10
```

Development logs everything to stdout. Staging logs info and above to a rotating file. Production only logs warnings and errors to a large rotating file kept for 10 rotations.

---

10. Exercise: Add Logging to an Order API

Add structured logging to an order management API.

Requirements

- Create `POST /api/shop/orders` that:
- Logs the start of each request with `customer_id`
- Logs validation failures at `WARNING` level
- Logs successful orders at `INFO` level with `order_id`, `total`, `item_count`
- Logs any exceptions at `ERROR` level with full error context
- Create `GET /api/shop/orders/{order_id}` that:
- Logs cache hits and misses at `DEBUG` level
- Logs 404s at `WARNING` level
- Set up the request logging middleware globally

Test with:

```
# Valid order
curl -X POST http://localhost:7146/api/shop/orders \
  -H "Content-Type: application/json" \
  -d '{"customer_id": 1, "items": [{"name": "Keyboard", "price": 79.99}]}'

# Invalid order (no items)
curl -X POST http://localhost:7146/api/shop/orders \
  -H "Content-Type: application/json" \
  -d '{"customer_id": 1}'

# Get order
curl http://localhost:7146/api/shop/orders/1001
```

---

11. Solution

Create `src/routes/shop_orders.py`:

```
import traceback
from tina4_python.core.router import get, post
from tina4_python.debug import Log
from tina4_python.cache import cache_get, cache_set

ORDER_STORE = {}
NEXT_ORDER_ID = 1001


@post("/api/shop/orders")
async def create_shop_order(request, response):
    body = request.body
    customer_id = body.get("customer_id")

    Log.info("Order creation started", customer_id=customer_id)

    if not body.get("items"):
        Log.warning("Order rejected: missing items", customer_id=customer_id)
        return response({"error": "At least one item is required"}, 400)
```

```
    if not customer_id:
        Log.warning("Order rejected: missing customer_id")
        return response({"error": "customer_id is required"}, 400)

    try:
        global NEXT_ORDER_ID
        order_id = NEXT_ORDER_ID
        NEXT_ORDER_ID += 1

        items = body["items"]
        total = round(sum(item.get("price", 0) for item in items), 2)

        order = {
            "order_id": order_id,
            "customer_id": customer_id,
            "items": items,
            "total": total,
            "status": "placed"
        }
        ORDER_STORE[order_id] = order
        cache_set(f"order:{order_id}", order, ttl=300)

        Log.info(
            "Order created",
            order_id=order_id,
            customer_id=customer_id,
            total=total,
            item_count=len(items)
        )

        return response(order, 201)

    except Exception as exc:
        Log.error(
            "Order creation failed",
            customer_id=customer_id,
            error=str(exc),
            traceback=traceback.format_exc()
        )
        return response({"error": "Internal server error"}, 500)


@get("/api/shop/orders/{order_id}")
async def get_shop_order(request, response):
    order_id = int(request.params["order_id"])

    cached = cache_get(f"order:{order_id}")
    if cached:
        Log.debug("Order cache hit", order_id=order_id)
        return response(cached)

    Log.debug("Order cache miss", order_id=order_id)

    order = ORDER_STORE.get(order_id)
    if order is None:
        Log.warning("Order not found", order_id=order_id)
        return response({"error": "Order not found"}, 404)

    cache_set(f"order:{order_id}", order, ttl=300)
```

```
    return response(order)
```

---

12. Gotchas

1. DEBUG logs flooding production

Problem: Hundreds of debug messages per second fill the log file and slow the server.

Fix: Set `TINA4_LOG_LEVEL=WARNING` in production. Debug calls are no-ops when the level is above `DEBUG`.

2. Logging sensitive data

Problem: `Log.info("User logged in", password=request.body["password"])` writes passwords to the log file.

Fix: Never log passwords, tokens, card numbers, or PII. Log identifiers and metadata only: `Log.info("User logged in", user_id=user["id"])`.

3. Log file grows without rotation

Problem: `TINA4_LOG_FILE` is set but `TINA4_LOG_MAX_SIZE` is not. The log file grows until the disk is full.

Fix: Always set `TINA4_LOG_MAX_SIZE` and `TINA4_LOG_KEEP` alongside `TINA4_LOG_FILE` in production.

4. Forgetting to log exceptions

Problem: A route returns 500 but there is no log entry explaining why.

Fix: Wrap the handler body in `try/except`. Log the exception with `Log.error(..., error=str(exc), traceback=traceback.format_exc())` before returning the 500 response.

Email with Messenger

1. Every App Sends Email

Signup confirmations. Password resets. Weekly digests. Invoices with PDF attachments. Every application needs email. Nobody enjoys building it.

SMTP configuration. Plain text fallbacks. Attachment encoding. Connection timeouts. The details pile up fast. Tina4's `Messenger` class absorbs all of it. Configure through `.env`. Create an instance. Send. In development mode, Messenger intercepts every outgoing email and displays it in the dev dashboard -- no real SMTP server required.

---

2. Messenger Configuration via .env

All email configuration lives in `.env`:

```
TINA4_MAIL_HOST=smtp.example.com
TINA4_MAIL_PORT=587
TINA4_MAIL_USERNAME=your-email@example.com
TINA4_MAIL_PASSWORD=your-app-password
TINA4_MAIL_ENCRYPTION=tls
TINA4_MAIL_FROM=noreply@example.com
TINA4_MAIL_FROM_NAME=My Store
```

Variable	Description	Common Values
`TINA4_MAIL_HOST`	SMTP server hostname	`smtp.gmail.com`, `smtp.mailgun.org`, `smtp.sendgrid.net`
`TINA4_MAIL_PORT`	SMTP port	`587` (TLS), `465` (SSL), `25` (unencrypted)
`TINA4_MAIL_USERNAME`	Login username	Usually your email address
`TINA4_MAIL_PASSWORD`	Login password or app-specific password	App passwords for Gmail
`TINA4_MAIL_ENCRYPTION`	Encryption method	`tls` (recommended), `ssl`, `none`
`TINA4_MAIL_FROM`	Default "From" address	`noreply@yourdomain.com`
`TINA4_MAIL_FROM_NAME`	Default "From" display name	`My Store`, `Acme Corp`

Messenger also accepts legacy `SMTP_*` prefixed variables as fallback. The `TINA4_MAIL_*` prefix takes priority.

Common Provider Configurations

Gmail:

```
TINA4_MAIL_HOST=smtp.gmail.com
TINA4_MAIL_PORT=587
TINA4_MAIL_USERNAME=your-email@gmail.com
TINA4_MAIL_PASSWORD=your-app-password
TINA4_MAIL_ENCRYPTION=tls
```

Gmail requires an "App Password" (not your regular password) when two-factor authentication is enabled.

Mailgun:

```
TINA4_MAIL_HOST=smtp.mailgun.org
TINA4_MAIL_PORT=587
TINA4_MAIL_USERNAME=postmaster@mg.yourdomain.com
TINA4_MAIL_PASSWORD=your-mailgun-smtp-password
TINA4_MAIL_ENCRYPTION=tls
```

SendGrid:

```
TINA4_MAIL_HOST=smtp.sendgrid.net
TINA4_MAIL_PORT=587
TINA4_MAIL_USERNAME=apikey
TINA4_MAIL_PASSWORD=your-sendgrid-api-key
TINA4_MAIL_ENCRYPTION=tls
```

---

3. Constructor Override Pattern

Different emails need different SMTP accounts. Transactional emails from one server. Marketing from another. Override the configuration in the constructor:

```
from tina4_python.messenger import Messenger

# Uses .env defaults
mailer = Messenger()

# Override specific settings
marketing_mailer = Messenger(
    host="smtp.mailgun.org",
    port=587,
    username="marketing@mg.yourdomain.com",
    password="marketing-smtp-password",
    encryption="tls",
    from_address="newsletter@yourdomain.com",
    from_name="My Store Newsletter"
)
```

Constructor arguments take priority over `.env` values. Any argument you omit falls back to the environment variable.

---

4. Sending Plain Text Email

The simplest email:

```
from tina4_python.core.router import post
from tina4_python.messenger import Messenger

@post("/api/contact")
async def contact_form(request, response):
    body = request.body

    mailer = Messenger()

    result = mailer.send(
        to=body["email"],
        subject="Contact Form Submission",
        body=f"Name: {body['name']}\n"
             f"Email: {body['email']}\n"
             f"Message:\n{body['message']}"
    )

    if result["success"]:
        return response({"message": "Email sent successfully"})

    return response({"error": "Failed to send email", "details": result["error"]}, 500)
```

```
curl -X POST http://localhost:7146/api/contact \
  -H "Content-Type: application/json" \
  -d '{"name": "Alice", "email": "alice@example.com", "message": "I love your products!"}'
```

```
{"message":"Email sent successfully"}
```

The `send()` method returns a dict with three keys: `"success"` (boolean), `"error"` (string or None), and `"message_id"` (string or None).

The send() Method Signature

```
mailer.send(
    to,                 # Recipient(s) -- string or list of strings
    subject,            # Email subject line
    body,               # Email body (plain text or HTML)
    html=False,         # If True, body is treated as HTML
    text=None,          # Plain text alternative (when body is HTML)
    cc=None,            # CC recipient(s) -- string or list
    bcc=None,           # BCC recipient(s) -- string or list
    reply_to=None,      # Reply-To address
    attachments=None,   # List of file paths or dicts
    headers=None        # Additional email headers (dict)
)
```

---

5. Sending HTML Email with Text Fallback

Most emails should carry HTML with a plain text fallback. Email clients that cannot render HTML display the text version instead:

```
from tina4_python.messenger import Messenger
```

```
mailer = Messenger()

html_body = """
<html>
<body style="font-family: Arial, sans-serif; max-width: 600px; margin: 0 auto;">
    <div style="background: #1a1a2e; color: white; padding: 20px; text-align: center;">
        <h1 style="margin: 0;">Welcome to My Store!</h1>
    </div>
    <div style="padding: 20px;">
        <p>Hi Alice,</p>
        <p>Thank you for creating your account. We are excited to have you!</p>
        <p>Here is what you can do next:</p>
        <ul>
            <li>Browse our <a href="https://mystore.com/products">product catalog</a></li>
            <li>Set up your <a href="https://mystore.com/profile">profile</a></li>
            <li>Check out our <a href="https://mystore.com/deals">current deals</a></li>
        </ul>
        <p>If you have any questions, reply to this email and we will get back to you within 24
        <p>Cheers,<br>The My Store Team</p>
    </div>
    <div style="background: #f5f5f5; padding: 12px; text-align: center; font-size: 12px; color:
        <p>You received this because you signed up at mystore.com</p>
    </div>
</body>
</html>
"""

text_body = (
    "Hi Alice,\n\n"
    "Thank you for creating your account. We are excited to have you!\n\n"
    "Here is what you can do next:\n"
    "- Browse our product catalog: https://mystore.com/products\n"
    "- Set up your profile: https://mystore.com/profile\n"
    "- Check out our current deals: https://mystore.com/deals\n\n"
    "If you have any questions, reply to this email.\n\n"
    "Cheers,\nThe My Store Team"
)

result = mailer.send(
    to="alice@example.com",
    subject="Welcome to My Store!",
    body=html_body,
    html=True,
    text=text_body
)
```

Pass `html=True` to tell Messenger the body contains HTML. The `text` parameter provides the plain text alternative. Messenger builds a `multipart/alternative` message that carries both versions.

---

6. Adding Attachments

Attach files by providing their paths:

```
from tina4_python.messenger import Messenger
```

```
mailer = Messenger()

result = mailer.send(
    to="accounting@example.com",
    subject="Monthly Invoice #1042",
    body="<h2>Invoice #1042</h2><p>Please find the invoice attached.</p>",
    html=True,
    attachments=[
        "/path/to/invoices/invoice-1042.pdf",
        "/path/to/reports/monthly-summary.csv"
    ]
)
```

Messenger reads each file, determines its MIME type, and encodes it for email transmission. Each entry can be a file path string, a `Path` object, or a dict for more control.

Attachments with Custom Names and Binary Content

```
result = mailer.send(
    to="alice@example.com",
    subject="Your Export",
    body="<p>Here is your data export.</p>",
    html=True,
    attachments=[
        {
            "filename": "my-store-export.csv",
            "content": csv_bytes,
            "mime": "text/csv"
        }
    ]
)
```

The dict format accepts `filename` (display name), `content` (raw bytes), and `mime` (MIME type string). The recipient sees the file named `my-store-export.csv` regardless of how it was generated.

---

7. CC, BCC, and Reply-To

```
from tina4_python.messenger import Messenger

mailer = Messenger()

result = mailer.send(
    to="alice@example.com",
    subject="Team Meeting Notes",
    body="<p>Here are the notes from today's meeting.</p>",
    html=True,
    cc=["bob@example.com", "charlie@example.com"],
    bcc=["manager@example.com"],
    reply_to="alice@example.com"
)
```

- **cc**: List of email addresses to carbon copy. All recipients see CC addresses.
- **bcc**: List of email addresses to blind carbon copy. Recipients cannot see BCC addresses.

• **reply_to**: When the recipient clicks "Reply", this address fills the "To" field instead of the "From" address.

Both `cc` and `bcc` accept a single string or a list of strings.

---

8. Custom Headers

Messenger supports two methods for adding headers. The `add_header()` method sets a default header on all emails sent by that instance. The `headers` parameter on `send()` sets headers for a single email.

Default Headers on an Instance

```
from tina4_python.messenger import Messenger

mailer = Messenger()
mailer.add_header("X-App-Name", "My Store")
mailer.add_header("X-Environment", "production")

# Every email from this instance carries both headers
mailer.send(to="alice@example.com", subject="Test", body="Hello")
```

Per-Email Headers

```
result = mailer.send(
    to="customer@example.com",
    subject="Your Support Ticket #123",
    body="We are looking into your issue.",
    reply_to="support@mystore.com",
    headers={
        "X-Ticket-Id": "123",
        "X-Priority": "1",
        "X-Mailer": "Tina4 Messenger"
    }
)
```

Per-email headers merge with default headers. If both define the same key, the per-email value wins.

Custom headers serve several purposes. Tracking headers like `X-Ticket-Id` let you correlate emails with support tickets. Priority headers influence some email clients' display. Bulk-sending headers like `Precedence: bulk` help mail servers classify newsletters.

---

9. Reading Inbox via IMAP

Messenger reads email through IMAP. Configure the IMAP server in `.env`:

```
TINA4_MAIL_IMAP_HOST=imap.example.com
TINA4_MAIL_IMAP_PORT=993
```

Messenger reuses `TINA4_MAIL_USERNAME` and `TINA4_MAIL_PASSWORD` for IMAP authentication. You can also override the IMAP host and port in the constructor:

```
mailer = Messenger(imap_host="imap.gmail.com", imap_port=993)
```

Listing Inbox Messages

The `inbox()` method fetches message headers from the mailbox:

```
from tina4_python.core.router import get
from tina4_python.messenger import Messenger

@get("/api/inbox")
async def get_inbox(request, response):
    mailer = Messenger()

    emails = mailer.inbox(limit=20, offset=0)

    messages = []
    for email in emails:
        messages.append({
            "uid": email["uid"],
            "from": email["from"],
            "subject": email["subject"],
            "date": email["date"],
            "snippet": email["snippet"],
            "seen": email["seen"]
        })

    return response({"messages": messages, "count": len(messages)})
```

```
curl http://localhost:7146/api/inbox
```

```
{
  "messages": [
    {
      "uid": "12345",
      "from": "customer@example.com",
      "subject": "Order question",
      "date": "2026-03-22T10:30:00+00:00",
      "snippet": "Hi, I have a question about my recent order...",
      "seen": false
    }
  ],
  "count": 1
}
```

The `inbox()` method returns messages newest-first. Each message contains `uid`, `subject`, `from`, `to`, `date`, `snippet` (first 150 characters of the body), and `seen` (boolean).

Reading a Specific Message

```
@get("/api/inbox/{uid}")
async def get_email(request, response):
    mailer = Messenger()
    uid = request.params["uid"]

    email = mailer.read(uid, mark_read=True)

    if not email:
        return response({"error": "Email not found"}, 404)

    return response({
        "uid": email["uid"],
        "from": email["from"],
        "to": email["to"],
        "cc": email["cc"],
        "subject": email["subject"],
        "date": email["date"],
        "body_html": email["body_html"],
        "body_text": email["body_text"],
        "attachments": [
            {"filename": a["filename"], "size": a["size"], "content_type": a["content_type"]}
            for a in email.get("attachments", [])
        ]
    })
```

The `read()` method fetches the full message including body and attachments. Pass `mark_read=False` to leave the message unread.

Searching Messages

```
@get("/api/inbox/search")
async def search_inbox(request, response):
    mailer = Messenger()

    results = mailer.search(
        subject=request.query.get("q"),
        sender=request.query.get("from"),
        unseen_only=request.query.get("unread") == "true",
        limit=20
    )

    return response({"messages": results, "count": len(results)})
```

The `search()` method accepts `subject`, `sender`, `since` (date string "DD-Mon-YYYY"), `before`, and `unseen_only` as filters. All filters combine with AND logic.

Other IMAP Operations

```
mailer = Messenger()

# Count unread messages
count = mailer.unread()

# Mark a message as read
mailer.mark_read("12345")

# Mark a message as unread
mailer.mark_unread("12345")

# Delete a message
mailer.delete("12345")

# List all mailbox folders
folders = mailer.folders()
# ["INBOX", "Sent", "Drafts", "Trash", "Spam"]
```

Testing IMAP Connectivity

```
mailer = Messenger()
result = mailer.test_imap_connection()
if result["success"]:
    print("IMAP connection works")
else:
    print(f"IMAP failed: {result['error']}")
```

---

10. Dev Mode: Email Interception

When `TINA4_DEBUG=true`, Tina4 intercepts all outgoing emails and stores them locally. No real recipients receive anything. No accidental emails during development.

Navigate to `/__dev` and look for the "Mail" section. You see:

- Every email "sent" during the current session
- The To, CC, and BCC addresses
- The subject and body (both HTML and plain text)
- Attachments (viewable inline)
- The timestamp

This is invaluable for testing email functionality without configuring a real SMTP server. Use `create_messenger()` instead of `Messenger()` to get automatic dev-mode interception:

```
from tina4_python.messenger import create_messenger

mailer = create_messenger()
# In dev mode: captures locally
# In production: sends via SMTP
```

Disabling Interception

If you need to test real email delivery during development, override the interception:

```
TINA4_MAILBOX_DIR=false
```

With this set, emails reach real recipients even when `TINA4_DEBUG=true`. Use with caution -- you do not want to accidentally email your entire user base from a dev machine.

---

11. Using Templates for Email Content

Hardcoded HTML in Python strings is ugly and hard to maintain. Templates fix this.

Create `src/templates/emails/welcome.html`:

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8">
</head>
<body style="font-family: Arial, sans-serif; max-width: 600px; margin: 0 auto; background: #f5f5
    <div style="background: #1a1a2e; color: white; padding: 24px; text-align: center; border-rad
        <h1 style="margin: 0; font-size: 24px;">Welcome, {{ name }}!</h1>
    </div>
    <div style="background: white; padding: 24px; border-radius: 0 0 8px 8px;">
        <p>Hi {{ name }},</p>
        <p>Your account has been created successfully. Here are your details:</p>

        <table style="width: 100%; border-collapse: collapse; margin: 16px 0;">
            <tr>
                <td style="padding: 8px; border-bottom: 1px solid #eee; font-weight: bold;">Emai
                <td style="padding: 8px; border-bottom: 1px solid #eee;">{{ email }}</td>
            </tr>
            <tr>
                <td style="padding: 8px; border-bottom: 1px solid #eee; font-weight: bold;">Acco
                <td style="padding: 8px; border-bottom: 1px solid #eee;">#{{ user_id }}</td>
            </tr>
            <tr>
                <td style="padding: 8px; border-bottom: 1px solid #eee; font-weight: bold;">Sign
                <td style="padding: 8px; border-bottom: 1px solid #eee;">{{ signed_up_at }}</td>
            </tr>
        </table>

        <p>Get started by exploring:</p>
        <ul>
            <li><a href="{{ base_url }}/products" style="color: #1a1a2e;">Our product catalog</a
            <li><a href="{{ base_url }}/profile" style="color: #1a1a2e;">Your profile settings</
        </ul>

        {% if promo_code %}
            <div style="background: #d4edda; padding: 16px; border-radius: 4px; margin: 16px 0;"
                <strong>Special offer!</strong> Use code <code>{{ promo_code }}</code> for 10% o
            </div>
        {% endif %}

        <p>Cheers,<br>The {{ app_name }} Team</p>
    </div>

    <div style="text-align: center; padding: 12px; color: #888; font-size: 12px;">
        <p>You received this because you signed up at {{ app_name }}.</p>
```

```
        <p><a href="{{ base_url }}/unsubscribe?token={{ unsubscribe_token }}" style="color: #888
    </div>
</body>
</html>
```

Rendering and Sending

```
import os
import secrets
from datetime import datetime
from tina4_python.core.router import post, template
from tina4_python.messenger import Messenger

@post("/api/register")
async def register_user(request, response):
    body = request.body

    # Create user (database logic)
    user_id = 42

    # Render the email template
    email_data = {
        "name": body["name"],
        "email": body["email"],
        "user_id": user_id,
        "signed_up_at": datetime.now().strftime("%B %d, %Y"),
        "base_url": os.getenv("APP_URL", "http://localhost:7146"),
        "app_name": "My Store",
        "promo_code": "WELCOME10",
        "unsubscribe_token": secrets.token_hex(16)
    }

    html_body = template("emails/welcome.html", **email_data)

    # Send the email
    mailer = Messenger()
    result = mailer.send(
        to=body["email"],
        subject=f"Welcome to My Store, {body['name']}!",
        body=html_body,
        html=True,
        text=f"Hi {body['name']},\n\nWelcome to My Store! "
             f"Your account (#{user_id}) has been created.\n\n"
             f"Cheers,\nThe My Store Team"
    )

    return response({
        "message": "Registration successful",
        "email_sent": result["success"],
        "user_id": user_id
    }, 201)
```

```
curl -X POST http://localhost:7146/api/register \
  -H "Content-Type: application/json" \
  -d '{"name": "Alice", "email": "alice@example.com", "password": "securePass123"}'
```

```
{
  "message": "Registration successful",
  "email_sent": true,
  "user_id": 42
```

```
}
```

With `TINA4_DEBUG=true`, the email appears in the dev dashboard instead of reaching a real inbox. Inspect the rendered HTML, check that template variables substituted, and verify the layout.

---

12. Sending Email via Queues

In production, never send email inside a route handler. The SMTP call blocks the response. Use the queue system from Chapter 12:

```
from tina4_python.core.router import post, template
from tina4_python.queue import Queue
from tina4_python.messenger import Messenger

# In the route handler, just queue the email
@post("/api/register")
async def register_user(request, response):
    body = request.body
    user_id = 42  # Simulated

    queue = Queue(topic="emails")
    queue.push({
        "template": "emails/welcome.html",
        "to": body["email"],
        "subject": f"Welcome to My Store, {body['name']}!",
        "data": {
            "name": body["name"],
            "email": body["email"],
            "user_id": user_id,
            "signed_up_at": "March 22, 2026",
            "base_url": "http://localhost:7146",
            "app_name": "My Store",
            "promo_code": "WELCOME10"
        }
    })

    return response({"message": "Registration successful", "user_id": user_id}, 201)


# The consumer sends the actual email
@Queue.consume("emails")
async def send_email_job(job):
    payload = job.payload

    html_body = template(payload["template"], **payload["data"])

    mailer = Messenger()
    result = mailer.send(
        to=payload["to"],
        subject=payload["subject"],
        body=html_body,
        html=True
    )

    if not result["success"]:
```

```
            print(f"Email failed: {result['error']}")
            return False  # Retry

        print(f"Email sent to {payload['to']}")
        return True
```

The route handler returns in under 50 milliseconds. The queue worker sends the email on its own timeline. If the SMTP server is down, retries happen automatically.

---

13. Exercise: Build a Contact Form with Email Notification

Build a contact form that sends an email notification when submitted.

Requirements

- Create a `GET /contact` page that renders a contact form with fields: name, email, subject, and message
- Create a `POST /contact` endpoint that:
- Validates all fields are present
- Sends an email notification to the site admin (`admin@example.com`)
- The email should include all form fields, formatted in HTML
- Shows a flash message on success
- Redirects back to the contact page
- Create an email template at `src/templates/emails/contact-notification.html` that formats the contact submission

Test with:

```
# View the form
curl http://localhost:7146/contact

# Submit the form
curl -X POST http://localhost:7146/contact \
  -H "Content-Type: application/json" \
  -d '{"name": "Bob", "email": "bob@example.com", "subject": "Product inquiry", "message": "Do y

# Check the dev dashboard for the intercepted email
# Navigate to http://localhost:7146/__dev
```

---

14. Solution

Create `src/templates/emails/contact-notification.html`:

```
<!DOCTYPE html>
<html>
<head><meta charset="UTF-8"></head>
<body style="font-family: Arial, sans-serif; max-width: 600px; margin: 0 auto;">
    <div style="background: #1a1a2e; color: white; padding: 16px; text-align: center;">
```

```
        <h2 style="margin: 0;">New Contact Form Submission</h2>
    </div>
    <div style="padding: 20px; background: white;">
        <table style="width: 100%; border-collapse: collapse;">
            <tr>
                <td style="padding: 10px; border-bottom: 1px solid #eee; font-weight: bold; widt
                <td style="padding: 10px; border-bottom: 1px solid #eee;">{{ name }} ({{ email }
            </tr>
            <tr>
                <td style="padding: 10px; border-bottom: 1px solid #eee; font-weight: bold;">Sub
                <td style="padding: 10px; border-bottom: 1px solid #eee;">{{ subject }}</td>
            </tr>
            <tr>
                <td style="padding: 10px; border-bottom: 1px solid #eee; font-weight: bold;">Dat
                <td style="padding: 10px; border-bottom: 1px solid #eee;">{{ submitted_at }}</td
            </tr>
        </table>

        <div style="margin-top: 16px; padding: 16px; background: #f8f9fa; border-radius: 4px;">
            <h3 style="margin-top: 0;">Message</h3>
            <p style="white-space: pre-wrap;">{{ message }}</p>
        </div>

        <p style="margin-top: 16px; color: #888; font-size: 12px;">
            Reply directly to this email to respond to {{ name }} at {{ email }}.
        </p>
    </div>
</body>
</html>
```

Create `src/templates/contact.html`:

```
{% extends "base.html" %}

{% block title %}Contact Us{% endblock %}

{% block content %}
    <h1>Contact Us</h1>

    {% if flash %}
        <div style="padding: 12px; border-radius: 4px; margin-bottom: 16px;
            {% if flash.type == 'success' %}background: #d4edda; color: #155724;{% endif %}
            {% if flash.type == 'error' %}background: #f8d7da; color: #721c24;{% endif %}">
            {{ flash.message }}
        </div>
    {% endif %}

    <form method="POST" action="/contact" style="max-width: 500px;">
        <div style="margin-bottom: 12px;">
            <label for="name" style="display: block; margin-bottom: 4px; font-weight: bold;">Nam
            <input type="text" name="name" id="name" required
                   style="width: 100%; padding: 8px; border: 1px solid #ddd; border-radius: 4px;
        </div>

        <div style="margin-bottom: 12px;">
            <label for="email" style="display: block; margin-bottom: 4px; font-weight: bold;">Em
            <input type="email" name="email" id="email" required
                   style="width: 100%; padding: 8px; border: 1px solid #ddd; border-radius: 4px;
        </div>
```

```
        <div style="margin-bottom: 12px;">
            <label for="subject" style="display: block; margin-bottom: 4px; font-weight: bold;">
            <input type="text" name="subject" id="subject" required
                   style="width: 100%; padding: 8px; border: 1px solid #ddd; border-radius: 4px;
        </div>

        <div style="margin-bottom: 12px;">
            <label for="message" style="display: block; margin-bottom: 4px; font-weight: bold;">
            <textarea name="message" id="message" rows="6" required
                      style="width: 100%; padding: 8px; border: 1px solid #ddd; border-radius: 4
        </div>

        <button type="submit"
                style="padding: 10px 20px; background: #1a1a2e; color: white; border: none; bord
            Send Message
        </button>
    </form>
{% endblock %}
```

Create `src/routes/contact.py`:

```
import os
from datetime import datetime
from tina4_python.core.router import get, post, template
from tina4_python.messenger import Messenger

@get("/contact")
async def contact_page(request, response):
    flash = request.session.get("_flash")
    if flash:
        del request.session["_flash"]

    return response(template("contact.html", flash=flash))


@post("/contact")
async def contact_submit(request, response):
    body = request.body

    # Validate
    errors = []
    if not body.get("name"):
        errors.append("Name is required")
    if not body.get("email"):
        errors.append("Email is required")
    if not body.get("subject"):
        errors.append("Subject is required")
    if not body.get("message"):
        errors.append("Message is required")

    if errors:
        request.session["_flash"] = {
            "type": "error",
            "message": "Please fill in all fields: " + ", ".join(errors)
        }
        return response.redirect("/contact")

    # Render the email template
    html_body = template("emails/contact-notification.html",
```

```
        name=body["name"],
        email=body["email"],
        subject=body["subject"],
        message=body["message"],
        submitted_at=datetime.now().strftime("%B %d, %Y at %I:%M %p")
    )

    # Send the email
    mailer = Messenger()
    admin_email = os.getenv("ADMIN_EMAIL", "admin@example.com")

    result = mailer.send(
        to=admin_email,
        subject=f"Contact Form: {body['subject']}",
        body=html_body,
        html=True,
        reply_to=body["email"],
        text=f"Contact form submission from {body['name']} ({body['email']}):\n\n"
             f"Subject: {body['subject']}\n\n"
             f"Message:\n{body['message']}"
    )

    if result["success"]:
        request.session["_flash"] = {
            "type": "success",
            "message": "Thank you for your message! We will get back to you shortly."
        }
    else:
        request.session["_flash"] = {
            "type": "error",
            "message": "Sorry, there was a problem sending your message. Please try again later.
        }

    return response.redirect("/contact")
```

Testing:

- Open `http://localhost:7146/contact` in your browser
- Fill in the form and submit
- You should see a green "Thank you" flash message
- Open `http://localhost:7146/__dev` to see the intercepted email
- The email shows the sender details, subject, message, and formatted HTML

API test:

```
curl -X POST http://localhost:7146/contact \
  -H "Content-Type: application/json" \
  -d '{"name": "Bob", "email": "bob@example.com", "subject": "Product inquiry", "message": "Do y
  -c cookies.txt -b cookies.txt
```

The response is a `302` redirect to `/contact`. Follow the redirect to see the flash message:

```
curl http://localhost:7146/contact -b cookies.txt
```

The HTML response includes the success flash message.

---

15. Gotchas

1. Gmail Blocks "Less Secure" Apps

Problem: Sending via Gmail fails with "Authentication failed" or "Username and Password not accepted".

Cause: Gmail blocks SMTP access from apps that do not use OAuth2 by default. Your regular password will not work when two-factor authentication is enabled.

Fix: Generate an "App Password" in your Google Account settings (Security > 2-Step Verification > App Passwords). Use this 16-character password as `TINA4_MAIL_PASSWORD`. This is separate from your regular Google password.

2. Emails Go to Spam

Problem: Emails land in the recipient's spam folder.

Cause: Your sending domain lacks proper DNS records (SPF, DKIM, DMARC) or you send from a free email provider (Gmail, Yahoo).

Fix: Use a dedicated sending domain with proper DNS records. Set up SPF, DKIM, and DMARC records. Use a transactional email service -- Mailgun, SendGrid, or Amazon SES -- that handles email reputation for you.

3. HTML Email Looks Broken

Problem: The email looks fine in Gmail but broken in Outlook or Apple Mail.

Cause: Email clients have different HTML/CSS support. CSS flexbox, grid, and many modern properties do not work in email.

Fix: Use inline styles (not external stylesheets or `<style>` blocks). Use table-based layouts for complex designs. Test with an email preview tool. Keep it simple -- most transactional emails do not need elaborate designs.

4. Attachment File Not Found

Problem: `Messenger.send()` returns an error about a missing file.

Cause: The attachment path is relative or incorrect. The file does not exist at the specified location.

Fix: Use absolute paths for attachments. Verify the file exists before calling `send()`: `if not os.path.exists(path): ...`. If the file is generated dynamically (a PDF, for example), make sure the generation completes before sending.

5. Dev Mode Silently Intercepts Emails

Problem: You configured SMTP but no emails arrive. No errors either.

Cause: `TINA4_DEBUG=true` intercepts all emails and stores them in the dev dashboard. The email never reaches the SMTP server.

Fix: Check the dev dashboard at `/__dev` for intercepted emails. If you want to send real emails during development, set `TINA4_MAILBOX_DIR=false`. Remove this setting before committing.

6. Email Template Variables Not Substituted

Problem: The email body shows `{{ name }}` literally instead of the user's name.

Cause: You passed the raw template file content instead of rendering it through the template engine.

Fix: Use `template("emails/template.html", **data)` to render the template with variables substituted. Do not read the file with `open()` -- that gives you the raw template source.

7. Connection Timeout on Send

Problem: `Messenger.send()` hangs for 30 seconds and then fails with a timeout error.

Cause: The SMTP server is unreachable from your network. The port may be blocked by a firewall, or the hostname may be wrong.

Fix: Test SMTP connectivity with `mailer.test_connection()`. Verify the hostname, port, and encryption settings. Check that your firewall allows outbound connections on the SMTP port. If you sit behind a corporate firewall, port 587 or 465 might be blocked -- ask your network administrator.

8. IMAP Host Not Configured

Problem: Calling `mailer.inbox()` raises `MessengerError: IMAP host not configured`.

Cause: You did not set `TINA4_MAIL_IMAP_HOST` in `.env` or pass `imap_host` to the constructor.

Fix: Add `TINA4_MAIL_IMAP_HOST=imap.example.com` and `TINA4_MAIL_IMAP_PORT=993` to `.env`. The IMAP host is separate from the SMTP host -- many providers use different hostnames for sending and reading.

Frontend with tina4css

1. The Problem with Frontend Toolchains

Your client wants a dashboard. You know the drill. Install Node.js. Run `npm install`. Wait for 200MB of `node_modules`. Configure webpack or Vite. Set up PostCSS. Add a CSS framework. Pray nothing breaks when you upgrade a dependency six months from now.

Tina4 skips all of that. The framework ships with **tina4css** -- a Bootstrap-compatible CSS framework -- and **frond.js** -- a lightweight JavaScript helper. Both arrive when you scaffold a project. No npm. No webpack. No build step. Link the files. Start building.

This chapter ends with a complete admin dashboard. Sidebar. Navigation. Cards. Tables. Modals. Dark mode. Progress bars. AJAX-driven user management. Zero npm dependencies.

---

2. What Ships with Tina4

Run `tina4 init python`. Several files appear in your project:

```
src/public/
■■■ css/
■   ■■■ tina4.css        # The CSS framework
■■■ js/
■   ■■■ tina4.min.js     # Core AJAX utilities
■   ■■■ frond.min.js     # Template engine client-side helpers
■   ■■■ tina4js.min.js   # Reactive frontend framework (tina4-js)
■■■ scss/
    ■■■ tina4.scss       # SCSS source (optional, for customization)
```

Include them in any template:

```
<link rel="stylesheet" href="/css/tina4.css">
<script src="/js/tina4.min.js"></script>
<script src="/js/frond.min.js"></script>
```

Include only what you need. See section 7 for the full JavaScript API reference. No CDN. No package manager. No version conflicts.

---

3. The Grid System

tina4css uses a 12-column responsive grid. The class names match Bootstrap. Know Bootstrap? You know tina4css.

```
<div class="container">
    <div class="row">
        <div class="col-md-4">
            <p>One third</p>
        </div>
        <div class="col-md-4">
            <p>One third</p>
        </div>
```

```
        <div class="col-md-4">
            <p>One third</p>
        </div>
    </div>
</div>
```

Breakpoints:

Class Prefix	Screen Width	Typical Device
`col-`	All sizes	Phones and up
`col-sm-`	>= 576px	Large phones
`col-md-`	>= 768px	Tablets
`col-lg-`	>= 992px	Desktops
`col-xl-`	>= 1200px	Large desktops

Columns stack vertically on screens smaller than their breakpoint. A `col-md-6` element takes half the row on tablets and up, full width on phones.

---

4. Components

Navbar

```
<nav class="navbar navbar-dark bg-dark">
    <div class="container">
        <a class="navbar-brand" href="/">My Dashboard</a>
        <ul class="navbar-nav">
            <li class="nav-item"><a class="nav-link" href="/dashboard">Dashboard</a></li>
            <li class="nav-item"><a class="nav-link" href="/products">Products</a></li>
            <li class="nav-item"><a class="nav-link" href="/settings">Settings</a></li>
        </ul>
    </div>
</nav>
```

Use `navbar-light bg-light` for a light theme, or `navbar-dark bg-primary` for a colored background.

Cards

```
<div class="card">
    <div class="card-header">Monthly Revenue</div>
    <div class="card-body">
        <h2 class="card-title">$12,450</h2>
        <p class="card-text">Up 12% from last month</p>
    </div>
    <div class="card-footer text-muted">Updated 5 minutes ago</div>
</div>
```

Buttons

```
<button class="btn btn-primary">Save</button>
<button class="btn btn-secondary">Cancel</button>
<button class="btn btn-danger">Delete</button>
<button class="btn btn-success">Publish</button>
<button class="btn btn-outline-primary">Outlined</button>
<button class="btn btn-sm btn-primary">Small</button>
<button class="btn btn-lg btn-primary">Large</button>
```

Tables

```
<table class="table table-striped table-hover">
    <thead>
        <tr>
            <th>Name</th>
            <th>Category</th>
            <th>Price</th>
            <th>Status</th>
        </tr>
    </thead>
    <tbody>
        <tr>
            <td>Wireless Keyboard</td>
            <td>Electronics</td>
            <td>$79.99</td>
            <td><span class="badge bg-success">In Stock</span></td>
        </tr>
        <tr>
            <td>Standing Desk</td>
            <td>Furniture</td>
            <td>$549.99</td>
            <td><span class="badge bg-danger">Out of Stock</span></td>
        </tr>
    </tbody>
</table>
```

Table variants: `table-bordered`, `table-striped`, `table-hover`, `table-sm` (compact), `table-responsive` (wraps in a scrollable container on small screens). Mix and match.

Alerts

```
<div class="alert alert-success">Product created.</div>
<div class="alert alert-danger">Failed to save changes.</div>
<div class="alert alert-warning alert-dismissible">
    Your trial expires in 3 days.
    <button type="button" class="close" data-dismiss="alert">&times;</button>
</div>
<div class="alert alert-info">A new version is available.</div>
```

Modals

```
<button class="btn btn-primary" data-toggle="modal" data-target="#confirmModal">Delete Product</

<div class="modal" id="confirmModal">
    <div class="modal-dialog">
        <div class="modal-content">
            <div class="modal-header">
                <h5 class="modal-title">Confirm Delete</h5>
                <button class="btn-close" data-dismiss="modal"></button>
            </div>
            <div class="modal-body">
                <p>Are you sure you want to delete this product? This action cannot be undone.</
            </div>
            <div class="modal-footer">
                <button class="btn btn-secondary" data-dismiss="modal">Cancel</button>
                <button class="btn btn-danger">Delete</button>
            </div>
        </div>
    </div>
</div>
```

tina4css includes the JavaScript for modal toggling. No jQuery required.

Progress Bars

Progress bars visualize completion, upload status, or system health. The outer `progress` container holds a `progress-bar` fill element.

```
<div class="progress">
    <div class="progress-bar bg-success" style="width: 75%">75%</div>
</div>

<div class="progress">
    <div class="progress-bar bg-warning progress-bar-striped" style="width: 45%">
        45% - Uploading...
    </div>
</div>
```

Set the width with inline `style`. Add `bg-success`, `bg-info`, `bg-warning`, or `bg-danger` for color. Add `progress-bar-striped` for animated stripes.

Badges

```
<span class="badge bg-primary">Primary</span>
<span class="badge bg-success">Active</span>
<span class="badge bg-warning">Pending</span>
<span class="badge bg-danger">Overdue</span>
<span class="badge bg-info">Info</span>
<span class="badge bg-dark">Dark</span>
```

Forms

```
<form>
    <div class="form-group">
        <label for="name" class="form-label">Product Name</label>
        <input type="text" class="form-control" id="name" placeholder="Enter product name">
    </div>

    <div class="form-group">
        <label for="category" class="form-label">Category</label>
        <select class="form-control" id="category">
            <option value="">Select a category</option>
            <option value="electronics">Electronics</option>
            <option value="furniture">Furniture</option>
        </select>
    </div>

    <div class="form-group">
        <label for="description" class="form-label">Description</label>
        <textarea class="form-control" id="description" rows="4"></textarea>
    </div>

    <div class="form-check">
        <input type="checkbox" class="form-check-input" id="featured">
        <label class="form-check-label" for="featured">Featured product</label>
    </div>

    <button type="submit" class="btn btn-primary mt-3">Save Product</button>
</form>
```

---

5. SCSS Customization

The default tina4css works out of the box. To customize colors, fonts, or spacing, edit the SCSS source.

Edit `src/public/scss/tina4.scss`:

```
// Override variables before importing the framework
$primary: #2d6a4f;
$secondary: #52b788;
$dark: #1b4332;
$font-family-base: 'Inter', sans-serif;
$border-radius: 8px;

// Import the framework
@import 'tina4-base';
```

Compile SCSS to CSS:

```
tina4 scss
```

```
Compiling SCSS...
  src/public/scss/tina4.scss -> src/public/css/tina4.css
Done (0.12s)
```

The compiled CSS replaces the default `tina4.css`. Your custom colors and fonts take effect across the entire application.

Live SCSS Compilation

During development, run SCSS compilation in watch mode:

```
tina4 scss --watch
```

Every save to a `.scss` file triggers a recompile. Combined with Tina4's live reload, changes appear in the browser within a second.

---

6. frond.js -- The JavaScript Helper

`frond.js` is a lightweight JavaScript library that ships with Tina4. It handles AJAX requests, form submission, JWT token management, and loading indicators. No jQuery. No Axios. No other dependency.

AJAX Requests

```
// GET request
frond.get("/api/products", function (data) {
    console.log("Products:", data);
});

// GET with error handling
frond.get("/api/products", function (data) {
    console.log(data);
}, function (error) {
    console.error("Failed:", error);
});

// POST request
frond.post("/api/products", {
    name: "New Product",
    price: 29.99
}, function (data) {
    console.log("Created:", data);
});

// PUT request
frond.put("/api/products/1", {
    name: "Updated Product"
}, function (data) {
    console.log("Updated:", data);
});

// DELETE request
frond.delete("/api/products/1", function (data) {
    console.log("Deleted:", data);
});
```

Form Submission via AJAX

```
<form id="product-form" data-frond-submit="/api/products" data-frond-method="POST">
    <input type="text" name="name" placeholder="Product name">
    <input type="number" name="price" placeholder="Price">
    <button type="submit">Create</button>
</form>

<script src="/js/frond.min.js"></script>
<script>
    frond.onFormSuccess("product-form", function (data) {
        alert("Product created: " + data.name);
    });

    frond.onFormError("product-form", function (error) {
        alert("Error: " + error.message);
    });
</script>
```

The `data-frond-submit` attribute tells frond.js to intercept the form submission and send it as an AJAX request. No page reload. frond.js serializes all form fields as JSON.

Token Management

frond.js manages JWT tokens. Store a token after login, and frond.js attaches it to every request.

```
// Store the token (usually after login)
frond.setToken("eyJhbGciOiJIUzI1NiIs...");

// All subsequent requests include:
// Authorization: Bearer eyJhbGciOiJIUzI1NiIs...
frond.get("/api/profile", function (data) {
    console.log("Profile:", data);
});

// Clear the token (logout)
frond.clearToken();
```

frond.js stores the token in `localStorage` and includes it as a `Bearer` token in the `Authorization` header on every request.

Loading Indicators

frond.js can show and hide a loading element during AJAX requests. Pass a CSS selector in the options object.

```
// Show a loading state while fetching
frond.get("/api/products", function (data) {
    renderProducts(data.products);
}, null, {
    loading: "#loadingSpinner"  // CSS selector for loading element
});
```

The element with id `loadingSpinner` appears while the request flies and disappears when it completes. Pair it with a spinner or "Loading..." text in your HTML:

```
<div id="loadingSpinner" class="text-center p-4" style="display: none;">
    Loading...
</div>
```

frond.js toggles `display: block` and `display: none` on the element. No extra CSS needed.

WebSocket (Covered in Chapter 23)

```
const ws = frond.ws("/ws/chat/general");
ws.on("message", function (data) {
    console.log("Message:", JSON.parse(data));
});
```

Connections drop. frond.js reconnects with exponential backoff. If the server restarts or the network blips, the client reconnects without intervention.

---

7. JavaScript API Reference

Tina4 ships three JavaScript files. Each serves a different purpose. Use them independently or together.

Including the Scripts

```
<script src="/js/tina4.min.js"></script>
<script src="/js/frond.min.js"></script>
<script src="/js/tina4js.min.js"></script>
```

All three live in `src/public/js/` and are served from `/js/`. Include only what you need.

7.1 tina4.min.js -- Core Utilities

Low-level helpers for AJAX page loading and form submission. Use this when you want simple dynamic page updates without a full framework.

`loadPage(url, targetId)`

Fetches HTML from `url` and injects it into the element with the given `id`.

```
<nav>
    <a href="#" onclick="loadPage('/dashboard', 'content')">Dashboard</a>
    <a href="#" onclick="loadPage('/settings', 'content')">Settings</a>
</nav>
<div id="content"><!-- pages load here --></div>

<script src="/js/tina4.min.js"></script>
```

`saveForm(formId, url, method)`

Serializes a form and submits it via AJAX. Prevents the default page reload.

```
<form id="product-form">
    <input type="text" name="name" placeholder="Product name">
    <input type="number" name="price" placeholder="Price">
    <button type="button" onclick="saveForm('product-form', '/api/products', 'POST')">
        Save
    </button>
</form>
```

`sendRequest(url, method, data, callback)`

Generic AJAX helper for any HTTP method. Returns the response to a callback function.

```
sendRequest("/api/products", "GET", null, function (response) {
    console.log("Products:", JSON.parse(response));
});

sendRequest("/api/products", "POST", { name: "Widget", price: 9.99 }, function (response) {
    console.log("Created:", JSON.parse(response));
});
```

7.2 frond.min.js -- Template Engine Client-Side Helpers

A companion to the Frond template engine. Handles AJAX form interception, WebSocket connections with auto-reconnect, JWT token refresh, and dynamic template loading.

AJAX Form Handling

Forms with `data-frond-submit` are intercepted. No page reload. No boilerplate.

```
<form id="login-form" data-frond-submit="/api/login" data-frond-method="POST">
    <input type="text" name="username" placeholder="Username">
    <input type="password" name="password" placeholder="Password">
    <button type="submit">Log In</button>
</form>

<script src="/js/frond.min.js"></script>
<script>
    frond.onFormSuccess("login-form", function (data) {
        frond.setToken(data.token);
        window.location.href = "/dashboard";
    });
</script>
```

WebSocket Auto-Reconnect

```
const ws = frond.ws("/ws/notifications");
ws.on("message", function (data) {
    const notification = JSON.parse(data);
    alert(notification.text);
});
// If the server restarts or the network blips, frond.js reconnects.
```

Token Refresh

JWT tokens stored via `frond.setToken()` are attached to every request. When a token expires, frond.js triggers a refresh before retrying the request.

```
frond.setToken("eyJhbGciOiJIUzI1NiIs...");
// All subsequent frond.get/post/put/delete calls include the token.
// When the token expires, frond.js calls the refresh endpoint.
```

Dynamic Template Loading

Load server-rendered Frond templates into any element without a full page reload.

```
frond.loadTemplate("/templates/user-card", { userId: 42 }, "user-panel");
```

```
// Fetches the rendered template and injects it into #user-panel.
```

7.3 tina4js.min.js -- Reactive Frontend Framework

A standalone reactive framework for building interactive client-side applications. Provides signals, computed values, effects, Web Components, client-side routing, and built-in fetch and WebSocket wrappers. This is the **tina4-js** project.

Reactive State: `signal()`, `computed()`, `effect()`

```
import { signal, computed, effect } from "/js/tina4js.min.js";

const count = signal(0);
const doubled = computed(() => count.value * 2);

effect(() => {
    console.log(`Count: ${count.value}, Doubled: ${doubled.value}`);
});

count.value = 5; // logs "Count: 5, Doubled: 10"
```

DOM Rendering: `html` Tagged Template

```
import { signal, html } from "/js/tina4js.min.js";

const name = signal("World");

const app = html`
    <div>
        <h1>Hello, ${name}!</h1>
        <input value="${name}" oninput="${(e) => name.value = e.target.value}" />
    </div>
`;

document.getElementById("app").append(app);
```

Web Components: `Tina4Element`

```
import { Tina4Element, signal, html } from "/js/tina4js.min.js";

class CounterButton extends Tina4Element {
    setup() {
        this.count = signal(0);
    }
    render() {
        return html`
            <button onclick="${() => this.count.value++}">
                Clicked ${this.count} times
            </button>
        `;
    }
}
customElements.define("counter-button", CounterButton);
```

Use it in HTML:

```
<counter-button></counter-button>
<script type="module" src="/js/counter-button.js"></script>
```

Fetch Wrapper: `api()`

```
import { api } from "/js/tina4js.min.js";

const products = await api("/api/products");            // GET
await api("/api/products", { method: "POST", body: { name: "Widget" } });
```

WebSocket Client: `ws()`

```
import { ws } from "/js/tina4js.min.js";

const socket = ws("/ws/chat");
socket.on("message", (data) => console.log(data));
socket.send({ text: "Hello" });
```

Client-Side Routing: `route()`, `navigate()`

```
import { route, navigate, html } from "/js/tina4js.min.js";

route("/", () => html`<h1>Home</h1>`);
route("/about", () => html`<h1>About</h1>`);

// Navigate programmatically
navigate("/about");
```

---

8. Building an Admin Dashboard

A complete admin dashboard. The kind of page that powers the backend of every web application.

Base Template

Create `src/templates/base.html`:

```
<!DOCTYPE html>
<html lang="en" data-theme="light">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>{% block title %}Dashboard{% endblock %}</title>
    <link rel="stylesheet" href="/css/tina4.css">
    <script>
        var t = localStorage.getItem("theme");
        if (t) document.documentElement.setAttribute("data-theme", t);
    </script>
</head>
<body>
    <nav class="navbar navbar-dark bg-dark">
        <div class="container-fluid">
            <a class="navbar-brand" href="/admin">TaskFlow Admin</a>
            <ul class="navbar-nav">
                <li class="nav-item"><a class="nav-link" href="/admin">Dashboard</a></li>
                <li class="nav-item"><a class="nav-link" href="/admin/products">Products</a></li
                <li class="nav-item"><a class="nav-link" href="/admin/users">Users</a></li>
                <li class="nav-item">
                    <button class="btn btn-sm btn-outline-light" onclick="toggleDarkMode()">Dark
                </li>
```

```
            </ul>
        </div>
    </nav>

    <div class="container-fluid mt-4">
        <div class="row">
            <div class="col-md-2">
                {% block sidebar %}
                <div class="list-group">
                    <a href="/admin" class="list-group-item list-group-item-action">Overview</a>
                    <a href="/admin/products" class="list-group-item list-group-item-action">Pro
                    <a href="/admin/orders" class="list-group-item list-group-item-action">Order
                    <a href="/admin/customers" class="list-group-item list-group-item-action">Cu
                    <a href="/admin/reports" class="list-group-item list-group-item-action">Repo
                    <a href="/admin/settings" class="list-group-item list-group-item-action">Set
                </div>
                {% endblock %}
            </div>
            <div class="col-md-10">
                {% block content %}{% endblock %}
            </div>
        </div>
    </div>

    <script src="/js/frond.min.js"></script>
    <script>
        function toggleDarkMode() {
            var html = document.documentElement;
            var current = html.getAttribute("data-theme");
            html.setAttribute("data-theme", current === "dark" ? "light" : "dark");
            localStorage.setItem("theme", current === "dark" ? "light" : "dark");
        }
    </script>
    {% block scripts %}{% endblock %}
</body>
</html>
```

Dashboard Page

Create `src/templates/dashboard.html`:

```
{% extends "base.html" %}

{% block title %}Dashboard - Admin{% endblock %}

{% block content %}
    <h1>Dashboard</h1>

    <div class="row mb-4">
        <div class="col-md-3">
            <div class="card">
                <div class="card-body">
                    <h6 class="card-subtitle text-muted">Total Products</h6>
                    <h2 class="card-title">{{ stats.total_products }}</h2>
                </div>
            </div>
        </div>
        <div class="col-md-3">
            <div class="card">
```

```
            <div class="card-body">
                <h6 class="card-subtitle text-muted">Total Orders</h6>
                <h2 class="card-title">{{ stats.total_orders }}</h2>
            </div>
        </div>
    </div>
    <div class="col-md-3">
        <div class="card">
            <div class="card-body">
                <h6 class="card-subtitle text-muted">Revenue</h6>
                <h2 class="card-title">${{ stats.revenue }}</h2>
            </div>
        </div>
    </div>
    <div class="col-md-3">
        <div class="card">
            <div class="card-body">
                <h6 class="card-subtitle text-muted">Active Users</h6>
                <h2 class="card-title">{{ stats.active_users }}</h2>
            </div>
        </div>
    </div>
</div>

<div class="row">
    <div class="col-md-8">
        <div class="card">
            <div class="card-header">Recent Orders</div>
            <div class="card-body">
                <table class="table table-striped">
                    <thead>
                        <tr>
                            <th>Order</th>
                            <th>Customer</th>
                            <th>Amount</th>
                            <th>Status</th>
                        </tr>
                    </thead>
                    <tbody>
                        {% for order in recent_orders %}
                        <tr>
                            <td>#{{ order.id }}</td>
                            <td>{{ order.customer }}</td>
                            <td>${{ order.amount }}</td>
                            <td>
                                <span class="badge bg-{{ order.badge }}">{{ order.status }}<
                            </td>
                        </tr>
                        {% endfor %}
                    </tbody>
                </table>
            </div>
        </div>
    </div>
    <div class="col-md-4">
        <div class="card">
            <div class="card-header">Quick Actions</div>
            <div class="card-body">
                <a href="/admin/products/new" class="btn btn-primary btn-block mb-2">Add Pro
```

```
                    <a href="/admin/orders" class="btn btn-outline-primary btn-block mb-2">View
                    <a href="/admin/reports" class="btn btn-outline-secondary btn-block">Generat
                </div>
            </div>

            <div class="card mt-3">
                <div class="card-header">System Health</div>
                <div class="card-body">
                    <p><strong>CPU:</strong></p>
                    <div class="progress mb-3">
                        <div class="progress-bar bg-success" style="width: {{ stats.cpu_usage }}
                            {{ stats.cpu_usage }}%
                        </div>
                    </div>
                    <p><strong>Memory:</strong></p>
                    <div class="progress mb-3">
                        <div class="progress-bar bg-info" style="width: {{ stats.memory_usage }}
                            {{ stats.memory_usage }}%
                        </div>
                    </div>
                    <p><strong>Disk:</strong></p>
                    <div class="progress">
                        <div class="progress-bar bg-warning" style="width: {{ stats.disk_usage }
                            {{ stats.disk_usage }}%
                        </div>
                    </div>
                </div>
            </div>
        </div>
    </div>
{% endblock %}
```

Dashboard Route

Create `src/routes/admin.py`:

```
from tina4_python.core.router import get, template

@get("/admin")
async def admin_dashboard(request, response):
    stats = {
        "total_products": 156,
        "total_orders": 1243,
        "revenue": "24,580",
        "active_users": 89,
        "cpu_usage": 42,
        "memory_usage": 68,
        "disk_usage": 55
    }

    recent_orders = [
        {"id": 1042, "customer": "Alice Johnson", "amount": "129.99", "status": "Shipped", "badg
        {"id": 1041, "customer": "Bob Smith", "amount": "549.99", "status": "Processing", "badge
        {"id": 1040, "customer": "Carol White", "amount": "79.99", "status": "Delivered", "badge
        {"id": 1039, "customer": "Dave Brown", "amount": "34.99", "status": "Cancelled", "badge"
    ]

    return response(template("dashboard.html", stats=stats, recent_orders=recent_orders))
```

Start the server and visit `http://localhost:7146/admin`. You see a sidebar, stat cards, a data table, quick action buttons, and system health progress bars. Zero npm dependencies.

---

9. Dark Mode

tina4css supports dark mode through a single attribute on the `<html>` element:

```
<!-- Light mode (default) -->
<html data-theme="light">

<!-- Dark mode -->
<html data-theme="dark">
```

The toggle function from the base template handles switching:

```
function toggleDarkMode() {
    var html = document.documentElement;
    var current = html.getAttribute("data-theme");
    html.setAttribute("data-theme", current === "dark" ? "light" : "dark");
    localStorage.setItem("theme", current === "dark" ? "light" : "dark");
}
```

Dark mode transforms every surface. Backgrounds darken. Text colors invert. Borders shift. Cards, tables, buttons -- all adjust contrast. Zero CSS rules required from you.

Respecting System Preference

Match the user's operating system dark mode setting:

```
var saved = localStorage.getItem("theme");
if (saved) {
    document.documentElement.setAttribute("data-theme", saved);
} else if (window.matchMedia("(prefers-color-scheme: dark)").matches) {
    document.documentElement.setAttribute("data-theme", "dark");
}
```

---

10. Responsive Design

tina4css is mobile-first. Components stack on small screens and expand on larger ones.

Responsive Sidebar

On mobile, the sidebar collapses into a toggle:

```
<button class="btn btn-dark d-md-none" onclick="toggleSidebar()">Menu</button>

<div id="sidebar" class="d-none d-md-block col-md-2">
    <!-- sidebar content -->
</div>

<script>
function toggleSidebar() {
    var sidebar = document.getElementById("sidebar");
    sidebar.classList.toggle("d-none");
```

```
}
</script>
```

Responsive Tables

Tables scroll horizontally on small screens:

```
<div class="table-responsive">
    <table class="table">
        <!-- table content -->
    </table>
</div>
```

Hiding Elements by Screen Size

```
<div class="d-none d-md-block">Only visible on tablet and up</div>
<div class="d-md-none">Only visible on mobile</div>
```

---

11. Building a Users Page with AJAX

A user management page. Data loads via AJAX using frond.js. No full page reloads. The table populates from the API. Users create, edit, and delete records through modals and inline actions.

The Template

Create `src/templates/admin/users.html`:

```
{% extends "base.html" %}

{% block title %}Users - Admin{% endblock %}

{% block content %}
    <div class="card">
        <div class="card-header d-flex justify-content-between align-items-center">
            <span>All Users</span>
            <button class="btn btn-sm btn-primary" data-toggle="modal" data-target="#addUserModa
                Add User
            </button>
        </div>
        <div class="card-body p-0">
            <div id="loadingSpinner" class="text-center p-4" style="display: none;">
                Loading...
            </div>
            <table class="table table-hover m-0" id="usersTable">
                <thead>
                    <tr>
                        <th>ID</th>
                        <th>Name</th>
                        <th>Email</th>
                        <th>Created</th>
                        <th>Actions</th>
                    </tr>
                </thead>
                <tbody id="usersBody">
                    <!-- Populated by JavaScript -->
                </tbody>
            </table>
```

```
        </div>
    </div>

    <!-- Add User Modal -->
    <div class="modal" id="addUserModal">
        <div class="modal-dialog">
            <div class="modal-content">
                <div class="modal-header">
                    <h5 class="modal-title">Add New User</h5>
                    <button type="button" class="btn-close" data-dismiss="modal"></button>
                </div>
                <div class="modal-body">
                    <form id="addUserForm">
                        <div class="form-group">
                            <label for="userName">Name</label>
                            <input type="text" class="form-control" name="name" id="userName" re
                        </div>
                        <div class="form-group">
                            <label for="userEmail">Email</label>
                            <input type="email" class="form-control" name="email" id="userEmail"
                        </div>
                    </form>
                </div>
                <div class="modal-footer">
                    <button class="btn btn-secondary" data-dismiss="modal">Cancel</button>
                    <button class="btn btn-primary" onclick="createUser()">Create User</button>
                </div>
            </div>
        </div>
    </div>

    <div id="alertArea" class="mt-3"></div>
{% endblock %}

{% block scripts %}
<script>
    function loadUsers() {
        frond.get("/api/users", function (data) {
            var tbody = document.getElementById("usersBody");
            tbody.innerHTML = "";

            if (data.data && data.data.length > 0) {
                data.data.forEach(function (user) {
                    var row = '<tr>'
                        + '<td>' + user.id + '</td>'
                        + '<td>' + user.name + '</td>'
                        + '<td>' + user.email + '</td>'
                        + '<td>' + user.created_at + '</td>'
                        + '<td>'
                        + '<button class="btn btn-sm btn-outline-primary" onclick="editUser(' +
                        + '<button class="btn btn-sm btn-outline-danger" onclick="deleteUser(' +
                        + '</td>'
                        + '</tr>';
                    tbody.innerHTML += row;
                });
            } else {
                tbody.innerHTML = '<tr><td colspan="5" class="text-center p-4">No users found</t
            }
        }, null, { loading: "#loadingSpinner" });
```

```
    }

    function createUser() {
        var name = document.getElementById("userName").value;
        var email = document.getElementById("userEmail").value;

        frond.post("/api/users", { name: name, email: email }, function (data) {
            document.getElementById("alertArea").innerHTML =
                '<div class="alert alert-success">User "' + data.name + '" created.</div>';
            loadUsers();
        }, function (error) {
            document.getElementById("alertArea").innerHTML =
                '<div class="alert alert-danger">Error creating user.</div>';
        });
    }

    function deleteUser(id) {
        if (confirm("Are you sure you want to delete this user?")) {
            frond.delete("/api/users/" + id, function () {
                loadUsers();
            });
        }
    }

    // Load users on page load
    loadUsers();
</script>
{% endblock %}
```

The Route

```
from tina4_python.core.router import get, template

@get("/admin/users")
async def admin_users(request, response):
    return response(template("admin/users.html"))
```

This page loads users via an AJAX call to `/api/users` (provided by auto-CRUD on the User model from Chapter 6). The loading indicator appears while the request flies. The table populates when data arrives. Users add records through a modal form and delete with confirmation -- all without full page reloads.

---

12. HTML Builder -- Generating HTML in Code

Sometimes you need HTML from a route handler without a template -- a dynamic email body, an AJAX fragment, or a one-off snippet. String concatenation is fragile. The `HTMLElement` class builds HTML with auto-escaping.

Direct Construction

```
from tina4_python.HtmlElement import HTMLElement

el = HTMLElement("div", {"class": "card"}, ["Hello"])
str(el)  # <div class="card">Hello</div>
```

The constructor takes three arguments: tag name, an attribute dictionary, and a list of children (strings or other `HTMLElement` instances).

Builder Pattern

Call an element to append children and return a new element:

```
page = HTMLElement("div")(
    HTMLElement("h1")("Dashboard"),
    HTMLElement("p")("Welcome back."),
)
```

Pass a dictionary to merge attributes:

```
link = HTMLElement("a")({"href": "/home", "class": "nav-link"}, "Home")
# <a href="/home" class="nav-link">Home</a>
```

Helper Functions

Typing `HTMLElement` everywhere gets verbose. `add_html_helpers()` injects shorthand functions -- `_div()`, `_p()`, `_a()`, `_span()`, `_h1()`, `_table()`, and every other standard HTML tag -- into your module:

```
from tina4_python.HtmlElement import add_html_helpers

add_html_helpers(globals())

html = _div({"class": "alert alert-success"},
    _strong("Done!"),
    _p("Your changes have been saved."),
)
```

The first argument can be an attribute dictionary. Everything else becomes children.

Void Tags and Auto-Escaping

Void tags render without a closing tag:

```
HTMLElement("img", {"src": "logo.png", "alt": "Logo"})
# <img src="logo.png" alt="Logo">

HTMLElement("br")
# <br>
```

All attribute values and text children are HTML-escaped. User input is safe by default -- no XSS vectors from forgotten escaping.

Use Case -- HTML from a Route

```
from tina4_python.core.router import get
from tina4_python.HtmlElement import add_html_helpers

add_html_helpers(globals())

@get("/api/status-badge")
async def status_badge(request, response):
    status = request.params.get("status", "unknown")
    color = "success" if status == "active" else "danger"

    badge = _span({"class": f"badge bg-{color}"}, status)
    return response(str(badge))
```

No template file needed. The builder handles escaping, nesting, and rendering in one place.

---

13. Exercise: Build an Admin Dashboard

Build an admin dashboard for a product management system.

Requirements

- Create a base template with:
- A dark navbar with the app name and navigation links
- A sidebar with menu items (Dashboard, Products, Orders, Settings)
- A main content area
- Dark mode toggle that persists across page loads
- Create a dashboard page at `GET /admin` with:
- Four stat cards (Products, Orders, Revenue, Users)
- A table showing recent orders with status badges
- Progress bars showing system health (CPU, Memory, Disk)
- Create a users page at `GET /admin/users` with:
- A table of users loaded via AJAX using frond.js
- An "Add User" button that opens a modal with a form
- AJAX form submission that refreshes the table without a page reload
- A loading indicator while data fetches
- Use tina4css classes throughout (no custom CSS needed)

Test by:

- Visit `http://localhost:7146/admin` -- see the dashboard with stats, orders, and progress bars
- Click "Dark Mode" -- the entire page switches to dark theme

- Refresh the page -- dark mode persists
- Resize the browser to mobile width -- the sidebar collapses
- Visit `http://localhost:7146/admin/users` -- see the user table loaded via AJAX

---

14. Solution

The base template, dashboard page, and users page are shown in sections 8, 11, and above. The product list page follows the same AJAX pattern from section 11, but for products instead of users.

For the users page, use auto-CRUD on the User model (`auto_crud = True`) so the API endpoints exist at `/api/users`. Load the table with `frond.get("/api/users", ...)` and handle form submission with `frond.post("/api/users", ...)`.

Add the product list route:

```
@get("/admin/products")
async def admin_products(request, response):
    search = request.params.get("search", "")
    selected_category = request.params.get("category", "")

    products = [
        {"id": 1, "name": "Wireless Keyboard", "category": "Electronics", "price": "79.99", "in_
        {"id": 2, "name": "Standing Desk", "category": "Furniture", "price": "549.99", "in_stock
        {"id": 3, "name": "Coffee Grinder", "category": "Kitchen", "price": "49.99", "in_stock":
        {"id": 4, "name": "Yoga Mat", "category": "Fitness", "price": "29.99", "in_stock": True}
        {"id": 5, "name": "USB-C Hub", "category": "Electronics", "price": "49.99", "in_stock":
    ]

    if selected_category:
        products = [p for p in products if p["category"] == selected_category]

    if search:
        products = [p for p in products if search.lower() in p["name"].lower()]

    categories = ["Electronics", "Furniture", "Kitchen", "Fitness"]

    return response(template("products.html",
        products=products,
        categories=categories,
        search=search,
        selected_category=selected_category
    ))
```

---

15. Gotchas

1. CSS Not Loading

Problem: The page looks unstyled -- no colors, no layout, plain text.

Cause: The path to `tina4.css` is wrong. The file lives in `src/public/css/` but the link points elsewhere.

Fix: Tina4 serves everything in `src/public/` from the root path. Link to `/css/tina4.css` (not `/src/public/css/tina4.css`). Verify the file exists at `src/public/css/tina4.css`.

2. frond.js Functions Not Found

Problem: `frond.get is not a function` or `frond is not defined` in the browser console.

Cause: The `frond.min.js` script tag is missing or placed after the code that uses it.

Fix: Include `<script src="/js/frond.min.js"></script>` before any script that calls `frond.*`. Put it at the bottom of the body, before your custom scripts.

3. Dark Mode Flickers on Page Load

Problem: The page loads in light mode and then flashes to dark mode.

Cause: The dark mode JavaScript runs after the page renders. The browser paints the light theme first, then switches.

Fix: Add the theme detection script in the `<head>` (before the body renders):

```
<head>
    <script>
        var t = localStorage.getItem("theme");
        if (t) document.documentElement.setAttribute("data-theme", t);
    </script>
</head>
```

4. SCSS Not Compiling

Problem: You edited `tina4.scss` but the CSS did not change.

Cause: SCSS does not compile on its own. You need to run `tina4 scss` or use `--watch` mode.

Fix: Run `tina4 scss --watch` during development. For production builds, run `tina4 scss` as part of your build process.

5. Modal Does Not Open

Problem: Clicking the button does nothing -- the modal stays hidden.

Cause: The `data-toggle` and `data-target` attributes require frond.js to be loaded. Without it, no JavaScript handles the modal toggle.

Fix: Ensure `frond.min.js` is loaded. Verify the `data-target` matches the modal's `id` exactly (including the `#` prefix).

6. Grid Columns Do Not Stack on Mobile

Problem: Columns stay side-by-side on phone screens instead of stacking vertically.

Cause: You used `col-4` instead of `col-md-4`. The `col-4` class applies at all screen sizes.

Fix: Use responsive prefixes: `col-md-4` means "one-third on medium screens and up, full width on small screens."

7. Static Files Return 404

Problem: CSS, JS, or image files return 404 Not Found.

Cause: The files are not in the `src/public/` directory.

Fix: Static files must be in `src/public/`. The URL path maps to the file path within that directory. `/css/tina4.css` maps to `src/public/css/tina4.css`.

8. Form Data Not Reaching the Server

Problem: `frond.post()` sends the request but `request.body` is empty on the server.

Cause: frond.js sends JSON by default. Passing a `FormData` object or building the data object wrong prevents correct parsing.

Fix: Pass a plain JavaScript object to `frond.post()`. Do not use `new FormData()` -- frond.js handles serialization. Make sure your object keys match what the server expects.

9. Loading Indicator Never Disappears

Problem: The loading spinner shows but never hides after the AJAX request completes.

Cause: The CSS selector passed to the `loading` option does not match any element, or the element uses a CSS class to toggle visibility instead of inline `display`.

Fix: frond.js toggles `display: block` and `display: none`. Make sure the element exists and its initial style is `display: none`. Use an `id` selector: `{ loading: "#loadingSpinner" }`.

Testing

1. Why Tests Matter More Than You Think

Friday afternoon. Your client reports a critical bug in production. You fix it -- one line of code. But did that fix break something else? 47 routes. 12 ORM models. 3 middleware functions. Clicking through every page: an hour. Running the test suite: 2 seconds.

```
tina4 test

Running tests...

  ProductTest
    [PASS] test_create_product
    [PASS] test_load_product
    [PASS] test_update_product
    [PASS] test_delete_product
    [PASS] test_list_products_with_filter

  AuthTest
    [PASS] test_login_with_valid_credentials
    [PASS] test_login_with_invalid_password
    [PASS] test_protected_route_without_token
    [PASS] test_protected_route_with_valid_token

  9 tests, 9 passed, 0 failed (0.34s)
```

Everything still works. You deploy with confidence. Weekend intact.

Tina4 includes an inline testing framework. No external packages. No configuration. Write a test. Run it. Done.

---

2. Your First Test

Tests live in the `tests/` directory. Every `.py` file in that directory is auto-discovered when you run `tina4 test`.

Create `tests/test_basic.py`:

```
from tina4_python.test import Test, assert_equal, assert_true

class BasicTest(Test):

    def test_addition(self):
        assert_equal(2 + 2, 4, "Basic addition should work")

    def test_string_contains(self):
        greeting = "Hello, World!"
        assert_true("World" in greeting, "Greeting should contain 'World'")

    def test_array_length(self):
        items = [1, 2, 3, 4, 5]
        assert_equal(len(items), 5, "List should have 5 items")
```

Run it:

```
tina4 test

Running tests...

  BasicTest
    [PASS] test_addition
    [PASS] test_string_contains
    [PASS] test_array_length

  3 tests, 3 passed, 0 failed (0.02s)
```

How It Works

- Your test class extends `Test`.
- Every method that starts with `test` is a test case. The method name is converted to a readable label: `test_addition` stays as `test_addition`.
- Inside each test, you call assertion functions to verify behavior.
- If all assertions pass, the test passes. If any assertion fails, the test fails and you see the failure message.

---

3. Assertion Methods

Tina4's testing framework provides these assertion functions:

assert_equal(actual, expected, message)

Checks that two values are equal.

```
assert_equal(4, 4, "Should be equal")              # PASS
assert_equal("hello", "hello", "Strings match")    # PASS
assert_equal(4, 5, "Not equal")                     # FAIL
```

assert_true(value, message)

Checks that a value is truthy.

```
assert_true(True, "Should be true")                # PASS
assert_true(1, "1 is truthy")                      # PASS
assert_true("yes", "Non-empty string is truthy")   # PASS
assert_true(False, "This fails")                   # FAIL
assert_true(0, "Zero is falsy")                    # FAIL
```

assert_false(value, message)

Checks that a value is falsy.

```
assert_false(False, "Should be false")             # PASS
assert_false(0, "Zero is falsy")                   # PASS
assert_false("", "Empty string is falsy")          # PASS
assert_false(True, "This fails")                   # FAIL
```

assert*raises(callable, exception*class, message)

Checks that a function raises a specific exception.

```
assert_raises(
    lambda: int("not-a-number"),
    ValueError,
    "Should raise ValueError"
)

assert_raises(
    lambda: 10 / 0,
    ZeroDivisionError,
    "Should raise on division by zero"
)
```

assert*not*equal(actual, expected, message)

Checks that two values are not equal.

```
assert_not_equal("hello", "world", "Strings differ")  # PASS
assert_not_equal(4, 4, "Same values")                  # FAIL
```

assert_none(value, message)

Checks that a value is None.

```
assert_none(None, "Should be None")     # PASS
assert_none("hello", "Not None")        # FAIL
```

assert*not*none(value, message)

Checks that a value is not None.

```
assert_not_none("hello", "Has value")   # PASS
assert_not_none(None, "Is None")        # FAIL
```

---

4. Testing ORM Models

Let us test a Product model. The test creates records, loads them, updates them, and deletes them.

Create `tests/test_product.py`:

```
from tina4_python.test import Test, assert_equal, assert_true, assert_not_none

class ProductTest(Test):

    def test_create_product(self):
        product = Product()
        product.name = "Test Widget"
        product.category = "Testing"
        product.price = 19.99
        product.in_stock = True
        product.save()

        assert_not_none(product.id, "Product should have an ID after save")
        assert_true(product.id > 0, "Product ID should be positive")
```

```
def test_load_product(self):
    product = Product()
    product.name = "Load Test Widget"
    product.category = "Testing"
    product.price = 29.99
    product.save()

    loaded = Product.find(product.id)

    assert_equal(loaded.name, "Load Test Widget", "Name should match")
    assert_equal(loaded.category, "Testing", "Category should match")
    assert_equal(loaded.price, 29.99, "Price should match")

def test_update_product(self):
    product = Product()
    product.name = "Update Test Widget"
    product.price = 10.00
    product.save()

    product_id = product.id

    product.name = "Updated Widget"
    product.price = 15.00
    product.save()

    reloaded = Product.find(product_id)

    assert_equal(reloaded.name, "Updated Widget", "Name should be updated")
    assert_equal(reloaded.price, 15.00, "Price should be updated")

def test_delete_product(self):
    product = Product()
    product.name = "Delete Me"
    product.price = 5.00
    product.save()

    product_id = product.id
    product.delete()

    gone = Product.find(product_id)

    assert_true(gone is None, "Deleted product should not be loadable")

def test_select_with_filter(self):
    p1 = Product()
    p1.name = "Filter Test A"
    p1.category = "FilterCat"
    p1.price = 10.00
    p1.save()

    p2 = Product()
    p2.name = "Filter Test B"
    p2.category = "FilterCat"
    p2.price = 20.00
    p2.save()

    products, count = Product.where("category = ?", ["FilterCat"])

    assert_true(len(products) >= 2, "Should find at least 2 FilterCat products")
```

```
        names = [p.name for p in products]
        assert_true("Filter Test A" in names, "Should include Filter Test A")
        assert_true("Filter Test B" in names, "Should include Filter Test B")
```

Run it:

```
tina4 test

Running tests...

  ProductTest
    [PASS] test_create_product
    [PASS] test_load_product
    [PASS] test_update_product
    [PASS] test_delete_product
    [PASS] test_select_with_filter

  5 tests, 5 passed, 0 failed (0.18s)
```

Test Database

By default, `tina4 test` uses a separate test database so your development data is not affected. The test database is created at `data/test.db` (SQLite) and is reset before each test run. If you want to use a different database for tests, set it in `.env`:

```
TINA4_DATABASE_URL=sqlite:///data/test.db
```

---

5. Testing Routes

Tina4 provides a test client for making HTTP requests to your routes without starting a server.

Create `tests/test_routes.py`:

```
from tina4_python.test import Test, assert_equal, assert_true, assert_not_none
import json

class RouteTest(Test):

    def test_health_endpoint(self):
        resp = self.get("/health")

        assert_equal(resp.status_code, 200, "Health check should return 200")

        body = json.loads(resp.body)
        assert_equal(body["status"], "ok", "Status should be 'ok'")
        assert_not_none(body.get("version"), "Should include version")

    def test_get_products(self):
        resp = self.get("/api/products")

        assert_equal(resp.status_code, 200, "Should return 200")

        body = json.loads(resp.body)
        assert_true("data" in body or "products" in body, "Should contain product data")

    def test_create_product(self):
        resp = self.post("/api/products", {
```

```
            "name": "Route Test Product",
            "category": "Testing",
            "price": 42.00
        })

        assert_equal(resp.status_code, 201, "Should return 201 Created")

        body = json.loads(resp.body)
        assert_equal(body["name"], "Route Test Product", "Name should match")
        assert_equal(body["price"], 42.00, "Price should match")

    def test_get_product_not_found(self):
        resp = self.get("/api/products/99999")

        assert_equal(resp.status_code, 404, "Should return 404 for missing product")

    def test_create_product_validation(self):
        resp = self.post("/api/products", {})

        assert_equal(resp.status_code, 400, "Should return 400 for empty body")

    def test_delete_product(self):
        # Create a product first
        create_resp = self.post("/api/products", {
            "name": "To Be Deleted",
            "price": 1.00
        })
        body = json.loads(create_resp.body)
        product_id = body["id"]

        # Delete it
        delete_resp = self.delete(f"/api/products/{product_id}")
        assert_equal(delete_resp.status_code, 204, "Should return 204 No Content")

        # Verify it is gone
        get_resp = self.get(f"/api/products/{product_id}")
        assert_equal(get_resp.status_code, 404, "Should return 404 after deletion")
```

Test Client Methods

The test client provides methods for all HTTP verbs:

```
# GET request
resp = self.get("/api/products")

# GET with query parameters
resp = self.get("/api/products?category=Electronics&page=2")

# POST with JSON body
resp = self.post("/api/products", {"name": "Widget", "price": 9.99})

# PUT with JSON body
resp = self.put("/api/products/1", {"name": "Updated Widget"})

# PATCH with JSON body
resp = self.patch("/api/products/1", {"price": 12.99})

# DELETE
resp = self.delete("/api/products/1")
```

```
# Request with custom headers
resp = self.get("/api/profile", headers={
    "Authorization": "Bearer eyJhbGciOiJIUzI1NiIs..."
})
```

Response Object

The response object has these properties:

```
resp.status_code   # HTTP status code (200, 201, 404, etc.)
resp.body          # Response body as a string
resp.headers       # Response headers as a dict
resp.content_type  # Content-Type header value
```

---

6. Testing Authentication

Create `tests/test_auth.py`:

```
from tina4_python.test import Test, assert_equal, assert_true, assert_not_none
import json

class AuthTest(Test):

    def test_login_with_valid_credentials(self):
        resp = self.post("/api/auth/login", {
            "email": "admin@example.com",
            "password": "correct-password"
        })

        assert_equal(resp.status_code, 200, "Login should succeed")

        body = json.loads(resp.body)
        assert_not_none(body.get("token"), "Should return a JWT token")
        assert_true(len(body["token"]) > 50, "Token should be a substantial string")

    def test_login_with_invalid_password(self):
        resp = self.post("/api/auth/login", {
            "email": "admin@example.com",
            "password": "wrong-password"
        })

        assert_equal(resp.status_code, 401, "Should reject invalid password")

    def test_login_with_missing_fields(self):
        resp = self.post("/api/auth/login", {
            "email": "admin@example.com"
        })

        assert_true(
            resp.status_code in (400, 401),
            "Should reject missing password"
        )

    def test_protected_route_without_token(self):
        resp = self.get("/api/profile")

        assert_equal(resp.status_code, 401, "Should reject unauthenticated request")
```

```
    def test_protected_route_with_valid_token(self):
        # Login first to get a token
        login_resp = self.post("/api/auth/login", {
            "email": "admin@example.com",
            "password": "correct-password"
        })
        login_body = json.loads(login_resp.body)
        token = login_body["token"]

        # Access protected route with token
        resp = self.get("/api/profile", headers={
            "Authorization": f"Bearer {token}"
        })

        assert_equal(resp.status_code, 200, "Should allow authenticated request")

        body = json.loads(resp.body)
        assert_equal(body["user"]["email"], "admin@example.com", "Should return user data")

    def test_protected_route_with_invalid_token(self):
        resp = self.get("/api/profile", headers={
            "Authorization": "Bearer invalid.token.here"
        })

        assert_equal(resp.status_code, 401, "Should reject invalid token")
```

---

7. Setup and Teardown

Use `set_up()` and `tear_down()` methods to run code before and after each test:

```
from tina4_python.test import Test, assert_equal, assert_not_none
import time

class UserTest(Test):

    def set_up(self):
        # Runs before each test
        user = User()
        user.name = "Test User"
        user.email = f"test-{int(time.time())}@example.com"
        user.save()
        self.user_id = user.id

    def tear_down(self):
        # Runs after each test
        if self.user_id:
            user = User.find(self.user_id)
            if user:
                user.delete()

    def test_user_exists(self):
        user = User.find(self.user_id)
        assert_not_none(user.id, "User should exist")
        assert_equal(user.name, "Test User", "Name should match")

    def test_update_user(self):
        user = User.find(self.user_id)
```

```
        user.name = "Updated Name"
        user.save()

        reloaded = User.find(self.user_id)
        assert_equal(reloaded.name, "Updated Name", "Name should be updated")
```

`set_up()` runs before every test method, and `tear_down()` runs after every test method, regardless of whether the test passed or failed. This keeps tests isolated -- each test starts with a clean state.

---

8. Running Tests

Run All Tests

```
tina4 test
```

Run a Specific Test File

```
tina4 test --file tests/test_product.py
```

Run a Specific Test Method

```
tina4 test --file tests/test_product.py --method test_create_product
```

Verbose Output

```
tina4 test --verbose
```

```
Running tests...

  ProductTest
    [PASS] test_create_product (0.03s)
      assert_not_none: Product should have an ID after save
      assert_true: Product ID should be positive
    [PASS] test_load_product (0.02s)
      assert_equal: Name should match
      assert_equal: Category should match
      assert_equal: Price should match
    [PASS] test_update_product (0.04s)
      assert_equal: Name should be updated
      assert_equal: Price should be updated
    [PASS] test_delete_product (0.02s)
      assert_true: Deleted product should not be loadable
    [PASS] test_select_with_filter (0.05s)
      assert_true: Should find at least 2 FilterCat products
      assert_true: Should include Filter Test A
      assert_true: Should include Filter Test B

  5 tests, 5 passed, 0 failed (0.16s)
```

Verbose mode shows each assertion within each test, along with timing information.

Failed Test Output

When a test fails, you see exactly what went wrong:

```
  ProductTest
    [PASS] test_create_product
```

```
    [FAIL] test_load_product
      assert_equal FAILED: Name should match
        Expected: "Load Test Widget"
        Actual:   "Wrong Name"
        File: tests/test_product.py:34
    [PASS] test_update_product

  3 tests, 2 passed, 1 failed (0.12s)
```

The failure message shows the assertion that failed, what was expected, what was actually received, and the file and line number.

---

9. pytest Integration

If your team already uses pytest, Tina4 tests can run alongside it. Install pytest:

```
uv add --dev pytest
```

Create `pyproject.toml` test configuration:

```
[tool.pytest.ini_options]
testpaths = ["tests"]
python_files = ["test_*.py"]
python_classes = ["*Test"]
python_functions = ["test_*"]
```

Tina4's `Test` class is compatible with pytest. Your test files work with both `tina4 test` and `pytest` without modification.

```
tina4 test
```

```
========================= test session starts ==========================
collected 5 items

tests/test_product.py .....                                       [100%]

========================== 5 passed in 0.34s ===========================
```

Code Coverage

With pytest, you can generate code coverage reports:

```
uv add --dev pytest-cov
tina4 test --cov=src --cov-report=term
```

```
---------- coverage: ... ----------
Name                      Stmts   Miss  Cover
-------------------------------------------
src/routes/products.py       45      5    89%
src/orm/product.py           28      0   100%
src/middleware/auth.py        32      8    75%
-------------------------------------------
TOTAL                        105     13    88%
```

For HTML coverage reports:

```
tina4 test --cov=src --cov-report=html
```

Open `htmlcov/index.html` in your browser to see a visual breakdown of which lines are covered by tests.

---

10. Testing Best Practices

Test One Thing Per Test

Each test method should verify one behavior. If it fails, you know exactly what broke.

```
# Good: each test verifies one thing
def test_create_product_returns_201(self):
    resp = self.post("/api/products", {"name": "Widget", "price": 9.99})
    assert_equal(resp.status_code, 201, "Should return 201")

def test_create_product_returns_created_product(self):
    resp = self.post("/api/products", {"name": "Widget", "price": 9.99})
    body = json.loads(resp.body)
    assert_equal(body["name"], "Widget", "Should return the product name")

# Avoid: testing multiple unrelated things
def test_everything(self):
    # Creates, reads, updates, deletes, checks auth, validates input...
    # When this fails, you don't know which part broke
    pass
```

Use Descriptive Assertion Messages

```
# Good: tells you what went wrong
assert_equal(user.name, "Alice", "User name should be 'Alice' after creation")

# Bad: unhelpful when it fails
assert_equal(user.name, "Alice", "Failed")
```

Isolate Tests

Each test should create its own data and clean up after itself. Never depend on data from another test or from the development database.

```
# Good: creates its own data
def test_delete_product(self):
    product = Product()
    product.name = "Temporary"
    product.price = 1.00
    product.save()

    product.delete()

    check = Product.find(product.id)
    assert_true(check is None, "Product should be deleted")

# Bad: depends on data from another test or the dev database
def test_delete_product(self):
    product = Product.find(1)  # Assumes product with ID 1 exists
    product.delete()
```

---

11. Exercise: Test a User Model and Authentication Flow

Write a complete test suite for a User model and authentication system.

Requirements

Create `tests/test_user_model.py` with these tests:

- **test*create*user** -- Create a user with name, email. Verify the user gets an ID and all fields are correct.
- **test*duplicate*email** -- Try to create two users with the same email. Verify the second one raises an exception or returns an error.
- **test*update*user** -- Create a user, update their name, reload and verify.
- **test*delete*user** -- Create a user, delete them, verify they cannot be loaded.
- **test*select*users** -- Create 3 users, query all, verify count is at least 3.

Create `tests/test_auth_flow.py` with these tests:

- **test*register*new_user** -- POST to `/api/auth/register` with name, email, password. Verify 201 status.
- **test*register*duplicate_email** -- Register the same email twice. Verify the second returns 409 Conflict.
- **test*login*success** -- Login with correct credentials. Verify you get a JWT token.
- **test*login*failure** -- Login with wrong password. Verify 401 status.
- **test*access*protected_route** -- Use the token from login to access `/api/profile`. Verify 200 status and correct user data.
- **test*access*with*expired*token** -- Use a manually crafted expired token. Verify 401 status.

Expected output:

```
tina4 test

Running tests...

  UserModelTest
    [PASS] test_create_user
    [PASS] test_duplicate_email
    [PASS] test_update_user
    [PASS] test_delete_user
    [PASS] test_select_users

  AuthFlowTest
    [PASS] test_register_new_user
    [PASS] test_register_duplicate_email
    [PASS] test_login_success
    [PASS] test_login_failure
    [PASS] test_access_protected_route
    [PASS] test_access_with_expired_token

  11 tests, 11 passed, 0 failed (0.52s)
```

---

12. Solution

tests/test*user*model.py

```
import uuid
from tina4_python.test import Test, assert_equal, assert_true, assert_not_none, assert_raises

class UserModelTest(Test):

    def test_create_user(self):
        user = User()
        user.name = "Test User"
        user.email = f"testuser-{uuid.uuid4().hex[:8]}@example.com"
        user.save()

        assert_not_none(user.id, "User should have an ID after save")
        assert_equal(user.name, "Test User", "Name should match")
        assert_true("@example.com" in user.email, "Email should be set")

    def test_duplicate_email(self):
        email = f"duplicate-{uuid.uuid4().hex[:8]}@example.com"

        user1 = User()
        user1.name = "First User"
        user1.email = email
        user1.save()

        def create_duplicate():
            user2 = User()
            user2.name = "Second User"
            user2.email = email
            user2.save()

        assert_raises(create_duplicate, Exception, "Should reject duplicate email")

    def test_update_user(self):
        user = User()
        user.name = "Original Name"
        user.email = f"update-{uuid.uuid4().hex[:8]}@example.com"
        user.save()

        user_id = user.id
        user.name = "New Name"
        user.save()

        reloaded = User.find(user_id)
        assert_equal(reloaded.name, "New Name", "Name should be updated")

    def test_delete_user(self):
        user = User()
        user.name = "Delete Me"
        user.email = f"delete-{uuid.uuid4().hex[:8]}@example.com"
        user.save()

        user_id = user.id
        user.delete()

        gone = User.find(user_id)
        assert_true(gone is None, "Deleted user should not exist")
```

```
def test_select_users(self):
    for i in range(3):
        user = User()
        user.name = f"Select Test User {i}"
        user.email = f"select-test-{i}-{uuid.uuid4().hex[:8]}@example.com"
        user.save()

    users, count = User.where("1=1")

    assert_true(len(users) >= 3, "Should have at least 3 users")
```

tests/test*auth*flow.py

```
import uuid
import json
from tina4_python.test import Test, assert_equal, assert_true, assert_not_none

class AuthFlowTest(Test):

    def set_up(self):
        self.test_email = f"auth-test-{uuid.uuid4().hex[:8]}@example.com"
        self.test_password = "SecurePassword123!"

    def test_register_new_user(self):
        resp = self.post("/api/auth/register", {
            "name": "Auth Test User",
            "email": self.test_email,
            "password": self.test_password
        })

        assert_equal(resp.status_code, 201, "Registration should return 201")

        body = json.loads(resp.body)
        assert_equal(body["name"], "Auth Test User", "Should return user name")
        assert_not_none(body.get("id"), "Should return user ID")

    def test_register_duplicate_email(self):
        email = f"dup-{uuid.uuid4().hex[:8]}@example.com"

        self.post("/api/auth/register", {
            "name": "First",
            "email": email,
            "password": self.test_password
        })

        resp = self.post("/api/auth/register", {
            "name": "Second",
            "email": email,
            "password": self.test_password
        })

        assert_equal(resp.status_code, 409, "Duplicate email should return 409")

    def test_login_success(self):
        email = f"login-{uuid.uuid4().hex[:8]}@example.com"

        self.post("/api/auth/register", {
            "name": "Login User",
            "email": email,
            "password": self.test_password
        })

        resp = self.post("/api/auth/login", {
            "email": email,
            "password": self.test_password
        })

        assert_equal(resp.status_code, 200, "Login should return 200")

        body = json.loads(resp.body)
```

```
        assert_not_none(body.get("token"), "Should return a token")

    def test_login_failure(self):
        resp = self.post("/api/auth/login", {
            "email": "nobody@example.com",
            "password": "wrong"
        })

        assert_equal(resp.status_code, 401, "Invalid login should return 401")

    def test_access_protected_route(self):
        email = f"profile-{uuid.uuid4().hex[:8]}@example.com"

        self.post("/api/auth/register", {
            "name": "Profile User",
            "email": email,
            "password": self.test_password
        })

        login_resp = self.post("/api/auth/login", {
            "email": email,
            "password": self.test_password
        })

        token = json.loads(login_resp.body)["token"]

        resp = self.get("/api/profile", headers={
            "Authorization": f"Bearer {token}"
        })

        assert_equal(resp.status_code, 200, "Should allow access with valid token")

        body = json.loads(resp.body)
        assert_equal(body["user"]["email"], email, "Should return correct user")

    def test_access_with_expired_token(self):
        resp = self.get("/api/profile", headers={
            "Authorization": "Bearer expired.invalid.token"
        })

        assert_equal(resp.status_code, 401, "Should reject invalid token")
```

---

13. Gotchas

1. Tests Run in Order

Problem: Test B depends on data created in Test A, but sometimes Test B fails.

Cause: While tests within a class run in the order they are defined, test classes may run in any order. If Test B depends on Test A being in a different class, this is fragile.

Fix: Each test should create its own data. Never depend on another test's side effects. Use `set_up()` to create the data each test needs.

2. Test Database vs Development Database

Problem: Running tests deletes your development data.

Cause: Tests are running against the same database as your development server.

Fix: Tina4 uses a separate test database by default (`data/test.db`). If your tests are modifying development data, check that `TINA4_DATABASE_URL` is set correctly in `.env`, or that you are running `tina4 test` (not executing test files manually with `python`).

3. Unique Constraint Failures

Problem: Tests fail with "UNIQUE constraint failed" on the email field.

Cause: Previous test runs left data in the test database, or two tests create records with the same email.

Fix: Use `uuid.uuid4().hex[:8]` or `int(time.time())` to generate unique values in tests:

```
user.email = f"test-{uuid.uuid4().hex[:8]}@example.com"
```

4. Test Method Not Discovered

Problem: You wrote a test method but it does not appear in the output.

Cause: The method name does not start with `test`. Tina4 only runs methods that begin with `test` (case-sensitive).

Fix: Rename the method to start with `test`: `test_create_product`, `test_login_failure`, etc.

5. Assertion Arguments Reversed

Problem: The failure message shows "Expected: 404, Actual: 200" but that seems backward.

Cause: You passed the arguments in the wrong order. `assert_equal(actual, expected)` -- actual comes first, expected comes second.

Fix: Follow the convention: `assert_equal(resp.status_code, 200, "message")`. The first argument is what you got, the second is what you expected.

6. Cannot Test Routes That Require Authentication Setup

Problem: Tests for protected routes fail because the authentication middleware expects a database table or JWT secret that does not exist in the test environment.

Cause: The test database is empty -- it does not have the users table or the JWT secret is not configured.

Fix: Use `set_up()` to create the necessary data:

```
def set_up(self):
    user = User()
    user.create_table()

    user.name = "Test Admin"
    user.email = "admin@test.com"
    user.password_hash = hash_password("password")
```

```
    user.save()
```

7. Tests Pass Locally but Fail in CI

Problem: All tests pass on your machine but fail in continuous integration.

Cause: The CI environment may have a different Python version, missing packages, or different filesystem permissions. Also, tests that depend on time or random values can be flaky.

Fix: Make tests deterministic. Avoid relying on the current time, filesystem state, or external services. If you must test time-dependent behavior, mock the clock or use fixed timestamps. Check that your CI environment matches your local Python version and has all required packages.

Scaffolding

One command. Six files. A working CRUD feature with routes, templates, tests, and Swagger docs -- ready to run.

That is Tina4's scaffolding system. It generates the boilerplate you write by hand in every project: models, migrations, routes, forms, views, and tests. You describe what you want. The generators produce it.

The CRUD Generator

This is the generator most developers reach for first. It creates everything a feature needs in one shot.

```
tina4python generate crud Product --fields "name:string,price:float"
```

That single command creates six files:

#	File	Purpose
1	`src/orm/Product.py`	ORM model with typed fields
2	`migrations/20260401_create_product.sql`	UP migration (CREATE TABLE)
3	`migrations/20260401_create_product.down.sql`	DOWN migration (DROP TABLE)
4	`src/routes/products.py`	CRUD routes with Swagger annotations
5	`src/templates/products/form.html`	Form template with typed inputs and `form_token`
6	`src/templates/products/view.html`	List and detail templates
7	`tests/test_products.py`	pytest stubs for all CRUD operations

What Each File Contains

The **model** maps the `product` table to a Python class:

```
from tina4 import ORM

class Product(ORM):
    table_name = "product"
    fields = {
        "id": "integer",
```

```
        "name": "string",
        "price": "float",
        "created_at": "datetime",
        "updated_at": "datetime"
    }
```

The **migration** creates the table:

```
-- migrations/20260401_create_product.sql
CREATE TABLE product (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    name VARCHAR(255),
    price FLOAT,
    created_at DATETIME DEFAULT CURRENT_TIMESTAMP,
    updated_at DATETIME DEFAULT CURRENT_TIMESTAMP
);
```

The **down migration** reverses it:

```
-- migrations/20260401_create_product.down.sql
DROP TABLE IF EXISTS product;
```

The **routes** file wires up five endpoints with Swagger docs:

```
from tina4 import get, post, put, delete
from src.orm.Product import Product

@get("/api/products", description="List all products")
async def get_products(request, response):
    products = Product().select()
    return response(products)

@get("/api/products/{id}", description="Get a product by ID")
async def get_product(request, response):
    product = Product()
    product.id = request.params["id"]
    product.load()
    return response(product)

@post("/api/products", description="Create a product")
async def create_product(request, response):
    product = Product(request.body)
    product.save()
    return response(product, 201)

@put("/api/products/{id}", description="Update a product")
async def update_product(request, response):
    product = Product(request.body)
    product.id = request.params["id"]
    product.save()
    return response(product)

@delete("/api/products/{id}", description="Delete a product")
async def delete_product(request, response):
    product = Product()
    product.id = request.params["id"]
    product.delete()
    return response(None, 204)
```

The **form template** renders typed inputs with CSRF protection:

```
<form method="POST" action="/api/products">
    <input type="hidden" name="form_token" value="{{ form_token }}">
    <label>Name</label>
    <input type="text" name="name" required>
    <label>Price</label>
    <input type="number" step="0.01" name="price" required>
    <button type="submit">Save</button>
</form>
```

The **test file** stubs out CRUD assertions:

```
import pytest
from tina4 import App

@pytest.fixture
def client():
    app = App()
    return app.test_client()

def test_create_product(client):
    response = client.post("/api/products", json={"name": "Widget", "price": 9.99})
    assert response.status_code == 201

def test_list_products(client):
    response = client.get("/api/products")
    assert response.status_code == 200

def test_get_product(client):
    response = client.get("/api/products/1")
    assert response.status_code == 200

def test_update_product(client):
    response = client.put("/api/products/1", json={"name": "Updated Widget"})
    assert response.status_code == 200

def test_delete_product(client):
    response = client.delete("/api/products/1")
    assert response.status_code == 204
```

Run It

After generating, run the migration and start the server:

```
tina4python migrate
tina4python serve
```

Open Swagger UI at `http://localhost:7146/swagger` and test every endpoint. The scaffolded code works out of the box.

Individual Generators

The CRUD generator calls several smaller generators under the hood. You can call each one directly when you need a single piece.

Model

```
tina4python generate model Product --fields "name:string,price:float"
```

Creates three files: the ORM model (`src/orm/Product.py`), the UP migration, and the DOWN migration. No routes, no templates, no tests.

Route

```
tina4python generate route products --model Product
```

Creates one file: `src/routes/products.py` with CRUD endpoints and Swagger annotations. The model must exist first.

Migration

```
tina4python generate migration add_category_to_product
```

Creates two files: `migrations/20260401_add_category_to_product.sql` and `migrations/20260401_add_category_to_product.down.sql`. Both are empty stubs. You write the SQL.

Middleware

```
tina4python generate middleware AuthLog
```

Creates one file with before and after stubs:

```
from tina4 import middleware

@middleware(before=True)
async def auth_log_before(request):
    print(f"Request: {request.method} {request.url}")
    return request

@middleware(after=True)
async def auth_log_after(request, response):
    print(f"Response: {response.status_code}")
    return response
```

Test

```
tina4python generate test products --model Product
```

Creates one file: `tests/test_products.py` with pytest CRUD stubs.

Form

```
tina4python generate form Product --fields "name:string,price:float"
```

Creates one file: `src/templates/products/form.html` with typed inputs and `form_token`.

View

```
tina4python generate view Product --fields "name:string,price:float"
```

Creates two templates: a list view and a detail view in `src/templates/products/`.

CRUD

```
tina4python generate crud Product --fields "name:string,price:float"
```

Shorthand for running all generators at once: model, migration, route, form, view, and test.

Auth

```
tina4python generate auth
```

Generates the full authentication scaffold: User model, migrations, login/register/logout routes, templates, and tests.

AutoCRUD

AutoCRUD automatically generates REST API endpoints from your ORM models:

- `GET /api/{table}` — List with pagination (`?limit=10&offset=0`)
- `GET /api/{table}/{id}` — Get single record
- `POST /api/{table}` — Create record
- `PUT /api/{table}/{id}` — Update record
- `DELETE /api/{table}/{id}` — Delete record

Usage

```
from tina4_python.crud import AutoCrud
AutoCrud.register(User)
AutoCrud.discover("src/orm", prefix="/api")
```

`AutoCrud.register(Model)` wires up a single model. `AutoCrud.discover(path)` scans a directory and registers every ORM model it finds. Both approaches mount the five standard endpoints automatically — no route files needed.

The Auth Generator

Authentication needs more than one file. The auth generator creates seven:

```
tina4python generate auth
```

#	File	Purpose
1	`src/orm/User.py`	User model with hashed password field
2	`migrations/20260401_create_user.sql`	UP migration
3	`migrations/20260401_create_user.down.sql`	DOWN migration
4	`src/routes/auth.py`	Login, register, logout routes
5	`src/templates/auth/login.html`	Login form

6	`src/templates/auth/register.html`	Registration form
7	`tests/test_auth.py`	Auth flow tests

The generated routes handle password hashing, JWT token creation, and session management. The templates include CSRF tokens. The tests cover registration, login, invalid credentials, and logout.

Run the migration, start the server, and you have working auth:

```
tina4python migrate
tina4python serve
```

Field Types

Generators accept these field types. Each type maps to a specific column type in migrations, input type in forms, and display format in views.

Field Type	Migration Column	Form Input	View Display
`string`	`VARCHAR(255)`	`<input type="text">`	Plain text
`int`	`INTEGER`	`<input type="number">`	Number
`float`	`FLOAT`	`<input type="number" step="0.01">`	Decimal
`bool`	`BOOLEAN`	`<input type="checkbox">`	Yes / No
`text`	`TEXT`	`<textarea>`	Paragraph
`datetime`	`DATETIME`	`<input type="datetime-local">`	Formatted date
`blob`	`BLOB`	`<input type="file">`	Download link

Table Naming Convention

Tina4 uses singular table names. The model name `Product` maps to the table `product`. The model name `OrderItem` maps to `order_item`. The generator handles the conversion.

Combining Generators

Sometimes you want a model with routes but no form. Or a model with a migration but no test. The `--with` flags let you compose:

```
tina4python generate model Product --fields "name:string,price:float" --with-route --with-migrat
```

Available flags:

Flag	Adds
`--with-route`	CRUD route file
`--with-migration`	Migration files (included by default with model)
`--with-test`	Test file
`--with-form`	Form template
`--with-view`	View templates

The `generate crud` command is equivalent to using all `--with` flags at once.

Exercise: Scaffold a Blog

Build a blog with three resources using generators.

Step 1: Generate the auth system.

```
tina4python generate auth
```

Step 2: Scaffold the Post resource.

```
tina4python generate crud Post --fields "title:string,body:text,published:bool"
```

Step 3: Scaffold the Category resource.

```
tina4python generate crud Category --fields "name:string,description:text"
```

Step 4: Add a migration to link posts to categories.

```
tina4python generate migration add_category_id_to_post
```

Edit the migration to add the foreign key:

```
ALTER TABLE post ADD COLUMN category_id INTEGER REFERENCES category(id);
```

Step 5: Run all migrations and start the server.

```
tina4python migrate
tina4python serve
```

You now have a working blog with authentication, posts, categories, and Swagger documentation. Total commands: five. Total hand-written SQL: one line.

Gotchas

Generators Do Not Overwrite

If a file exists, the generator skips it and prints a warning. This protects your edits. To regenerate, delete the file first.

Run Migrate After Generate

The model generator creates migration files. Those files do nothing until you run `tina4python migrate`. Generate and migrate are separate steps by design.

File Naming Matters

The generator derives file names from the model name. `Product` becomes `Product.py` for the model and `products.py` for routes. Do not rename generated files unless you update all imports.

Field Changes Need New Migrations

Changing `--fields` and re-running the generator does not update existing migrations. Create a new migration with `generate migration` and write the ALTER TABLE by hand.

Singular Table Names

Tina4 uses singular table names: `product`, not `products`. The route paths use plural (`/api/products`), but the table stays singular. The generator handles this split.

Swagger / OpenAPI

1. The 47-Endpoint Problem

Your team ships 47 API endpoints. The frontend developer asks what each one accepts. You email a spreadsheet. It rots. You write a wiki page. Nobody updates it. You add comments to the code. Nobody reads them.

Swagger kills this problem for good. It generates interactive API documentation from decorators on your route files. The docs stay current because they live inside the code itself. Your frontend developer browses every endpoint, sees expected request and response formats, and tests endpoints from the browser.

Tina4 Python auto-generates a Swagger UI at `/swagger` from your route decorators. No build step. No extra tooling. Write the decorators. The documentation appears.

---

2. What Swagger/OpenAPI Is

OpenAPI is a specification for describing REST APIs. Swagger is the toolset that reads OpenAPI specs and produces documentation, client SDKs, and server stubs.

An OpenAPI spec describes:

- Every endpoint (path + HTTP method)
- Parameters (path, query, header, body)
- Responses (status codes, body schemas)
- Data schemas (what a "User" or "Product" looks like)
- Authentication requirements
- Grouping and tagging

The spec follows a standard JSON structure. Tools across the industry consume it -- Postman, Insomnia, code generators, testing frameworks, API gateways. One spec feeds them all.

Tina4 builds this spec from Python decorators on your routes. No JSON or YAML by hand. The framework inspects your decorators at startup and constructs the full OpenAPI 3.0.3 document. You write Python. Tina4 writes the spec.

---

3. Accessing the Swagger UI

When `TINA4_DEBUG=true`, the Swagger UI appears at:

```
http://localhost:7146/swagger
```

Open it in your browser. You see all registered routes, organized by tags, with request and response details.

The underlying OpenAPI JSON spec lives at:

```
http://localhost:7146/swagger/json
```

This raw JSON feeds tools that import API definitions. Postman, Insomnia, and code generators all consume it.

```
curl http://localhost:7146/swagger/json
```

```
{
  "openapi": "3.0.3",
  "info": {
    "title": "My Store API",
    "version": "1.0.0"
  },
  "paths": {
    "/api/products": {
      "get": {
        "summary": "List all products",
        "responses": {
          "200": {
            "description": "Successful response"
          }
        }
      }
    }
  }
}
```

---

4. Documenting Routes with Decorators

@description -- Describe What an Endpoint Does

```
from tina4_python.core.router import get, post
from tina4_python.swagger import description

@get("/api/users")
@description("List all users", "Returns a paginated list of all registered users. Supports filte
async def list_users(request, response):
    return response.json({"users": [], "count": 0})
```

The first argument is a short summary (shown in the route list). The second is a detailed description (shown when you expand the route).

@tags -- Organize Endpoints into Groups

```
from tina4_python.swagger import tags

@get("/api/users")
@tags("Users")
@description("List all users")
async def list_users(request, response):
    return response.json({"users": []})

@post("/api/users")
@tags("Users")
@description("Create a new user")
async def create_user(request, response):
    return response.json({"user": request.body}, 201)

@get("/api/products")
@tags("Products")
@description("List all products")
async def list_products(request, response):
    return response.json({"products": []})
```

Tags group related endpoints in the Swagger UI. All "Users" endpoints appear under one collapsible section, all "Products" under another. Without tags, every endpoint sits in one flat list. With tags, the UI becomes navigable.

@example -- Document Request Body

```
from tina4_python.swagger import example

@post("/api/users")
@tags("Users")
@description("Create a new user", "Registers a new user account. Email must be unique.")
@example({
    "name": "Alice Smith",
    "email": "alice@example.com",
    "password": "securePass123",
    "role": "user"
})
async def create_user(request, response):
    return response.json({"user": request.body}, 201)
```

The `@example` decorator shows a sample request body in the Swagger UI. Developers click "Try it out" and the example pre-fills the input fields.

@example_response -- Document Response Body

```
from tina4_python.swagger import example_response

@get("/api/users/{id:int}")
@tags("Users")
@description("Get a user by ID")
@example_response(200, {
    "id": 1,
    "name": "Alice Smith",
    "email": "alice@example.com",
    "role": "user",
    "created_at": "2026-03-22T14:30:00"
})
@example_response(404, {
    "error": "User not found"
})
async def get_user(request, response):
    user_id = request.params["id"]
    return response.json({"id": user_id, "name": "Alice"})
```

Stack multiple `@example_response` decorators for different status codes. Developers see what to expect for both success and failure.

---

5. Documenting Path Parameters

The framework detects path parameters from the route pattern. Add descriptions with the `@description` decorator's extended syntax:

```
@get("/api/users/{id:int}/posts/{status}")
@tags("Posts")
@description(
    "Get user posts by status",
    "Returns all posts for a specific user filtered by status.",
    params={
        "id": "The user's unique identifier (integer)",
        "status": "Post status filter: 'draft', 'published', or 'archived'"
    }
)
async def user_posts(request, response):
    return response.json({"posts": []})
```

Documenting Query Parameters

```
@get("/api/products")
@tags("Products")
@description(
    "List products",
    "Returns a filtered and paginated list of products.",
    query={
        "category": {"type": "string", "description": "Filter by category name", "required": Fal
        "min_price": {"type": "number", "description": "Minimum price filter", "required": False
        "max_price": {"type": "number", "description": "Maximum price filter", "required": False
        "page": {"type": "integer", "description": "Page number (default: 1)", "required": False
        "limit": {"type": "integer", "description": "Items per page (default: 20)", "required":
    }
)
async def list_products(request, response):
    return response.json({"products": [], "count": 0})
```

---

6. Authentication in Swagger

Routes marked with `@secured` show a lock icon in the Swagger UI. Developers click "Authorize", paste their JWT token, and all subsequent requests include the header.

```
from tina4_python.core.router import get, secured
from tina4_python.swagger import description, tags

@get("/api/profile")
@secured()
@tags("Auth")
@description("Get current user profile", "Requires a valid JWT token in the Authorization header
async def get_profile(request, response):
    return response.json({"user": request.user})
```

Public routes marked with `@noauth` show as unlocked:

```
from tina4_python.core.router import post, noauth

@post("/api/login")
@noauth()
@tags("Auth")
@description("Login", "Authenticate with email and password. Returns a JWT token.")
@example({"email": "alice@example.com", "password": "securePass123"})
@example_response(200, {
    "message": "Login successful",
    "token": "eyJhbGciOiJIUzI1NiIs...",
    "user": {"id": 1, "name": "Alice", "email": "alice@example.com"}
})
@example_response(401, {"error": "Invalid email or password"})
async def login(request, response):
    return response.json({"token": "..."})
```

---

7. Try-It-Out from the Swagger UI

Every endpoint in the Swagger UI has a "Try it out" button. Click it and the interface transforms.

- Input fields expand for every parameter
- Example values pre-fill (when provided via `@example`)
- Edit parameters, headers, and the request body
- Click "Execute" -- the actual HTTP request fires against your running server
- The response appears: status code, headers, body

This turns your documentation into a live testing tool. No Postman needed. No curl commands to remember.

Authentication in Try-It-Out

Endpoints that require auth display a lock icon. Click the "Authorize" button at the top of the Swagger UI. Paste your JWT token or API key. The UI stores it and includes the `Authorization` header on every subsequent request.

The workflow for testing a secured endpoint:

- Call your `/api/login` endpoint through Swagger to get a token
- Click "Authorize" and paste the token
- Test any protected endpoint -- the token travels with each request
- Click "Authorize" again and "Logout" to clear it

Tina4 auto-detects auth requirements from `@secured` and `@noauth` decorators. Secured routes show the lock. Public routes show an open lock. Routes without either decorator inherit the default security scheme.

---

8. Customizing the Swagger Info Block

The Swagger UI header and OpenAPI spec carry metadata about your API. Configure this metadata through environment variables in `.env`:

```
TINA4_SWAGGER_TITLE=My Store API
TINA4_SWAGGER_DESCRIPTION=REST API for managing products, orders, and users
TINA4_SWAGGER_VERSION=1.0.0
SWAGGER_DEV_URL=http://localhost:7146
```

These values appear in the OpenAPI spec under the `info` block:

```
{
  "openapi": "3.0.3",
  "info": {
    "title": "My Store API",
    "description": "REST API for managing products, orders, and users",
    "version": "1.0.0"
  },
  "servers": [
    {
      "url": "http://localhost:7146"
    }
  ]
```

```
}
```

Variable	Purpose	Default
`TINA4_SWAGGER_TITLE`	API name shown in the UI header	`Tina4 API`
`TINA4_SWAGGER_DESCRIPTION`	Brief description below the title	(empty)
`TINA4_SWAGGER_VERSION`	API version number	`1.0.0`
`SWAGGER_DEV_URL`	Server URL for the spec	`http://localhost:7146`

When you version your API, update `TINA4_SWAGGER_VERSION` so consumers know which version they target. The title and description give context -- a developer who opens your Swagger page should know what the API does before scrolling.

---

9. Generating Client SDKs from the Spec

The OpenAPI spec at `/swagger/json` feeds code generation tools. One spec produces client libraries in any language.

Using OpenAPI Generator

```
npm install -g @openapitools/openapi-generator-cli

# TypeScript client
openapi-generator-cli generate \
  -i http://localhost:7146/swagger/json \
  -g typescript-fetch \
  -o ./frontend/api-client

# Python client
openapi-generator-cli generate \
  -i http://localhost:7146/swagger/json \
  -g python \
  -o ./python-client
```

The generated code carries types. IDE autocompletion works:

```
const api = new ProductsApi();

const product = await api.getProductById({ id: 42 });
console.log(product.name);  // TypeScript knows this is a string

const newProduct = await api.createProduct({
    name: "Widget",
    category: "General",
    price: 9.99,
    inStock: true
});
```

Update your decorators. Regenerate. The client stays in sync with the server.

Other Generators

The ecosystem supports dozens of languages and frameworks:

Generator	Output
`typescript-fetch`	Browser-ready TypeScript client
`typescript-axios`	Axios-based TypeScript client
`python`	Python client with type hints
`swift5`	iOS/macOS client
`kotlin`	Android/JVM client
`csharp-netcore`	.NET client
`go`	Go client

Every generator reads the same spec. Your API documentation becomes the single source of truth for every consumer.

---

10. Complete API Documentation Example

Here is a fully documented User API with all decorator features:

```
from tina4_python.core.router import get, post, put, delete, noauth, secured, middleware
from tina4_python.swagger import description, tags, example, example_response
from tina4_python.auth import Auth
from tina4_python.database.connection import Database


async def auth_middleware(request, response, next_handler):
    auth_header = request.headers.get("Authorization", "")
    if not auth_header or not auth_header.startswith("Bearer "):
        return response.json({"error": "Authorization required"}, 401)
    token = auth_header[7:]
    if not Auth.valid_token(token):
        return response.json({"error": "Invalid or expired token"}, 401)
    request.user = Auth.get_payload(token)
    return await next_handler(request, response)


@post("/api/users")
@noauth()
@tags("Users")
@description(
    "Register a new user",
    "Creates a new user account. Email must be unique. Password must be at least 8 characters."
)
@example({
    "name": "Alice Smith",
    "email": "alice@example.com",
    "password": "securePass123"
```

```
})
@example_response(201, {
    "message": "Registration successful",
    "user": {"id": 1, "name": "Alice Smith", "email": "alice@example.com", "role": "user"}
})
@example_response(400, {"errors": ["Password must be at least 8 characters"]})
@example_response(409, {"error": "Email already registered"})
async def register_user(request, response):
    # Registration logic...
    return response.json({"message": "Registration successful"}, 201)


@get("/api/users")
@middleware(auth_middleware)
@tags("Users")
@description(
    "List all users",
    "Returns a paginated list of users. Admin only.",
    query={
        "role": {"type": "string", "description": "Filter by role", "required": False},
        "page": {"type": "integer", "description": "Page number", "required": False},
        "limit": {"type": "integer", "description": "Results per page", "required": False}
    }
)
@example_response(200, {
    "users": [
        {"id": 1, "name": "Alice", "email": "alice@example.com", "role": "admin"},
        {"id": 2, "name": "Bob", "email": "bob@example.com", "role": "user"}
    ],
    "count": 2,
    "page": 1,
    "total_pages": 1
})
async def list_users(request, response):
    return response.json({"users": []})


@get("/api/users/{id:int}")
@middleware(auth_middleware)
@tags("Users")
@description(
    "Get a user by ID",
    "Returns the full profile of a single user.",
    params={"id": "The user's unique identifier"}
)
@example_response(200, {
    "id": 1, "name": "Alice Smith", "email": "alice@example.com",
    "role": "user", "created_at": "2026-03-22T14:30:00"
})
@example_response(404, {"error": "User not found"})
async def get_user(request, response):
    return response.json({"id": request.params["id"]})


@put("/api/users/{id:int}")
@middleware(auth_middleware)
@tags("Users")
@description("Update a user", "Update user profile. Users can only update themselves. Admins can
@example({"name": "Alice Johnson", "email": "alice.j@example.com"})
```

```
@example_response(200, {"message": "User updated", "user": {"id": 1, "name": "Alice Johnson"}})
@example_response(404, {"error": "User not found"})
async def update_user(request, response):
    return response.json({"message": "User updated"})


@delete("/api/users/{id:int}")
@middleware(auth_middleware)
@tags("Users")
@description("Delete a user", "Permanently removes a user account. Admin only.")
@example_response(204, None)
@example_response(404, {"error": "User not found"})
async def delete_user(request, response):
    return response.json(None, 204)
```

---

11. Swagger Configuration

Control Swagger behavior in `.env`:

```
# Swagger is only available when debug mode is on (default behavior)
TINA4_DEBUG=true

# Custom API title and version shown in Swagger UI
TINA4_SWAGGER_TITLE=My Store API
TINA4_SWAGGER_VERSION=1.0.0
TINA4_SWAGGER_DESCRIPTION=REST API for the My Store e-commerce platform
```

---

12. Exercise: Document a User API

Take the authentication routes from Chapter 8 (register, login, profile, update profile, change password) and add full Swagger documentation.

Requirements

- Add `@tags("Auth")` to all auth-related endpoints
- Add `@description()` with both summary and detailed description
- Add `@example()` for all POST and PUT endpoints
- Add `@example_response()` for success and error cases on every endpoint
- Document query parameters on any endpoint that accepts them
- Visit `/swagger` and verify all routes appear with examples

Expected Result

Open `http://localhost:7146/swagger`. You should see:

- An "Auth" section with 5 endpoints
- Each endpoint carries a summary and description
- POST/PUT endpoints show example request bodies

- Every endpoint shows example responses for success and error cases
- The "Authorize" button works for testing protected endpoints

---

13. Solution

Update `src/routes/auth.py` with Swagger decorators (showing the decorator additions -- the function bodies remain the same as Chapter 8):

```
from tina4_python.core.router import get, post, put, noauth, middleware
from tina4_python.swagger import description, tags, example, example_response
from tina4_python.auth import Auth
from tina4_python.database.connection import Database


async def auth_middleware(request, response, next_handler):
    auth_header = request.headers.get("Authorization", "")
    if not auth_header or not auth_header.startswith("Bearer "):
        return response.json({"error": "Authorization required"}, 401)
    token = auth_header[7:]
    if not Auth.valid_token(token):
        return response.json({"error": "Invalid or expired token"}, 401)
    request.user = Auth.get_payload(token)
    return await next_handler(request, response)


@post("/api/register")
@noauth()
@tags("Auth")
@description(
    "Register a new account",
    "Creates a new user account. All fields are required. Password must be at least 8 characters
)
@example({
    "name": "Alice Smith",
    "email": "alice@example.com",
    "password": "securePass123"
})
@example_response(201, {
    "message": "Registration successful",
    "user": {"id": 1, "name": "Alice Smith", "email": "alice@example.com", "role": "user", "crea
})
@example_response(400, {"errors": ["Password must be at least 8 characters"]})
@example_response(409, {"error": "Email already registered"})
async def register(request, response):
    # ... same as Chapter 8 ...
    pass


@post("/api/login")
@noauth()
@tags("Auth")
@description(
    "Login",
    "Authenticate with email and password. Returns a JWT token valid for 1 hour. Include the tok
)
@example({"email": "alice@example.com", "password": "securePass123"})
```

```
@example_response(200, {
    "message": "Login successful",
    "token": "eyJhbGciOiJIUzI1NiIs...",
    "user": {"id": 1, "name": "Alice Smith", "email": "alice@example.com", "role": "user"}
})
@example_response(400, {"error": "Email and password are required"})
@example_response(401, {"error": "Invalid email or password"})
async def login(request, response):
    # ... same as Chapter 8 ...
    pass


@get("/api/profile")
@middleware(auth_middleware)
@tags("Auth")
@description(
    "Get current user profile",
    "Returns the profile of the currently authenticated user. Requires a valid JWT token."
)
@example_response(200, {
    "id": 1, "name": "Alice Smith", "email": "alice@example.com",
    "role": "user", "created_at": "2026-03-22 16:00:00"
})
@example_response(401, {"error": "Authorization required"})
async def get_profile(request, response):
    # ... same as Chapter 8 ...
    pass


@put("/api/profile")
@middleware(auth_middleware)
@tags("Auth")
@description(
    "Update profile",
    "Update the current user's name and/or email. Only the fields you include will be updated."
)
@example({"name": "Alice Johnson", "email": "alice.j@example.com"})
@example_response(200, {
    "message": "Profile updated",
    "user": {"id": 1, "name": "Alice Johnson", "email": "alice.j@example.com", "role": "user"}
})
@example_response(409, {"error": "Email already in use by another account"})
async def update_profile(request, response):
    # ... same as Chapter 8 ...
    pass


@put("/api/profile/password")
@middleware(auth_middleware)
@tags("Auth")
@description(
    "Change password",
    "Change the current user's password. Requires the current password for verification. New pas
)
@example({"current_password": "securePass123", "new_password": "evenMoreSecure456"})
@example_response(200, {"message": "Password changed successfully"})
@example_response(400, {"error": "New password must be at least 8 characters"})
@example_response(401, {"error": "Current password is incorrect"})
async def change_password(request, response):
```

```
    # ... same as Chapter 8 ...
    pass
```

---

14. Gotchas

1. Swagger not showing up

Problem: Visiting `/swagger` returns a 404.

Cause: `TINA4_DEBUG` is not set to `true` in your `.env`. The Swagger UI only appears in debug mode.

Fix: Set `TINA4_DEBUG=true` in `.env` and restart the server.

2. Route appears but has no documentation

Problem: A route shows up in Swagger but carries no description, examples, or parameter documentation.

Cause: The route has no Swagger decorators (`@description`, `@example`, etc.).

Fix: Add at least `@description()` and `@tags()` to every route you want documented. Without these, Swagger shows the route with minimal information.

3. Decorator order causes issues

Problem: Adding `@description` or `@tags` breaks the route registration.

Cause: Swagger decorators must sit below the route decorator. Python applies decorators bottom to top, so the route decorator executes first.

Fix: Follow this order (bottom to top): route decorator first, then middleware, then Swagger decorators:

```
@tags("Users")               # Applied last
@description("List users")   # Applied third
@middleware(auth_middleware)  # Applied second
@get("/api/users")           # Applied first
async def list_users(...):
    ...
```

4. Example does not match actual response

Problem: The example response in Swagger differs from what the endpoint returns.

Cause: You updated the route handler but forgot to update the `@example_response` decorator.

Fix: Keep examples in sync with your actual response format. Decorators are not enforced by the runtime. Consider writing tests that validate your responses match the documented examples.

5. Sensitive data in examples

Problem: Your `@example` decorator contains a real password or API key.

Cause: You copy-pasted from a test and forgot to replace real values.

Fix: Use placeholder values in examples: `"password": "securePass123"`, `"token": "eyJhbGciOiJIUzI1NiIs..."`. Never include real credentials.

6. Swagger in production

Problem: Your production API exposes the Swagger UI, revealing all endpoints and their parameters.

Cause: `TINA4_DEBUG=true` in production.

Fix: Set `TINA4_DEBUG=false` in production. The Swagger UI disappears. If you need API docs in production, export the OpenAPI JSON and host it separately with access controls.

7. Too many tags

Problem: Your Swagger UI has 20 tags. Navigation becomes painful.

Cause: You created a separate tag for every resource, sub-resource, and action.

Fix: Use broad tags that group related functionality: "Users", "Products", "Orders", "Auth". Most APIs need 5-10 tags. Avoid tags like "User Profile", "User Settings", "User Notifications" -- use "Users" for all of them.

8. SDK generation produces incorrect types

Problem: The generated TypeScript client has `any` types everywhere.

Cause: Your `@example_response` data uses generic strings instead of typed values.

Fix: Use correct types in examples: strings for text, numbers for numeric values, booleans for flags, arrays for lists. The generator infers types from example values. `"price": 9.99` produces `number`. `"price": "9.99"` produces `string`.

API Client

1. Calling External APIs

Every real application talks to something external. A payment processor. A weather service. A shipping carrier. A CRM. You write the HTTP call, parse the response, handle timeouts, retry on failure, add auth headers.

Tina4's `Api` class makes outbound HTTP requests from your server-side code. It returns a consistent response format, handles auth in one line, and supports SSL and timeout configuration — with no extra libraries.

---

2. Basic Requests

```
from tina4_python.api import Api

api = Api()

# GET
result = api.get("https://api.example.com/products")

# POST with JSON body
result = api.post("https://api.example.com/orders", {
    "product_id": 42,
    "quantity": 3
})

# PUT
result = api.put("https://api.example.com/orders/101", {
    "status": "shipped"
})

# PATCH
result = api.patch("https://api.example.com/orders/101", {
    "tracking_number": "1Z999AA10123456784"
})

# DELETE
result = api.delete("https://api.example.com/orders/101")
```

Every method returns the same structure:

```
{
    "http_code": 200,
    "body": { ... },     # Parsed JSON, or raw string if not JSON
    "headers": { ... },  # Response headers
    "error": None        # None on success, error string on failure
}
```

---

3. Reading the Response

```
from tina4_python.api import Api

api = Api()
result = api.get("https://jsonplaceholder.typicode.com/posts/1")

if result["error"]:
    print(f"Request failed: {result['error']}")
else:
    print(f"Status: {result['http_code']}")
    print(f"Title: {result['body']['title']}")
```

Check `result["error"]` first. If it is `None`, the request succeeded. The HTTP status code is in `result["http_code"]`. The parsed response is in `result["body"]`.

```
# In a route handler
from tina4_python.core.router import get
from tina4_python.api import Api

@get("/api/posts/{post_id}")
async def proxy_post(request, response):
    api = Api()
    post_id = request.params["post_id"]

    result = api.get(f"https://jsonplaceholder.typicode.com/posts/{post_id}")

    if result["error"]:
        return response({"error": result["error"]}, 502)

    if result["http_code"] == 404:
        return response({"error": "Post not found"}, 404)

    return response(result["body"])
```

---

4. Authentication Headers

Bearer Token

```
from tina4_python.api import Api

api = Api(bearer_token="eyJhbGciOiJSUzI1NiJ9...")

result = api.get("https://api.example.com/me")
```

The `Authorization: Bearer <token>` header is sent with every request made by this `Api` instance.

Basic Authentication

```
api = Api(username="api_user", password="secret123")

result = api.get("https://api.example.com/data")
```

Sends `Authorization: Basic <base64(username:password)>` with every request.

Custom Headers

```
api = Api(headers={
    "X-API-Key": "my-api-key-here",
    "X-Client-Version": "1.0.0",
    "Accept": "application/json"
})

result = api.get("https://api.example.com/data")
```

Custom headers are merged with any authentication headers and sent with every request.

Mixing Auth and Custom Headers

```
api = Api(
    bearer_token="eyJhbGciOiJSUzI1NiJ9...",
    headers={"X-Request-Source": "tina4-app"}
)
```

---

5. SSL Verification and Timeouts

```
# Disable SSL verification (dev only, never in production)
api = Api(verify_ssl=False)

# Set a 10-second timeout
api = Api(timeout=10)

# Both
api = Api(verify_ssl=False, timeout=5)
```

The default timeout is 30 seconds. The default SSL behaviour is to verify certificates (`verify_ssl=True`).

Never disable SSL verification in production. If an external API has a self-signed certificate, obtain the CA bundle and pass it explicitly, or ask the provider for their CA bundle.

---

6. Sending Query Parameters

Pass query parameters as a dictionary to the `params` argument:

```
api = Api()

result = api.get("https://api.example.com/products", params={
    "category": "Electronics",
    "page": 1,
    "limit": 20,
    "in_stock": True
})
# Requests: GET /products?category=Electronics&page=1&limit=20&in_stock=True
```

---

7. Real-World Patterns

Payment Gateway

```
import os
from tina4_python.api import Api

class PaymentGateway:
    def __init__(self):
        self.api = Api(
            bearer_token=os.environ["PAYMENT_API_KEY"],
            timeout=15
        )
        self.base = "https://api.payment-provider.com/v1"

    def charge(self, amount_cents, currency, card_token):
        result = self.api.post(f"{self.base}/charges", {
            "amount": amount_cents,
            "currency": currency,
            "source": card_token
        })

        if result["error"]:
            return {"success": False, "error": result["error"]}

        if result["http_code"] not in (200, 201):
            return {
                "success": False,
                "error": result["body"].get("message", "Payment declined")
            }

        return {
            "success": True,
            "charge_id": result["body"]["id"],
            "status": result["body"]["status"]
        }

    def refund(self, charge_id, amount_cents=None):
        body = {"charge": charge_id}
        if amount_cents:
            body["amount"] = amount_cents

        result = self.api.post(f"{self.base}/refunds", body)
        return result["body"]
```

Weather Service

```
import os
from tina4_python.api import Api

class WeatherService:
    def __init__(self):
        self.api = Api(timeout=5)
        self.api_key = os.environ["OPENWEATHER_API_KEY"]
        self.base = "https://api.openweathermap.org/data/2.5"

    def get_current(self, city):
        result = self.api.get(f"{self.base}/weather", params={
            "q": city,
            "appid": self.api_key,
            "units": "metric"
        })

        if result["error"] or result["http_code"] != 200:
            return None

        data = result["body"]
        return {
            "city": data["name"],
            "temp_c": data["main"]["temp"],
            "description": data["weather"][0]["description"],
            "humidity": data["main"]["humidity"]
        }
```

Usage in a route:

```
from tina4_python.core.router import get

weather = WeatherService()

@get("/api/weather/{city}")
async def get_weather(request, response):
    city = request.params["city"]
    data = weather.get_current(city)

    if data is None:
        return response({"error": "Weather data unavailable"}, 502)

    return response(data)
```

---

8. Retry Logic

For transient failures, add retry logic around `Api` calls:

```
import time
from tina4_python.api import Api
from tina4_python.debug import Log

def api_get_with_retry(url, max_retries=3, backoff=1.0, **kwargs):
    api = Api(**kwargs)
    last_error = None

    for attempt in range(1, max_retries + 1):
```

```
        result = api.get(url)

        if not result["error"] and result["http_code"] < 500:
            return result

        last_error = result["error"] or f"HTTP {result['http_code']}"
        Log.warning("API call failed, retrying", url=url, attempt=attempt, error=last_error)

        if attempt < max_retries:
            time.sleep(backoff * attempt)

    Log.error("API call failed after retries", url=url, error=last_error)
    return {"http_code": 0, "body": None, "headers": {}, "error": last_error}
```

---

9. Exercise: Weather Dashboard API

Build a route that fetches weather for multiple cities and returns a dashboard-ready response.

Requirements

- Create `GET /api/dashboard/weather` that:
- Accepts a `cities` query parameter (comma-separated)
- Calls a mock weather API for each city
- Returns aggregated results
- Returns 502 if any city fails
- Use proper error checking on every API call
- Add a 5-second timeout and Bearer token auth

Test with:

```
curl "http://localhost:7146/api/dashboard/weather?cities=London,Berlin,Tokyo"
```

---

10. Solution

Create `src/routes/weather_dashboard.py`:

```
from tina4_python.core.router import get
from tina4_python.api import Api
from tina4_python.debug import Log

# Simulate a weather API with mock data
MOCK_WEATHER = {
    "london": {"city": "London", "temp_c": 12.3, "description": "Cloudy", "humidity": 78},
    "berlin": {"city": "Berlin", "temp_c": 8.1, "description": "Clear", "humidity": 55},
    "tokyo": {"city": "Tokyo", "temp_c": 18.7, "description": "Partly cloudy", "humidity": 65},
    "paris": {"city": "Paris", "temp_c": 14.0, "description": "Rainy", "humidity": 82},
}

def fetch_weather(city_name):
    # In production this would be a real API call:
```

```
    # api = Api(bearer_token=os.environ["WEATHER_API_KEY"], timeout=5)
    # return api.get(f"https://api.weather.com/v1/current?city={city_name}")

    key = city_name.lower().strip()
    data = MOCK_WEATHER.get(key)
    if data is None:
        return {"http_code": 404, "body": None, "headers": {}, "error": f"City not found: {city_
    return {"http_code": 200, "body": data, "headers": {}, "error": None}


@get("/api/dashboard/weather")
async def weather_dashboard(request, response):
    cities_param = request.params.get("cities", "")

    if not cities_param:
        return response({"error": "Provide at least one city via ?cities=City1,City2"}, 400)

    cities = [c.strip() for c in cities_param.split(",") if c.strip()]
    results = []
    errors = []

    for city in cities:
        Log.debug("Fetching weather", city=city)
        result = fetch_weather(city)

        if result["error"]:
            Log.warning("Weather fetch failed", city=city, error=result["error"])
            errors.append({"city": city, "error": result["error"]})
        else:
            results.append(result["body"])

    if errors:
        return response({
            "error": "One or more cities could not be fetched",
            "failed": errors,
            "succeeded": results
        }, 502)

    return response({
        "cities": results,
        "count": len(results)
    })
```

```
curl "http://localhost:7146/api/dashboard/weather?cities=London,Berlin,Tokyo"
```

```
{
  "cities": [
    {"city": "London", "temp_c": 12.3, "description": "Cloudy", "humidity": 78},
    {"city": "Berlin", "temp_c": 8.1, "description": "Clear", "humidity": 55},
    {"city": "Tokyo", "temp_c": 18.7, "description": "Partly cloudy", "humidity": 65}
  ],
  "count": 3
}
```

---

11. Gotchas

1. Not checking result["error"]

Problem: Code accesses `result["body"]["id"]` but the request timed out, so `body` is `None`.

Fix: Always check `result["error"]` before accessing `result["body"]`. Treat any non-None error as a failure.

2. Using verify_ssl=False in production

Problem: A man-in-the-middle intercepts requests to a payment gateway because SSL verification is disabled.

Fix: Only use `verify_ssl=False` against local services or during development. Never disable SSL for external APIs.

3. No timeout set for slow external APIs

Problem: An external API hangs for 90 seconds. Your route handler blocks for 90 seconds. All workers are eventually held waiting.

Fix: Always set `timeout` to a sensible value (5–15 seconds for most external APIs). Return a 502 or 504 to the caller if the external API does not respond in time.

4. Hardcoding API keys

Problem: `api = Api(bearer_token="sk-live-abc123...")` exposes the key in source control.

Fix: Read credentials from environment variables: `bearer_token=os.environ["PAYMENT_API_KEY"]`. Never hardcode secrets.

GraphQL and SOAP

1. The Problem GraphQL Solves

Your mobile app needs a list of products. Each product carries a name, price, 20 image URLs, a full description, 15 review objects, and 8 fields you do not need. REST sends all of it -- 50KB of JSON when you needed 2KB. On a mobile connection, that waste stings.

Now your web dashboard needs the same products plus category, stock status, and supplier info. REST forces you to make three requests and stitch them together, or build a custom endpoint with query parameter parsing.

GraphQL ends both problems. The client asks for the fields it needs. The server returns those fields. One endpoint. One request. One response shaped to fit.

Tina4 includes a built-in GraphQL engine. No external packages. No Strawberry. No Ariadne. Part of the framework.

---

2. GraphQL vs REST -- A Quick Comparison

Aspect	REST	GraphQL
Endpoints	One per resource (`/products`, `/users`)	One endpoint (`/graphql`)
Data shape	Server decides	Client decides
Over-fetching	Common (get all fields)	Never (get only requested fields)
Under-fetching	Common (multiple requests needed)	Never (get related data in one query)
Versioning	`/api/v1/`, `/api/v2/`	Schema evolves, no versioning needed
Caching	HTTP caching works naturally	Requires client-side cache management
Learning curve	Low	Moderate

REST is still great for simple APIs. GraphQL shines when clients have diverse data needs -- mobile apps, dashboards, third-party integrations.

---

3. Enabling GraphQL

GraphQL is available by default in Tina4. The engine serves requests at `/graphql`. If you want to change the endpoint, set it in `.env`:

```
TINA4_GRAPHQL_ENDPOINT=/graphql
```

To verify it is working, start the server and send a test query:

```
curl -X POST http://localhost:7146/graphql \
  -H "Content-Type: application/json" \
  -d '{"query": "{ __schema { queryType { name } } }"}'
```

```
{
  "data": {
    "__schema": {
      "queryType": {
        "name": "Query"
      }
    }
  }
}
```

If you see that response, GraphQL is running.

---

4. Defining a Schema

A GraphQL schema defines the types of data your API can return and the queries and mutations clients can execute.

Create `src/graphql/schema.graphql`:

```
type Product {
    id: Int!
    name: String!
    category: String!
    price: Float!
    inStock: Boolean!
    createdAt: String
}

type Query {
    products: [Product!]!
    product(id: Int!): Product
}
```

This schema says:

- A `Product` has six fields. The `!` means the field is non-nullable.
- The `Query` type has two operations: `products` returns a list of products, and `product(id)` returns a single product (or null if not found).

Tina4 auto-discovers `.graphql` files in `src/graphql/`. You do not need to register them.

---

5. Writing Resolvers

A resolver is the function that runs when a query or mutation is executed. Resolvers live in `src/graphql/` as Python files.

Create `src/graphql/resolvers.py`:

```
from tina4_python.graphql import GraphQL

# Resolve the "products" query
@GraphQL.resolve("Query", "products")
async def resolve_products(root, args):
    products, count = Product.where("1=1")
    return [p.to_dict() for p in products]


# Resolve the "product" query
@GraphQL.resolve("Query", "product")
async def resolve_product(root, args):
    product = Product.find(args["id"])

    if product is None:
        return None

    return product.to_dict()
```

The `@GraphQL.resolve()` decorator takes two arguments:

- **Type name** -- The GraphQL type this resolver belongs to (`Query`, `Mutation`, or a custom type).
- **Field name** -- The field within the type this resolver handles.

The resolver function receives `root` (the parent value, if any) and `args` (the query arguments).

Testing the Queries

List all products:

```
curl -X POST http://localhost:7146/graphql \
  -H "Content-Type: application/json" \
  -d '{"query": "{ products { id name price inStock } }"}'
```

```
{
  "data": {
    "products": [
      {"id": 1, "name": "Wireless Keyboard", "price": 79.99, "inStock": true},
      {"id": 2, "name": "USB-C Hub", "price": 49.99, "inStock": true},
      {"id": 3, "name": "Standing Desk", "price": 549.99, "inStock": false}
    ]
  }
}
```

Notice: the response only contains the four fields we asked for (`id`, `name`, `price`, `inStock`), not `category` or `createdAt`. That is GraphQL in action.

Get a single product with all fields:

```
curl -X POST http://localhost:7146/graphql \
  -H "Content-Type: application/json" \
```

```
  -d '{"query": "{ product(id: 1) { id name category price inStock createdAt } }"}'
```

```
{
  "data": {
    "product": {
      "id": 1,
      "name": "Wireless Keyboard",
      "category": "Electronics",
      "price": 79.99,
      "inStock": true,
      "createdAt": "2026-03-22 14:30:00"
    }
  }
}
```

Request a product that does not exist:

```
curl -X POST http://localhost:7146/graphql \
  -H "Content-Type: application/json" \
  -d '{"query": "{ product(id: 999) { id name } }"}'
```

```
{
  "data": {
    "product": null
  }
}
```

---

6. Mutations

Mutations are how clients create, update, or delete data. They are defined in the schema alongside queries.

Update `src/graphql/schema.graphql`:

```
type Product {
    id: Int!
    name: String!
    category: String!
    price: Float!
    inStock: Boolean!
    createdAt: String
}

input ProductInput {
    name: String!
    category: String
    price: Float!
    inStock: Boolean
}

input ProductUpdateInput {
    name: String
    category: String
    price: Float
    inStock: Boolean
}

type DeleteResult {
```

```
    success: Boolean!
    message: String!
}

type Query {
    products: [Product!]!
    product(id: Int!): Product
    productsByCategory(category: String!): [Product!]!
}

type Mutation {
    createProduct(input: ProductInput!): Product!
    updateProduct(id: Int!, input: ProductUpdateInput!): Product
    deleteProduct(id: Int!): DeleteResult!
}
```

Key concepts:

- `input` **types** define the shape of data the client sends. They are like regular types but used for arguments.
- `Mutation` **type** lists all write operations. By convention, mutations are named with verbs: `createProduct`, `updateProduct`, `deleteProduct`.
- Fields in `ProductUpdateInput` are all optional (no `!`) because the client may only want to update one field.

Mutation Resolvers

Add to `src/graphql/resolvers.py`:

```
@GraphQL.resolve("Mutation", "createProduct")
async def resolve_create_product(root, args):
    input_data = args["input"]

    product = Product()
    product.name = input_data["name"]
    product.category = input_data.get("category", "Uncategorized")
    product.price = float(input_data["price"])
    product.in_stock = bool(input_data.get("inStock", True))
    product.save()

    return product.to_dict()


@GraphQL.resolve("Mutation", "updateProduct")
async def resolve_update_product(root, args):
    product = Product.find(args["id"])

    if product is None:
        return None

    input_data = args["input"]
    if "name" in input_data:
        product.name = input_data["name"]
    if "category" in input_data:
        product.category = input_data["category"]
    if "price" in input_data:
        product.price = float(input_data["price"])
```

```
    if "inStock" in input_data:
        product.in_stock = bool(input_data["inStock"])
    product.save()

    return product.to_dict()


@GraphQL.resolve("Mutation", "deleteProduct")
async def resolve_delete_product(root, args):
    product = Product.find(args["id"])

    if product is None:
        return {"success": False, "message": "Product not found"}

    product.delete()
    return {"success": True, "message": "Product deleted"}


@GraphQL.resolve("Query", "productsByCategory")
async def resolve_products_by_category(root, args):
    products, count = Product.where("category = ?", [args["category"]])
    return [p.to_dict() for p in products]
```

Testing Mutations

Create a product:

```
curl -X POST http://localhost:7146/graphql \
  -H "Content-Type: application/json" \
  -d '{
    "query": "mutation { createProduct(input: { name: \"Desk Lamp\", category: \"Office\", price
  }'
```

```
{
  "data": {
    "createProduct": {
      "id": 4,
      "name": "Desk Lamp",
      "price": 39.99
    }
  }
}
```

Update a product:

```
curl -X POST http://localhost:7146/graphql \
  -H "Content-Type: application/json" \
  -d '{
    "query": "mutation { updateProduct(id: 4, input: { price: 44.99, inStock: false }) { id name
  }'
```

```
{
  "data": {
    "updateProduct": {
      "id": 4,
      "name": "Desk Lamp",
      "price": 44.99,
      "inStock": false
    }
  }
```

```
}
```

Delete a product:

```
curl -X POST http://localhost:7146/graphql \
  -H "Content-Type: application/json" \
  -d '{
    "query": "mutation { deleteProduct(id: 4) { success message } }"
  }'
```

```
{
  "data": {
    "deleteProduct": {
      "success": true,
      "message": "Product deleted"
    }
  }
}
```

---

7. Nested Types and Relationships

The real power of GraphQL: traversing relationships in a single query. Authors, posts, comments -- all in one request.

Update `src/graphql/schema.graphql`:

```
type User {
    id: Int!
    name: String!
    email: String!
    posts: [Post!]!
}

type Post {
    id: Int!
    title: String!
    body: String!
    published: Boolean!
    author: User!
    comments: [Comment!]!
    commentCount: Int!
}

type Comment {
    id: Int!
    authorName: String!
    body: String!
    post: Post!
}

type Query {
    posts: [Post!]!
    post(id: Int!): Post
    user(id: Int!): User
}

type Mutation {
```

```
    createPost(userId: Int!, title: String!, body: String!, published: Boolean): Post!
    addComment(postId: Int!, authorName: String!, body: String!): Comment!
}
```

Add resolvers for the nested types:

```
@GraphQL.resolve("Query", "posts")
async def resolve_posts(root, args):
    posts, count = Post.where("published = ?", [1])
    return [p.to_dict() for p in posts]


@GraphQL.resolve("Query", "post")
async def resolve_post(root, args):
    post = Post.find(args["id"])
    return post.to_dict() if post else None


# Resolve Post.author (nested field)
@GraphQL.resolve("Post", "author")
async def resolve_post_author(post, args):
    user = User.find(post["user_id"])
    return user.to_dict() if user else None


# Resolve Post.comments (nested field)
@GraphQL.resolve("Post", "comments")
async def resolve_post_comments(post, args):
    comments, count = Comment.where("post_id = ?", [post["id"]])
    return [c.to_dict() for c in comments]


# Resolve Post.commentCount (computed field)
@GraphQL.resolve("Post", "commentCount")
async def resolve_post_comment_count(post, args):
    comments, count = Comment.where("post_id = ?", [post["id"]])
    return count


# Resolve User.posts (nested field)
@GraphQL.resolve("User", "posts")
async def resolve_user_posts(user, args):
    posts, count = Post.where("user_id = ?", [user["id"]])
    return [p.to_dict() for p in posts]
```

Now clients can query across relationships in a single request:

```
curl -X POST http://localhost:7146/graphql \
  -H "Content-Type: application/json" \
  -d '{
    "query": "{ posts { id title author { name email } comments { authorName body } commentCount
  }'
```

```
{
  "data": {
    "posts": [
      {
        "id": 1,
        "title": "My First Post",
        "author": {
```

```
          "name": "Alice",
          "email": "alice@example.com"
        },
        "comments": [
          {"authorName": "Bob", "body": "Great post!"}
        ],
        "commentCount": 1
      }
    ]
  }
}
```

One request. Exactly the fields the client needs. No over-fetching. No under-fetching.

---

8. Auto-Generating Schema from ORM Models

If your ORM models have `auto_crud = True`, Tina4 can auto-generate GraphQL types and basic CRUD resolvers for them. Enable this in `.env`:

```
TINA4_GRAPHQL_AUTO_SCHEMA=true
```

With this setting, every ORM model with `auto_crud = True` automatically gets:

- A GraphQL type with fields matching the model properties
- A query to list all records (e.g., `products`)
- A query to get one by ID (e.g., `product(id: Int!)`)
- A mutation to create (e.g., `createProduct(input: ProductInput!)`)
- A mutation to update (e.g., `updateProduct(id: Int!, input: ProductUpdateInput!)`)
- A mutation to delete (e.g., `deleteProduct(id: Int!)`)

You can still define custom resolvers that override the auto-generated ones. Custom resolvers always take precedence.

---

9. The GraphiQL Playground

When `TINA4_DEBUG=true`, Tina4 serves a GraphiQL interactive playground at:

```
http://localhost:7146/graphql/playground
```

GraphiQL gives you:

- A query editor with syntax highlighting and auto-completion
- A documentation explorer showing all types, queries, and mutations
- A results panel showing the response
- Query history

This is the fastest way to explore and test your GraphQL API during development. Type a query on the left, click the play button, and see the results on the right. The documentation explorer on the right sidebar shows every type, field, and argument available in your schema.

GraphiQL is only available when `TINA4_DEBUG=true`. In production, it is disabled automatically.

---

10. Query Variables

For production use, clients should send query variables separately from the query string. This prevents injection and allows query caching.

```
curl -X POST http://localhost:7146/graphql \
  -H "Content-Type: application/json" \
  -d '{
    "query": "query GetProduct($id: Int!) { product(id: $id) { id name price } }",
    "variables": {"id": 1}
  }'
```

```
{
  "data": {
    "product": {
      "id": 1,
      "name": "Wireless Keyboard",
      "price": 79.99
    }
  }
}
```

The query uses `$id` as a placeholder, and the actual value comes from the `variables` object. This is safer and more efficient than string interpolation.

---

11. Exercise: Build a GraphQL API for a Blog

Build a complete GraphQL API for a blog with posts, authors, and comments.

Requirements

- **Schema** -- Define types for `User`, `Post`, and `Comment` with the following fields:
- `User`: id, name, email, posts (nested)
- `Post`: id, title, body, published, author (nested), comments (nested), commentCount (computed)
- `Comment`: id, authorName, body, createdAt
- **Queries**:
- `posts` -- List all published posts
- `post(id: Int!)` -- Get a single post by ID
- `user(id: Int!)` -- Get a user with their posts
- **Mutations**:

- `createPost(userId: Int!, title: String!, body: String!, published: Boolean)` -- Create a new post
- `addComment(postId: Int!, authorName: String!, body: String!)` -- Add a comment to a post
- Use the ORM models from Chapter 6 (User, Post, Comment).

Test with these queries:

```
# List published posts with author names
curl -X POST http://localhost:7146/graphql \
  -H "Content-Type: application/json" \
  -d '{"query": "{ posts { id title author { name } commentCount } }"}'

# Get a specific post with all comments
curl -X POST http://localhost:7146/graphql \
  -H "Content-Type: application/json" \
  -d '{"query": "{ post(id: 1) { title body author { name email } comments { authorName body } }

# Create a post
curl -X POST http://localhost:7146/graphql \
  -H "Content-Type: application/json" \
  -d '{"query": "mutation { createPost(userId: 1, title: \"GraphQL is great\", body: \"Here is w

# Add a comment
curl -X POST http://localhost:7146/graphql \
  -H "Content-Type: application/json" \
  -d '{"query": "mutation { addComment(postId: 1, authorName: \"Carol\", body: \"Nice article!\"
```

---

12. Solution

Schema

Create `src/graphql/blog_schema.graphql`:

```
type User {
    id: Int!
    name: String!
    email: String!
    posts: [Post!]!
}

type Post {
    id: Int!
    title: String!
    body: String!
    published: Boolean!
    author: User!
    comments: [Comment!]!
    commentCount: Int!
    createdAt: String
}

type Comment {
    id: Int!
    authorName: String!
```

```
    body: String!
    createdAt: String
}

type Query {
    posts: [Post!]!
    post(id: Int!): Post
    user(id: Int!): User
}

type Mutation {
    createPost(userId: Int!, title: String!, body: String!, published: Boolean): Post!
    addComment(postId: Int!, authorName: String!, body: String!): Comment!
}
```

Resolvers

Create `src/graphql/blog_resolvers.py`:

```
from tina4_python.graphql import GraphQL


@GraphQL.resolve("Query", "posts")
async def resolve_posts(root, args):
    posts, count = Post.where("published = ?", [1])
    return [p.to_dict() for p in posts]


@GraphQL.resolve("Query", "post")
async def resolve_post(root, args):
    post = Post.find(args["id"])
    return post.to_dict() if post else None


@GraphQL.resolve("Query", "user")
async def resolve_user(root, args):
    user = User.find(args["id"])
    return user.to_dict() if user else None


@GraphQL.resolve("Post", "author")
async def resolve_post_author(post, args):
    user = User.find(post["user_id"])
    return user.to_dict() if user else None


@GraphQL.resolve("Post", "comments")
async def resolve_post_comments(post, args):
    comments, count = Comment.where("post_id = ?", [post["id"]])
    return [c.to_dict() for c in comments]


@GraphQL.resolve("Post", "commentCount")
async def resolve_post_comment_count(post, args):
    comments, count = Comment.where("post_id = ?", [post["id"]])
    return count


@GraphQL.resolve("User", "posts")
```

```
async def resolve_user_posts(user, args):
    posts, count = Post.where("user_id = ?", [user["id"]])
    return [p.to_dict() for p in posts]


@GraphQL.resolve("Mutation", "createPost")
async def resolve_create_post(root, args):
    user = User.find(args["userId"])

    if user is None:
        raise Exception("User not found")

    post = Post()
    post.user_id = args["userId"]
    post.title = args["title"]
    post.body = args["body"]
    post.published = bool(args.get("published", False))
    post.save()

    return post.to_dict()


@GraphQL.resolve("Mutation", "addComment")
async def resolve_add_comment(root, args):
    post = Post.find(args["postId"])

    if post is None:
        raise Exception("Post not found")

    comment = Comment()
    comment.post_id = args["postId"]
    comment.author_name = args["authorName"]
    comment.body = args["body"]
    comment.save()

    return comment.to_dict()
```

Expected output for `{ posts { id title author { name } commentCount } }`:

```
{
  "data": {
    "posts": [
      {
        "id": 1,
        "title": "My First Post",
        "author": {"name": "Alice"},
        "commentCount": 1
      },
      {
        "id": 2,
        "title": "GraphQL is great",
        "author": {"name": "Alice"},
        "commentCount": 0
      }
    ]
  }
}
```

---

13. Input Validation

Tina4's GraphQL engine validates arguments against their declared types before the resolver runs. If validation fails, the resolver is skipped and a clean error is returned.

What Gets Validated

- **Non-null enforcement** — `ID!` means the argument cannot be null or empty
- **Scalar type checking** — `Int` must be numeric, `Boolean` must be bool, `String` must be scalar
- **Type coercion** — string `"123"` coerces to Int `123`, string `"true"` coerces to Boolean `True`
- **List item validation** — `[Int!]` checks each item in the list

Example

```
gql.schema.add_query("user", {"id": "ID!"}, "User", lambda r, a, c: find_user(a["id"]))
```

Query with missing required argument:

```
{ user { id name } }
```

Response:

```
{
  "data": { "user": null },
  "errors": [
    { "message": "Argument 'id' on field 'user' is required (type: ID!)", "path": ["user"] }
  ]
}
```

The resolver never runs. The client gets a clear error describing what went wrong and where.

---

14. Field-Level Auth Directives

Control access to individual fields using directives. No middleware setup, no wrapper functions — add a directive to the query and Tina4 checks the context.

Available Directives

Directive	Meaning
`@auth`	Field requires any authenticated user
`@role(role: "admin")`	Field requires a specific role
`@guest`	Field is only accessible to unauthenticated users

Passing Auth Context

The `execute()` method accepts a `context` parameter. Pass the authenticated user from your request handler:

```
from tina4_python import get
from tina4_python.graphql import GraphQL

gql = GraphQL()

@get("/api/graphql")
async def graphql_handler(request, response):
    body = request.body
    context = {}
    if request.user:
        context["user"] = {"id": request.user.id, "role": request.user.role}

    result = gql.execute(body["query"], body.get("variables"), context)
    return response.json(result)
```

Using Directives in Queries

```
# Only authenticated users can see this field
{ profile @auth { id name email } }

# Only admins can see salary
{ employees { name salary @role(role: "admin") } }

# Only guests see the registration prompt
{ registrationBanner @guest }
```

When a directive check fails, the field returns `null` and is silently excluded from the response. No error is added — the field simply doesn't appear, as if it doesn't exist for that user.

---

15. Gotchas

1. Schema File Not Found

Problem: Queries return errors about unknown types or fields.

Cause: The `.graphql` file is not in `src/graphql/`, or has a syntax error that prevents parsing.

Fix: Make sure schema files are in `src/graphql/` and end with `.graphql`. Check for syntax errors -- a missing `!` or unmatched brace will cause the entire schema to fail silently.

2. Resolver Not Called

Problem: A query returns `null` even though data exists.

Cause: The resolver is not registered, or the type/field names do not match the schema exactly. GraphQL is case-sensitive.

Fix: Verify that `@GraphQL.resolve("Query", "products")` matches `type Query { products: ... }` exactly. Check capitalization.

3. Nested Resolver Returns Wrong Data

Problem: `post.author` returns the entire post instead of the user.

Cause: The nested resolver receives the parent object as its first argument (`post`), not a fresh model. If you return the parent instead of loading the related record, you get the wrong data.

Fix: In a nested resolver, always load the related record from the database using the foreign key from the parent:

```
@GraphQL.resolve("Post", "author")
async def resolve_post_author(post, args):
    user = User.find(post["user_id"])  # Use the foreign key from the parent
    return user.to_dict() if user else None
```

4. Mutation Input Not Parsed

Problem: `args["input"]` is `None` inside a mutation resolver.

Cause: The mutation signature in the schema uses an `input` type, but the client query passes arguments directly instead of wrapping them in an `input` object.

Fix: Match the query to the schema. If the schema says `createProduct(input: ProductInput!)`, the client must send `createProduct(input: { name: "Widget", price: 9.99 })`, not `createProduct(name: "Widget", price: 9.99)`.

5. N+1 Query Problem

Problem: Loading 50 posts with their authors generates 51 database queries (1 for posts + 50 for authors).

Cause: Each nested resolver runs independently. When you resolve `author` for each post, that is one query per post.

Fix: Use the `include` parameter in your ORM queries to eager-load relationships. Tina4's GraphQL engine injects the current selection set into context as `context["__selections"]`, so `fromOrm` resolvers can detect which relationships the client requested and batch-load them:

```
# The fromOrm resolver automatically checks __selections and eager-loads
gql.schema.from_orm(Post)  # Generates queries that use include= when sub-fields match relations
```

For custom resolvers, check `context.get("__selections")` and pre-load related data in a single query.

6. GraphQL Playground Returns 404

Problem: `/graphql/playground` returns a 404 error.

Cause: `TINA4_DEBUG` is set to `false`. The playground is only available in debug mode.

Fix: Set `TINA4_DEBUG=true` in your `.env` file. The playground is intentionally disabled in production.

7. Type Mismatch Between Schema and Resolver

Problem: A field declared as `Int!` in the schema receives a string from the resolver.

Cause: Python's dynamic typing means your resolver might return `"42"` (a string) for an `Int!` field.

Fix: Tina4's input validation now coerces argument types automatically (string `"123"` becomes Int `123`). For return values, `to_dict()` handles type casting for ORM model properties. For computed fields, cast explicitly:

```
return {
    "id": int(product.id),
    "price": float(product.price),
    "inStock": bool(product.in_stock)
}
```

---

14. SOAP / WSDL Services

What is SOAP?

SOAP (Simple Object Access Protocol) is an XML-based messaging protocol for exchanging structured data between systems. It predates REST and GraphQL, but remains common in enterprise integrations -- banking, healthcare, government, and ERP systems all rely on SOAP services.

A WSDL (Web Services Description Language) file describes a SOAP service: what operations are available, what parameters they accept, and what they return. Clients use the WSDL to auto-generate code for calling the service.

Tina4 includes a zero-dependency SOAP 1.1 server that generates WSDL definitions automatically from Python classes and type annotations. No XML authoring required.

Defining a SOAP Service

Create a class that extends `WSDL`. Each method decorated with `@wsdl_operation` becomes a SOAP operation. The decorator takes a dict describing the response fields and their types.

```
from tina4_python.wsdl import WSDL, wsdl_operation
from tina4_python.core.router import get, post

class Calculator(WSDL):
    @wsdl_operation({"Result": int})
    def Add(self, a: int, b: int):
        return {"Result": a + b}

    @wsdl_operation({"Result": int})
    def Multiply(self, a: int, b: int):
        return {"Result": a * b}

@get("/calculator")
@post("/calculator")
async def calculator_endpoint(request, response):
    service = Calculator(request)
    return response(service.handle())
```

That is the entire service. The `handle()` method inspects the request:

- **GET** (or `?wsdl` query parameter) -- returns the auto-generated WSDL definition
- **POST** with SOAP XML body -- parses the XML, finds the operation, converts parameters, calls the method, and returns a SOAP XML response

Type Annotations Map to XSD

Python type annotations on method parameters control how incoming XML values are converted. The WSDL generator also uses them to produce correct XSD type declarations.

Python type	XSD type
`str`	`xsd:string`
`int`	`xsd:int`
`float`	`xsd:double`
`bool`	`xsd:boolean`
`bytes`	`xsd:base64Binary`
`List[T]`	Element of type T (repeated)
`Optional[T]`	Type T (nillable)

A More Complete Example

```
from typing import List, Optional
from tina4_python.wsdl import WSDL, wsdl_operation
from tina4_python.core.router import get, post

class UserService(WSDL):
    @wsdl_operation({"Name": str, "Email": str, "Active": bool})
    def GetUser(self, user_id: int):
        user = User.find(user_id)
        if user:
            return {
                "Name": user.name,
                "Email": user.email,
                "Active": bool(user.active),
            }
        return {"Name": "", "Email": "", "Active": False}

    @wsdl_operation({"Total": int, "Average": float, "Error": Optional[str]})
    def SumList(self, Numbers: List[int]):
        if not Numbers:
            return {"Total": 0, "Average": 0.0, "Error": "Empty list"}
        return {
            "Total": sum(Numbers),
            "Average": sum(Numbers) / len(Numbers),
            "Error": None,
        }

@get("/api/users/soap")
@post("/api/users/soap")
async def user_soap(request, response):
    service = UserService(request)
    return response(service.handle())
```

Lifecycle Hooks

Override `on_request` and `on_result` to add validation, logging, or transformation around every operation call.

```
class AuditedService(WSDL):
    def on_request(self, request):
        print(f"SOAP request from {request.headers.get('x-forwarded-for', 'unknown')}")

    def on_result(self, result):
        result["Timestamp"] = datetime.now().isoformat()
        return result

    @wsdl_operation({"Status": str, "Timestamp": str})
    def Ping(self):
        return {"Status": "ok"}
```

Testing with curl

Fetch the WSDL definition:

```
curl http://localhost:7146/calculator?wsdl
```

Call the Add operation with a SOAP request:

```
curl -X POST http://localhost:7146/calculator \
  -H "Content-Type: text/xml" \
  -d '<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <Add>
      <a>5</a>
      <b>3</b>
    </Add>
  </soap:Body>
</soap:Envelope>'
```

Response:

```
<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <AddResponse>
      <Result>8</Result>
    </AddResponse>
  </soap:Body>
</soap:Envelope>
```

If the operation name is wrong or the XML is malformed, the service returns a SOAP fault:

```
<soap:Fault>
  <faultcode>Client</faultcode>
  <faultstring>Unknown operation: Subtract</faultstring>
</soap:Fault>
```

When to Use SOAP vs REST vs GraphQL

Scenario	Use
New API for web/mobile clients	REST or GraphQL
Integrating with legacy enterprise systems (SAP, banks, government)	SOAP
Clients require a machine-readable service contract	SOAP (WSDL)
Simple internal microservices	REST
Clients with diverse data needs	GraphQL

SOAP is rarely the right choice for new APIs, but when you need to expose a service that legacy systems can consume, Tina4 makes it straightforward -- define a class, annotate your types, and the framework handles the XML.

Real-time with WebSocket

1. The Refresh Button Problem

Your project management app needs live updates. Someone moves a card from "In Progress" to "Done." Everyone else on the team should see it. No page refresh. No polling. No waiting.

Traditional HTTP is request-response. The client asks. The server answers. The server cannot push data on its own. WebSocket breaks that wall. A persistent, bi-directional connection between browser and server. Either side sends messages at any time. The connection stays open until one side closes it.

Tina4 treats WebSocket the same way it treats routing. A decorator. A handler. The path determines which handler processes the connection.

---

2. What WebSocket Is

HTTP works like this:

```
Client: "Give me /api/products"
Server: "Here are the products" (connection closes)
Client: "Give me /api/products" (new connection)
Server: "Here are the products" (connection closes)
```

WebSocket works like this:

```
Client: "I want to upgrade to WebSocket on /ws/chat"
Server: "Upgrade accepted. Connection open."
Client: "Hello everyone!"
Server: "Alice says: Hello everyone!"  (pushed to all clients)
Server: "Bob joined the chat"  (pushed to all clients, at any time)
Client: "Goodbye!"
Server: "Alice says: Goodbye!"
Client: (closes connection)
```

Key differences:

- **Persistent**: The connection stays open. No repeated handshakes.
- **Bi-directional**: The server can push data without the client asking.
- **Low overhead**: After the initial handshake, messages are tiny (no HTTP headers per message).
- **Real-time**: Messages arrive within milliseconds.

---

3. @websocket -- WebSocket as a Route

In Tina4, you define WebSocket handlers using the `@websocket` decorator:

```
from tina4_python.core.router import websocket
```

```
@websocket("/ws/echo")
async def echo_handler(connection, event, data):
    if event == "message":
        await connection.send(f"Echo: {data}")
```

This is the simplest WebSocket handler: it receives a message and sends it back with "Echo: " prepended. The handler receives three arguments:

- **connection**: The WebSocket connection object. Use it to send messages.
- **event**: The event type: `"open"`, `"message"`, or `"close"`.
- **data**: The message data (only present for `"message"` events).

Starting the Server

WebSocket runs alongside your HTTP server:

```
tina4 serve
```

```
  Tina4 Python v3.0.0
  HTTP server running at http://0.0.0.0:7146
  WebSocket server running at ws://0.0.0.0:7146
  Press Ctrl+C to stop
```

Both HTTP and WebSocket share the same port. The server detects the protocol upgrade request and routes it to the correct handler.

---

4. Connection Events

Every WebSocket connection goes through three lifecycle events:

Open

Fires when a client connects:

```
import json
from tina4_python.core.router import websocket

@websocket("/ws/notifications")
async def notifications_handler(connection, event, data):
    if event == "open":
        print(f"Client connected: {connection.id}")
        await connection.send(json.dumps({
            "type": "welcome",
            "message": "Connected to notifications",
            "connection_id": connection.id
        }))
```

Message

Fires when a client sends data:

```
import json
from datetime import datetime, timezone
from tina4_python.core.router import websocket

@websocket("/ws/notifications")
```

```
async def notifications_handler(connection, event, data):
    if event == "open":
        await connection.send(json.dumps({
            "type": "welcome",
            "connection_id": connection.id
        }))

    if event == "message":
        message = json.loads(data)
        print(f"Received from {connection.id}: {data}")

        if message.get("type") == "ping":
            await connection.send(json.dumps({
                "type": "pong",
                "timestamp": datetime.now(timezone.utc).isoformat()
            }))
```

Close

Fires when a client disconnects:

```
import json
from tina4_python.core.router import websocket

@websocket("/ws/notifications")
async def notifications_handler(connection, event, data):
    if event == "open":
        print(f"Client connected: {connection.id}")

    if event == "message":
        print(f"Message from {connection.id}: {data}")

    if event == "close":
        print(f"Client disconnected: {connection.id}")
        # Clean up: remove from tracking, notify others, etc.
```

A Complete Handler

Here is a handler that responds to all three events:

```
import json
from datetime import datetime, timezone
from tina4_python.core.router import websocket

@websocket("/ws/chat")
async def chat_handler(connection, event, data):
    if event == "open":
        print(f"[Chat] New connection: {connection.id}")
        await connection.send(json.dumps({
            "type": "system",
            "message": "Welcome to the chat!",
            "your_id": connection.id
        }))

    elif event == "message":
        message = json.loads(data)
        print(f"[Chat] {connection.id}: {message.get('text', data)}")

        await connection.send(json.dumps({
            "type": "message",
```

```
            "from": connection.id,
            "text": message.get("text", data),
            "timestamp": datetime.now(timezone.utc).isoformat()
        }))

    elif event == "close":
        print(f"[Chat] Disconnected: {connection.id}")
```

---

5. Sending to a Single Client

`connection.send()` sends a message to the specific client that triggered the event:

```
import json
import psutil
from datetime import datetime, timezone
from tina4_python.core.router import websocket

@websocket("/ws/private")
async def private_handler(connection, event, data):
    if event == "message":
        message = json.loads(data)
        action = message.get("action", "")

        if action == "get-time":
            await connection.send(json.dumps({
                "type": "time",
                "server_time": datetime.now(timezone.utc).isoformat()
            }))

        if action == "get-status":
            await connection.send(json.dumps({
                "type": "status",
                "uptime": 3600,
                "connections": 42,
                "memory_mb": round(psutil.Process().memory_info().rss / 1024 / 1024, 2)
            }))
```

Only the client that sent the message receives the response. Other connected clients do not see it.

---

6. Broadcasting to All Clients

`connection.broadcast()` sends a message to every client connected to the same WebSocket path:

```
import json
from datetime import datetime, timezone
from tina4_python.core.router import websocket

@websocket("/ws/announcements")
async def announcements_handler(connection, event, data):
    if event == "open":
        await connection.broadcast(json.dumps({
            "type": "system",
```

```
            "message": "A new user joined",
            "online_count": connection.connection_count()
        }))

    if event == "message":
        message = json.loads(data)

        await connection.broadcast(json.dumps({
            "type": "announcement",
            "from": connection.id,
            "text": message.get("text", ""),
            "timestamp": datetime.now(timezone.utc).isoformat()
        }))

    if event == "close":
        await connection.broadcast(json.dumps({
            "type": "system",
            "message": "A user left",
            "online_count": connection.connection_count()
        }))
```

When one client sends a message, every client connected to `/ws/announcements` receives it. This is the foundation for chat rooms, live dashboards, and collaborative editing.

Broadcast Excluding Sender

Sometimes you want to send to everyone except the client that triggered the event:

```
if event == "message":
    message = json.loads(data)

    # Send to sender (confirmation)
    await connection.send(json.dumps({
        "type": "sent",
        "text": message.get("text")
    }))

    # Send to everyone else
    await connection.broadcast(json.dumps({
        "type": "message",
        "from": message.get("username", "Anonymous"),
        "text": message.get("text"),
        "timestamp": datetime.now(timezone.utc).isoformat()
    }), exclude_self=True)
```

The `exclude_self=True` argument to `broadcast()` excludes the sender. The sender gets a "sent" confirmation, and everyone else gets the "message".

Sending JSON

Use `connection.send_json()` to send a Python dict or list as a JSON string without manually calling `json.dumps()`:

```
@websocket("/ws/status")
async def status_handler(connection, event, data):
    if event == "open":
        await connection.send_json({
            "type": "welcome",
            "connection_id": connection.id
```

```
        })

    if event == "message":
        await connection.send_json({
            "type": "ack",
            "received": data
        })
```

`send_json()` serialises the data to JSON for you. It is equivalent to `connection.send(json.dumps(data))` but saves you the import and the call.

Closing a Connection

Use `connection.close()` to close the connection from the server side:

```
@websocket("/ws/secure")
async def secure_handler(connection, event, data):
    if event == "open":
        token = connection.params.get("token")
        if not token or not valid_token(token):
            await connection.send_json({"error": "Unauthorized"})
            await connection.close()
            return

        await connection.send_json({"type": "welcome"})
```

Connection Methods Summary

Method	Description
`await connection.send(message)`	Send a string message to this connection only
`await connection.send_json(data)`	Send a dict/list as JSON to this connection only
`await connection.broadcast(message)`	Send to all connections on the same path
`await connection.broadcast(message, exclude_self=True)`	Send to all except this connection
`await connection.close()`	Close this connection from the server side
`connection.id`	Unique identifier for this connection
`connection.params`	Path parameters extracted from the URL
`connection.connection_count()`	Number of active connections on this path

---

7. Path-Scoped Isolation

Different WebSocket paths are completely isolated. Clients connected to `/ws/chat/room-1` do not see messages from `/ws/chat/room-2`:

```
import json
from tina4_python.core.router import websocket

@websocket("/ws/chat/{room}")
async def room_handler(connection, event, data):
    room = connection.params["room"]

    if event == "open":
        print(f"[Room {room}] New connection: {connection.id}")
        await connection.broadcast(json.dumps({
            "type": "system",
            "message": f"Someone joined room {room}",
            "room": room,
            "online": connection.connection_count()
        }))

    if event == "message":
        message = json.loads(data)

        await connection.broadcast(json.dumps({
            "type": "message",
            "room": room,
            "from": message.get("username", "Anonymous"),
            "text": message.get("text", ""),
            "timestamp": datetime.now(timezone.utc).isoformat()
        }))

    if event == "close":
        await connection.broadcast(json.dumps({
            "type": "system",
            "message": f"Someone left room {room}",
            "room": room,
            "online": connection.connection_count()
        }))
```

Connect to different rooms:

```
ws://localhost:7146/ws/chat/general    -- general chat
ws://localhost:7146/ws/chat/random     -- random chat
ws://localhost:7146/ws/chat/dev-team   -- dev team chat
```

Broadcasting in `/ws/chat/general` only reaches clients connected to `/ws/chat/general`. The `dev-team` and `random` rooms are separate.

Chat rooms. Project-specific notifications. Per-user channels. No extra configuration. The URL path is the isolation boundary.

---

8. Building a Live Chat

Here is a complete chat application with usernames, typing indicators, and message history:

WebSocket Handler

Create `src/routes/chat_ws.py`:

```
import json
from datetime import datetime, timezone
```

```
from tina4_python.core.router import websocket

chat_users = {}

@websocket("/ws/livechat/{room}")
async def livechat_handler(connection, event, data):
    room = connection.params["room"]

    if event == "open":
        chat_users[connection.id] = {
            "id": connection.id,
            "username": "Anonymous",
            "room": room,
            "joined_at": datetime.now(timezone.utc).isoformat()
        }

        await connection.send(json.dumps({
            "type": "welcome",
            "message": f"Connected to room: {room}",
            "your_id": connection.id,
            "online": connection.connection_count()
        }))

    if event == "message":
        message = json.loads(data)
        msg_type = message.get("type", "message")

        if msg_type == "set-username":
            old_name = chat_users[connection.id]["username"]
            chat_users[connection.id]["username"] = message["username"]

            await connection.broadcast(json.dumps({
                "type": "system",
                "message": f"{old_name} is now known as {message['username']}"
            }))

        if msg_type == "message":
            username = chat_users[connection.id]["username"]

            await connection.broadcast(json.dumps({
                "type": "message",
                "from": username,
                "from_id": connection.id,
                "text": message.get("text", ""),
                "timestamp": datetime.now(timezone.utc).isoformat()
            }))

        if msg_type == "typing":
            username = chat_users[connection.id]["username"]

            await connection.broadcast(json.dumps({
                "type": "typing",
                "username": username
            }), exclude_self=True)

    if event == "close":
        username = chat_users.get(connection.id, {}).get("username", "Unknown")
        chat_users.pop(connection.id, None)
```

```
        await connection.broadcast(json.dumps({
            "type": "system",
            "message": f"{username} left the chat",
            "online": connection.connection_count()
        }))
```

---

9. Live Notifications

WebSocket is great for pushing notifications to users in real time:

```
import json
from datetime import datetime, timezone
from tina4_python.core.router import websocket, post, push_to_websocket

@websocket("/ws/notifications/{user_id}")
async def notification_handler(connection, event, data):
    user_id = connection.params["user_id"]

    if event == "open":
        print(f"[Notifications] User {user_id} connected")
        await connection.send(json.dumps({
            "type": "connected",
            "message": "Listening for notifications"
        }))

    if event == "message":
        message = json.loads(data)
        if message.get("type") == "mark-read":
            print(f"[Notifications] User {user_id} read notification {message['id']}")


@post("/api/orders/{order_id}/ship")
async def ship_order(request, response):
    order_id = request.params["order_id"]
    user_id = request.body.get("user_id", 0)

    # Update order status in database...

    # Send real-time notification via WebSocket
    await push_to_websocket(f"/ws/notifications/{user_id}", json.dumps({
        "type": "notification",
        "title": "Order Shipped",
        "message": f"Your order #{order_id} has been shipped!",
        "action_url": f"/orders/{order_id}",
        "timestamp": datetime.now(timezone.utc).isoformat()
    }))

    return response({"message": "Order shipped, user notified"})
```

The `push_to_websocket()` function lets your HTTP handlers send messages to WebSocket clients. This bridges the gap between traditional request-response endpoints and real-time notifications.

---

10. Connecting from JavaScript

Tina4 provides `frond.js`, a built-in JavaScript helper library that includes WebSocket support:

Basic Connection

```
<script src="/js/frond.js"></script>
<script>
    const ws = frond.ws("/ws/chat/general");

    ws.on("open", function () {
        console.log("Connected to chat");
    });

    ws.on("message", function (data) {
        const message = JSON.parse(data);
        console.log("Received:", message);

        if (message.type === "message") {
            addMessageToUI(message.from, message.text, message.timestamp);
        }

        if (message.type === "system") {
            addSystemMessage(message.message);
        }
    });

    ws.on("close", function () {
        console.log("Disconnected from chat");
    });

    function sendMessage(text) {
        ws.send(JSON.stringify({
            type: "message",
            text: text
        }));
    }

    function setUsername(name) {
        ws.send(JSON.stringify({
            type: "set-username",
            username: name
        }));
    }
</script>
```

Auto-Reconnect

`frond.js` automatically reconnects when the connection drops:

```
const ws = frond.ws("/ws/notifications/42", {
    reconnect: true,           // Enable auto-reconnect (default: true)
    reconnectInterval: 3000,   // Retry every 3 seconds
    maxReconnectAttempts: 10   // Give up after 10 attempts
});

ws.on("reconnect", function (attempt) {
    console.log("Reconnecting... attempt " + attempt);
});
```

```
ws.on("reconnect_failed", function () {
    console.log("Failed to reconnect after maximum attempts");
});
```

Using Native WebSocket

If you prefer not to use `frond.js`, the native WebSocket API works too:

```
const ws = new WebSocket("ws://localhost:7146/ws/chat/general");

ws.onopen = function () {
    console.log("Connected");
    ws.send(JSON.stringify({ type: "set-username", username: "Alice" }));
};

ws.onmessage = function (event) {
    const message = JSON.parse(event.data);
    console.log("Received:", message);
};

ws.onclose = function () {
    console.log("Disconnected");
};

ws.onerror = function (error) {
    console.error("WebSocket error:", error);
};
```

`frond.js` gives you auto-reconnect, message buffering during reconnection, and a cleaner event API. Native WebSocket does not.

---

11. A Complete Chat Page

Here is a full chat page using templates and WebSocket:

Create `src/templates/chat.html`:

```
{% extends "base.html" %}

{% block title %}Chat - {{ room }}{% endblock %}

{% block content %}
    <h1>Chat Room: {{ room }}</h1>
    <p id="status">Connecting...</p>
    <p id="online">Online: 0</p>

    <div id="messages" style="border: 1px solid #ddd; height: 400px; overflow-y: scroll; padding
    </div>

    <div id="typing" style="height: 20px; color: #888; font-style: italic; margin-bottom: 4px;">

    <form id="chat-form" style="display: flex; gap: 8px;">
        <input type="text" id="message-input" placeholder="Type a message..."
               style="flex: 1; padding: 8px; border: 1px solid #ddd; border-radius: 4px;">
        <button type="submit" style="padding: 8px 16px; background: #333; color: white; border:
            Send
        </button>
```

```
</form>

<script src="/js/frond.js"></script>
<script>
    const room = "{{ room }}";
    const ws = frond.ws("/ws/livechat/" + room);
    const messagesDiv = document.getElementById("messages");
    const statusEl = document.getElementById("status");
    const onlineEl = document.getElementById("online");
    const typingEl = document.getElementById("typing");

    let username = prompt("Enter your username:") || "Anonymous";

    ws.on("open", function () {
        statusEl.textContent = "Connected";
        ws.send(JSON.stringify({ type: "set-username", username: username }));
    });

    ws.on("message", function (data) {
        const msg = JSON.parse(data);

        if (msg.type === "message") {
            const div = document.createElement("div");
            div.style.marginBottom = "8px";
            div.innerHTML = "<strong>" + msg.from + ":</strong> " + msg.text;
            messagesDiv.appendChild(div);
            messagesDiv.scrollTop = messagesDiv.scrollHeight;
        }

        if (msg.type === "system") {
            const div = document.createElement("div");
            div.style.cssText = "color: #888; font-style: italic; margin-bottom: 8px;";
            div.textContent = msg.message;
            messagesDiv.appendChild(div);
            messagesDiv.scrollTop = messagesDiv.scrollHeight;
        }

        if (msg.type === "typing") {
            typingEl.textContent = msg.username + " is typing...";
            setTimeout(function () { typingEl.textContent = ""; }, 2000);
        }

        if (msg.online !== undefined) {
            onlineEl.textContent = "Online: " + msg.online;
        }
    });

    ws.on("close", function () {
        statusEl.textContent = "Disconnected. Reconnecting...";
    });

    document.getElementById("chat-form").addEventListener("submit", function (e) {
        e.preventDefault();
        const input = document.getElementById("message-input");
        if (input.value.trim()) {
            ws.send(JSON.stringify({ type: "message", text: input.value }));
            input.value = "";
        }
    });
```

```
        document.getElementById("message-input").addEventListener("input", function () {
            ws.send(JSON.stringify({ type: "typing" }));
        });
    </script>
{% endblock %}
```

Create the route to serve the page:

```
from tina4_python.core.router import get, template

@get("/chat/{room}")
async def chat_page(request, response):
    room = request.params["room"]
    return response(template("chat.html", room=room))
```

Visit `http://localhost:7146/chat/general` in two browser tabs. Type in one tab and watch the message appear in the other instantly.

---

12. Exercise: Build a Real-Time Chat Room

Build a WebSocket chat room with the following features:

Requirements

- WebSocket endpoint at `/ws/room/{room_name}` that handles:
- `open`: Send welcome message with connection count
- `message` with type `set-name`: Set the user's display name
- `message` with type `chat`: Broadcast the message to all clients with the sender's name
- `close`: Broadcast that the user left
- HTTP endpoint at `GET /room/{room_name}` that serves an HTML chat page
- The chat page should:
- Prompt for a username on load
- Display messages in real time
- Show system messages (join/leave) in a different style
- Show the count of online users

Test by:

- Open `http://localhost:7146/room/test` in two browser tabs
- Set different usernames in each
- Send messages from both tabs and verify they appear in both
- Close one tab and verify the "user left" message appears in the other

---

13. Solution

Create `src/routes/chat_room.py`:

```
import json
from datetime import datetime, timezone
from tina4_python.core.router import websocket, get, template

room_users = {}

@websocket("/ws/room/{room_name}")
async def room_ws_handler(connection, event, data):
    room = connection.params["room_name"]
    key = f"{room}:{connection.id}"

    if event == "open":
        room_users[key] = "Anonymous"

        await connection.send(json.dumps({
            "type": "system",
            "message": f"Welcome to room: {room}. Set your name with: "
                       '{"type": "set-name", "name": "YourName"}',
            "online": connection.connection_count()
        }))

        await connection.broadcast(json.dumps({
            "type": "system",
            "message": "A new user joined",
            "online": connection.connection_count()
        }), exclude_self=True)

    if event == "message":
        msg = json.loads(data)
        msg_type = msg.get("type", "chat")

        if msg_type == "set-name":
            old_name = room_users.get(key, "Anonymous")
            new_name = msg.get("name", "Anonymous")
            room_users[key] = new_name

            await connection.broadcast(json.dumps({
                "type": "system",
                "message": f"{old_name} changed their name to {new_name}"
            }))

        if msg_type == "chat":
            username = room_users.get(key, "Anonymous")

            await connection.broadcast(json.dumps({
                "type": "chat",
                "from": username,
                "text": msg.get("text", ""),
                "timestamp": datetime.now(timezone.utc).strftime("%H:%M:%S")
            }))

    if event == "close":
        username = room_users.get(key, "Anonymous")
        room_users.pop(key, None)
```

```
        await connection.broadcast(json.dumps({
            "type": "system",
            "message": f"{username} left the room",
            "online": connection.connection_count()
        }))


@get("/room/{room_name}")
async def room_page(request, response):
    room = request.params["room_name"]
    return response(template("room.html", room=room))
```

Create `src/templates/room.html`:

```
{% extends "base.html" %}

{% block title %}Room: {{ room }}{% endblock %}

{% block content %}
    <h1>Room: {{ room }}</h1>
    <p id="online" style="color: #666;">Online: 0</p>

    <div id="messages" style="border: 1px solid #ddd; height: 400px; overflow-y: auto; padding:
    </div>

    <form id="form" style="display: flex; gap: 8px;">
        <input type="text" id="input" placeholder="Type a message..."
               style="flex: 1; padding: 10px; border: 1px solid #ddd; border-radius: 4px; font-s
        <button type="submit"
                style="padding: 10px 20px; background: #1a1a2e; color: white; border: none; bord
            Send
        </button>
    </form>

    <script src="/js/frond.js"></script>
    <script>
        const room = "{{ room }}";
        const username = prompt("Choose a username:") || "Anonymous";
        const ws = frond.ws("/ws/room/" + room);
        const msgs = document.getElementById("messages");

        ws.on("open", function () {
            ws.send(JSON.stringify({ type: "set-name", name: username }));
        });

        ws.on("message", function (raw) {
            const msg = JSON.parse(raw);
            const div = document.createElement("div");
            div.style.marginBottom = "8px";

            if (msg.type === "chat") {
                div.innerHTML = "<strong>" + msg.from + "</strong> <span style='color:#999;font-
            } else if (msg.type === "system") {
                div.style.color = "#888";
                div.style.fontStyle = "italic";
                div.textContent = msg.message;
            }

            msgs.appendChild(div);
```

```
            msgs.scrollTop = msgs.scrollHeight;

            if (msg.online !== undefined) {
                document.getElementById("online").textContent = "Online: " + msg.online;
            }
        });

        document.getElementById("form").addEventListener("submit", function (e) {
            e.preventDefault();
            const input = document.getElementById("input");
            if (input.value.trim()) {
                ws.send(JSON.stringify({ type: "chat", text: input.value }));
                input.value = "";
            }
        });
    </script>
{% endblock %}
```

Open `http://localhost:7146/room/test` in two browser tabs. Set different usernames. Send messages from one tab and watch them appear in both. Close one tab and verify the "left the room" message appears.

---

14. Scaling with a Backplane

When you run a single server instance, `broadcast()` reaches every connected client. But in production you often run multiple instances behind a load balancer. Each instance only knows about its own connections. A message broadcast on instance A never reaches clients connected to instance B.

A backplane solves this. It relays WebSocket messages across all instances using a shared pub/sub channel. Tina4 supports Redis as a backplane out of the box.

Configuration

Set two environment variables in your `.env`:

```
TINA4_WS_BACKPLANE=redis
TINA4_WS_BACKPLANE_URL=redis://localhost:6379
```

When `TINA4_WS_BACKPLANE` is set, every `broadcast()` call publishes the message to Redis. Every instance subscribes to the same channel and forwards the message to its local connections. No code changes required -- your existing WebSocket routes work as before.

Requirements

The Redis backplane requires a Redis client package as an optional dependency:

```
uv add redis
```

If `TINA4_WS_BACKPLANE` is not set (the default), Tina4 broadcasts only to local connections. This is fine for single-instance deployments.

---

15. Gotchas

1. WebSocket Needs a Persistent Server

Problem: WebSocket connections drop immediately or do not work at all.

Cause: You are running behind a traditional WSGI server like Gunicorn with sync workers. WSGI does not support persistent connections. Each request is handled and the connection is closed.

Fix: Use `tina4 serve` which runs Tina4's async server that handles both HTTP and WebSocket. For production, use an ASGI server like uvicorn or hypercorn (Tina4 auto-detects them). You can still use Nginx as a reverse proxy, but it must be configured for WebSocket proxying with `proxy_set_header Upgrade $http_upgrade` and `proxy_set_header Connection "upgrade"`.

2. Messages Are Strings, Not Dicts

Problem: `data` in the message handler is a string, not a Python dict.

Cause: WebSocket transmits raw strings. If the client sends JSON, you receive the JSON string, not a decoded dict.

Fix: Always `json.loads(data)` when you expect JSON messages. Always `json.dumps()` when you send structured data. WebSocket does not know about content types -- it is just bytes.

3. Connection Count Is Per-Path

Problem: `connection.connection_count()` returns a lower number than expected.

Cause: Connection count is scoped to the WebSocket path. Clients on `/ws/chat/room-1` are counted separately from `/ws/chat/room-2`.

Fix: This is by design. Each path is an isolated group. If you need a global connection count across all paths, maintain a counter in a shared variable or use a module-level dict.

4. Broadcasting Does Not Scale Across Servers

Problem: Users connected to different server instances do not see each other's messages.

Cause: `connection.broadcast()` only sends to clients connected to the same server process. With multiple server instances behind a load balancer, each instance has its own set of connections.

Fix: Use a pub/sub backend like Redis to relay messages across server instances. Each server subscribes to a Redis channel, and broadcast messages are published to Redis so all servers receive them.

5. Large Messages Cause Disconnects

Problem: The connection drops when sending a large message.

Cause: WebSocket servers typically have a maximum message size. The default is usually around 1MB. If your message exceeds this, the server closes the connection.

Fix: Keep messages small (under 64KB is a good target). For large data transfers, use HTTP endpoints instead of WebSocket. WebSocket is designed for small, frequent messages -- not bulk data transfer.

6. Memory Leak from Tracking Connected Users

Problem: The server's memory usage grows over time and eventually crashes.

Cause: You store user data in a dict (like `chat_users`) but do not clean it up when users disconnect. If the `close` event handler does not remove the user from the dict, the dict grows indefinitely.

Fix: Always clean up in the `close` handler. Remove disconnected users from tracking dicts: `chat_users.pop(connection.id, None)`. Test by connecting and disconnecting repeatedly to verify memory stays stable.

7. No Authentication on WebSocket Connect

Problem: Anyone can connect to your WebSocket endpoint and see all messages.

Cause: The WebSocket upgrade request does not carry your JWT token in the `Authorization` header (browsers do not support custom headers on WebSocket connections).

Fix: Pass the token as a query parameter: `ws://localhost:7146/ws/chat?token=eyJ...`. In your `open` handler, validate the token and disconnect if invalid. Use a short-lived token specifically for WebSocket connections. Alternatively, authenticate via an HTTP endpoint first, store the session, and check the session cookie during the WebSocket upgrade.

Server-Sent Events (SSE)

1. The Polling Problem

Your monitoring dashboard needs live metrics. CPU usage. Memory. Active connections. The numbers change every few seconds.

The obvious solution: poll. A timer fires every three seconds. The browser sends a GET request. The server queries the database. The server responds. The browser updates the page. Repeat.

That works. It also wastes resources. Nineteen out of twenty polls return the same data. Each one opens a connection, sends headers, waits for a response, and closes the connection. The server does the same work whether the data changed or not.

WebSocket solves this. A persistent, bi-directional connection. The server pushes when data changes. But WebSocket is designed for two-way communication. Chat rooms. Collaborative editing. Live gaming. Your dashboard only needs one direction: server to client.

Server-Sent Events (SSE) is the middle ground. One-way streaming over plain HTTP. The server pushes. The client listens. The browser reconnects automatically if the connection drops. No upgrade handshake. No special protocol. Just HTTP with a twist.

---

2. What SSE Is

HTTP works like this:

```
Client: "Give me /api/metrics"
Server: "Here are the metrics" (connection closes)
Client: "Give me /api/metrics" (new connection, 3 seconds later)
Server: "Here are the metrics" (connection closes)
```

WebSocket works like this:

```
Client: "Upgrade to WebSocket on /ws/metrics"
Server: "Upgrade accepted. Connection open."
Client: "Subscribe to CPU metrics"
Server: "CPU: 42%"
Server: "CPU: 38%"  (pushed at any time)
Client: "Unsubscribe"
```

SSE works like this:

```
Client: "Give me /events (keep the connection open)"
Server: "data: CPU 42%"
Server: "data: CPU 38%"  (pushed, no client request)
Server: "data: CPU 45%"  (keeps flowing)
Client: (just listens)
```

The key differences:

Feature	HTTP Polling	WebSocket	SSE
Direction	Client to server	Both ways	Server to client

Connection	New each time	Persistent	Persistent
Protocol	HTTP	WebSocket (upgrade)	HTTP
Auto-reconnect	No	No (manual)	Yes (built-in)
Binary data	Yes	Yes	No (text only)
Browser support	Universal	Universal	Universal (except IE)
Complexity	Simple	Moderate	Simple

SSE uses plain HTTP. No protocol upgrade. No special server support. The server sets `Content-Type: text/event-stream` and keeps the connection open. The browser's `EventSource` API handles reconnection, event parsing, and last-event-ID tracking.

---

3. Your First Stream

Create `src/routes/events.py`:

```
import asyncio
from tina4_python import get

@get("/events")
async def stream_events(request, response):
    async def generate():
        for i in range(10):
            yield f"data: {{\"count\": {i}, \"message\": \"tick\"}}\n\n"
            await asyncio.sleep(1)
        yield "data: {\"done\": true}\n\n"

    return response.stream(generate())
```

Three things happen here:

- `generate()` is an async generator. Each `yield` produces one SSE message.
- `response.stream()` tells Tina4 to keep the connection open and send each chunk as it arrives.
- The framework sets `Content-Type: text/event-stream`, `Cache-Control: no-cache`, and `Connection: keep-alive` for you.

Start the server and test with curl:

```
curl -N http://localhost:7146/events
```

You see one message per second:

```
data: {"count": 0, "message": "tick"}

data: {"count": 1, "message": "tick"}

data: {"count": 2, "message": "tick"}
```

Each message arrives the moment the server yields it. No buffering. No waiting for the full response.

---

4. The SSE Protocol

SSE messages are plain text with a simple format. Each message is one or more field lines followed by a blank line.

The `data:` Field

The most common field. Contains the message payload.

```
data: Hello, world
```

Multiple `data:` lines in one message get joined with newlines:

```
data: line one
data: line two
```

The client receives this as `"line one\nline two"`.

The `event:` Field

Names the event type. Without it, the browser fires `onmessage`. With it, the browser fires a named event listener.

```
yield "event: metrics\ndata: {\"cpu\": 42}\n\n"
yield "event: alert\ndata: {\"level\": \"high\", \"message\": \"CPU spike\"}\n\n"
```

On the client:

```
source.addEventListener("metrics", (e) => {
    const data = JSON.parse(e.data);
    updateDashboard(data);
});

source.addEventListener("alert", (e) => {
    const data = JSON.parse(e.data);
    showAlert(data.message);
});
```

Named events let you multiplex different data types on a single connection.

The `id:` Field

Sets the last event ID. If the connection drops, the browser sends `Last-Event-ID` in the reconnection request. Your server can resume from where it left off.

```
yield f"id: {i}\ndata: {{\"count\": {i}}}\n\n"
```

The `retry:` Field

Tells the browser how many milliseconds to wait before reconnecting after a disconnect.

```
yield "retry: 5000\n\n"  # reconnect after 5 seconds
```

The Delimiter

Every SSE message ends with two newlines: `\n\n`. This tells the browser that the message is complete. A single `\n` separates fields within one message.

```
# One complete SSE message:
yield "event: update\nid: 42\ndata: {\"value\": 100}\n\n"

# Broken down:
# event: update\n     ← event type
# id: 42\n            ← event ID
# data: {"value": 100}\n  ← payload
# \n                   ← end of message (blank line)
```

---

5. Frontend: EventSource

The browser provides `EventSource` for consuming SSE streams. It handles connection management, reconnection, and message parsing.

Basic Usage

```
const source = new EventSource("/events");

source.onmessage = function (event) {
    const data = JSON.parse(event.data);
    console.log("Received:", data);
};

source.onerror = function () {
    console.log("Connection lost. Reconnecting...");
};

source.onopen = function () {
    console.log("Connected");
};
```

`EventSource` reconnects automatically when the connection drops. The `onerror` handler fires, the browser waits (default 3 seconds), then reconnects. Your server receives a new request with the `Last-Event-ID` header if you sent event IDs.

Named Events

```
const source = new EventSource("/events");

source.addEventListener("metrics", function (event) {
    updateChart(JSON.parse(event.data));
});

source.addEventListener("notification", function (event) {
    showToast(JSON.parse(event.data));
});

source.addEventListener("heartbeat", function (event) {
    // Connection is alive
});
```

Closing the Connection

```
source.close();
```

The browser stops reconnecting. The server receives a client disconnect.

With Authentication

`EventSource` does not support custom headers. Pass tokens as query parameters:

```
const token = getAuthToken();
const source = new EventSource(`/events?token=${token}`);
```

On the server:

```
@get("/events")
async def secure_events(request, response):
    token = request.query.get("token")
    payload = Auth.valid_token(token)
    if not payload:
        return response({"error": "Unauthorized"}, 401)

    async def generate():
        while True:
            yield f"data: {{\"user\": \"{payload['email']}\"}}\n\n"
            await asyncio.sleep(5)

    return response.stream(generate())
```

---

6. Real Examples

Live Dashboard

Stream database metrics every five seconds:

```
import asyncio
from tina4_python import get
from tina4_python.database import Database

@get("/events/dashboard")
async def dashboard_stream(request, response):
    async def generate():
        db = Database()
        while True:
            orders = db.fetch_one("SELECT count(*) as total FROM orders")
            revenue = db.fetch_one("SELECT sum(total) as revenue FROM orders WHERE date(created_

            yield f"data: {{\"orders\": {orders['total']}, \"revenue\": {revenue['revenue'] or 0
            await asyncio.sleep(5)

    return response.stream(generate())
```

The client updates the dashboard numbers without polling. One connection. One query every five seconds. No wasted requests.

Build Progress

Stream deployment status as it happens:

```
import asyncio
import json
from tina4_python import get

@get("/events/deploy/{build_id}")
async def deploy_stream(request, response):
    build_id = request.params["build_id"]

    async def generate():
        steps = [
            ("pull", "Pulling latest code..."),
            ("install", "Installing dependencies..."),
            ("test", "Running tests..."),
            ("build", "Building application..."),
            ("deploy", "Deploying to production..."),
            ("done", "Deployment complete"),
        ]

        for step, message in steps:
            yield f"event: progress\ndata: {json.dumps({'step': step, 'message': message, 'build
            await asyncio.sleep(2)  # simulate work

    return response.stream(generate())
```

The browser shows a progress bar that updates in real time. Each step arrives as it completes. The user watches the deployment unfold instead of staring at a spinner.

LLM Chat Streaming

Stream AI responses token by token. This is how ChatGPT works:

```
import asyncio
import json
import urllib.request
from tina4_python import get

@get("/events/chat")
async def chat_stream(request, response):
    prompt = request.query.get("q", "Hello")

    async def generate():
        # Call the LLM API with streaming enabled
        req_data = json.dumps({
            "model": "claude-sonnet-4-20250514",
            "max_tokens": 512,
            "stream": True,
            "messages": [{"role": "user", "content": prompt}],
        }).encode()

        req = urllib.request.Request(
            "https://api.anthropic.com/v1/messages",
            data=req_data,
            headers={
                "Content-Type": "application/json",
                "x-api-key": os.environ["ANTHROPIC_API_KEY"],
```

```
                "anthropic-version": "2023-06-01",
            },
        )

        with urllib.request.urlopen(req) as resp:
            for line in resp:
                line = line.decode().strip()
                if line.startswith("data: "):
                    chunk = json.loads(line[6:])
                    if chunk.get("type") == "content_block_delta":
                        text = chunk["delta"].get("text", "")
                        yield f"data: {json.dumps({'token': text})}\n\n"
                        await asyncio.sleep(0)

        yield "data: {\"done\": true}\n\n"

    return response.stream(generate())
```

The frontend appends each token as it arrives:

```
const source = new EventSource(`/events/chat?q=${encodeURIComponent(question)}`);
const output = document.getElementById("response");

source.onmessage = function (event) {
    const data = JSON.parse(event.data);
    if (data.done) {
        source.close();
        return;
    }
    output.textContent += data.token;
};
```

The response builds character by character. The user reads as the AI writes. No waiting for the full answer.

File Processing Progress

Upload a CSV, stream the processing status:

```
import asyncio
import json
from tina4_python import post

@post("/api/import")
async def import_csv(request, response):
    file = request.files.get("csv")
    if not file:
        return response({"error": "No file"}, 400)

    lines = file["content"].decode().strip().split("\n")
    total = len(lines) - 1  # minus header

    async def generate():
        for i, line in enumerate(lines[1:], 1):
            # Process each row
            cols = line.split(",")
            # ... insert into database ...
            yield f"data: {json.dumps({'processed': i, 'total': total, 'percent': round(i/total*
            await asyncio.sleep(0)  # yield control
```

```
        yield f"data: {json.dumps({'done': True, 'total': total})}\n\n"

    return response.stream(generate(), content_type="text/event-stream")
```

The browser shows a progress bar that fills as each row is processed. The user sees exactly where the import stands.

---

7. Custom Content Types

SSE defaults to `text/event-stream`, but `response.stream()` accepts any content type. Use this for newline-delimited JSON (NDJSON), chunked binary, or custom protocols.

NDJSON Streaming

```
import asyncio
import json
from tina4_python import get

@get("/api/export")
async def export_stream(request, response):
    async def generate():
        db = Database()
        result = db.fetch("SELECT * FROM products", limit=1000)
        for row in result.records:
            yield json.dumps(row) + "\n"
            await asyncio.sleep(0)

    return response.stream(generate(), content_type="application/x-ndjson")
```

Each line is a complete JSON object. The client reads line by line. No need to parse a giant array. Memory stays flat regardless of how many rows you export.

Consuming NDJSON on the Client

```
async function streamProducts() {
    const response = await fetch("/api/export");
    const reader = response.body.getReader();
    const decoder = new TextDecoder();
    let buffer = "";

    while (true) {
        const { done, value } = await reader.read();
        if (done) break;

        buffer += decoder.decode(value, { stream: true });
        const lines = buffer.split("\n");
        buffer = lines.pop(); // keep incomplete line

        for (const line of lines) {
            if (line.trim()) {
                const product = JSON.parse(line);
                addToTable(product);
            }
        }
    }
}
```

---

8. SSE vs WebSocket

Both keep a persistent connection. Both push data in real time. The choice depends on the direction of communication.

Use Case	Choose	Why
Live dashboard	SSE	Server pushes metrics. Client only listens.
Chat application	WebSocket	Both sides send messages.
Notification feed	SSE	Server pushes. Client marks as read via separate HTTP POST.
Collaborative editor	WebSocket	Both sides send cursor positions and edits.
Build/deploy progress	SSE	Server streams status. Client watches.
LLM token streaming	SSE	Server streams tokens. Client displays them.
Online gaming	WebSocket	Low-latency, bi-directional input and state.
Stock ticker	SSE	Server pushes prices. Client displays.
File upload progress	HTTP (native)	Browser tracks upload. Server tracks processing via SSE.

Rule of thumb: If the client only listens, use SSE. If the client sends data back on the same connection, use WebSocket.

SSE has one advantage WebSocket does not: automatic reconnection. The browser's `EventSource` reconnects without any code. WebSocket requires manual reconnection logic (or `frond.js` which handles it for you).

---

9. Production Considerations

Nginx Proxy Buffering

Nginx buffers responses by default. SSE messages accumulate in the buffer and arrive in batches instead of one at a time. Tina4 sets `X-Accel-Buffering: no` on streaming responses, which tells Nginx to disable buffering for that response.

If you configure Nginx manually:

```
location /events/ {
    proxy_pass http://localhost:7146;
    proxy_buffering off;
    proxy_cache off;
    proxy_set_header Connection '';
    proxy_http_version 1.1;
    chunked_transfer_encoding off;
}
```

Connection Limits

Each SSE connection is a persistent HTTP connection. Most servers limit concurrent connections. The default asyncio server handles thousands. In production:

- Monitor open connections
- Set timeouts on long-running streams
- Close streams when the client no longer needs them

```
@get("/events/metrics")
async def metrics_with_timeout(request, response):
    async def generate():
        for _ in range(3600):  # max 1 hour (one message per second)
            yield f"data: {{\"cpu\": {get_cpu()}}}\n\n"
            await asyncio.sleep(1)
        yield "event: timeout\ndata: {\"message\": \"Stream expired. Reconnect.\"}\n\n"

    return response.stream(generate())
```

Heartbeat Messages

Some proxies and load balancers close idle connections after a timeout (typically 30-60 seconds). Send a heartbeat comment to keep the connection alive:

```
async def generate():
    counter = 0
    while True:
        if has_new_data():
            yield f"data: {json.dumps(get_data())}\n\n"
        else:
            yield ": heartbeat\n\n"  # SSE comment (colon prefix) — ignored by EventSource
        counter += 1
        await asyncio.sleep(5)
```

Lines starting with `:` are SSE comments. The browser ignores them, but they keep the connection alive through proxies.

---

10. Exercise: Build a Live Metrics Dashboard

Build a dashboard that streams server metrics to the browser.

Requirements

- SSE endpoint at `GET /events/server-metrics` that streams every 3 seconds:
- Current timestamp
- A random CPU percentage (simulate with `random.randint(10, 90)`)
- A random memory percentage
- A random request count
- HTML page at `GET /dashboard` that:
- Connects to the SSE endpoint
- Displays the four metrics in cards
- Updates the values in real time (no page refresh)
- Shows a "Connected" / "Reconnecting..." status indicator

Test by:

- Open `http://localhost:7146/dashboard`
- Watch the numbers update every 3 seconds
- Stop the server and verify "Reconnecting..." appears
- Restart the server and verify it reconnects and resumes updating

---

11. Solution

Create `src/routes/server_metrics.py`:

```
import asyncio
import json
import random
from datetime import datetime, timezone
from tina4_python import get

@get("/events/server-metrics")
async def server_metrics_stream(request, response):
    async def generate():
        yield "retry: 3000\n\n"
        counter = 0
        while True:
            yield f"id: {counter}\ndata: {json.dumps({
                'timestamp': datetime.now(timezone.utc).isoformat(),
                'cpu': random.randint(10, 90),
                'memory': random.randint(30, 85),
                'requests': random.randint(100, 5000),
            })}\n\n"
            counter += 1
```

```
            await asyncio.sleep(3)

    return response.stream(generate())


@get("/dashboard")
async def dashboard_page(request, response):
    html = """
    <!DOCTYPE html>
    <html>
    <head><title>Live Dashboard</title></head>
    <body style="font-family: system-ui; max-width: 600px; margin: 40px auto; padding: 0 20px;">
        <h1>Server Metrics</h1>
        <p id="status" style="color: #888;">Connecting...</p>
        <div style="display: grid; grid-template-columns: 1fr 1fr; gap: 16px; margin-top: 20px;"
            <div style="background: #f5f5f5; padding: 20px; border-radius: 8px;">
                <div style="color: #888; font-size: 14px;">CPU</div>
                <div id="cpu" style="font-size: 32px; font-weight: 700;">--</div>
            </div>
            <div style="background: #f5f5f5; padding: 20px; border-radius: 8px;">
                <div style="color: #888; font-size: 14px;">Memory</div>
                <div id="memory" style="font-size: 32px; font-weight: 700;">--</div>
            </div>
            <div style="background: #f5f5f5; padding: 20px; border-radius: 8px;">
                <div style="color: #888; font-size: 14px;">Requests/min</div>
                <div id="requests" style="font-size: 32px; font-weight: 700;">--</div>
            </div>
            <div style="background: #f5f5f5; padding: 20px; border-radius: 8px;">
                <div style="color: #888; font-size: 14px;">Last Update</div>
                <div id="time" style="font-size: 16px; font-weight: 500;">--</div>
            </div>
        </div>
        <script>
            const source = new EventSource("/events/server-metrics");
            const status = document.getElementById("status");

            source.onopen = function () {
                status.textContent = "Connected";
                status.style.color = "#22c55e";
            };

            source.onmessage = function (event) {
                const data = JSON.parse(event.data);
                document.getElementById("cpu").textContent = data.cpu + "%";
                document.getElementById("memory").textContent = data.memory + "%";
                document.getElementById("requests").textContent = data.requests.toLocaleString()
                document.getElementById("time").textContent = new Date(data.timestamp).toLocaleT
            };

            source.onerror = function () {
                status.textContent = "Reconnecting...";
                status.style.color = "#ef4444";
            };
        </script>
    </body>
    </html>
    """
    return response(html)
```

Open `http://localhost:7146/dashboard`. The numbers update every three seconds. Stop the server. The status turns red: "Reconnecting..." Start it again. The status turns green. The numbers resume.

No polling. No WebSocket. No JavaScript timers. The browser handles everything.

---

12. What You Learned

- **SSE streams data from server to client** over a single HTTP connection. No upgrade. No special protocol.
- `response.stream(generator)` keeps the connection open and flushes each chunk as the generator yields it.
- **The SSE protocol** uses `data:`, `event:`, `id:`, and `retry:` fields. Messages end with `\n\n`.
- `EventSource` on the client handles connection, parsing, and automatic reconnection.
- **Named events** let you multiplex different data types on one stream.
- **Custom content types** let you stream NDJSON, binary, or any format.
- **SSE is for one-way server pushes.** Use WebSocket when the client needs to send data back on the same connection.
- **Heartbeat comments** keep connections alive through proxies. Tina4 sets `X-Accel-Buffering: no` to disable nginx buffering.

One direction. One connection. Real-time data. The server speaks. The client listens. The rest is infrastructure.

WSDL / SOAP

1. SOAP Is Not Dead

Banks. Governments. Insurance companies. Logistics providers. Decades of enterprise software runs on SOAP. If you integrate with any of them, you speak SOAP. Tina4 makes it bearable.

The `WSDL` class exposes your Python functions as SOAP 1.1 operations. It auto-generates the WSDL document at `?wsdl`. You write normal Python functions with type annotations. Tina4 handles the XML envelope, the SOAP headers, and the schema generation.

---

2. Your First SOAP Service

```
from tina4_python.wsdl import WSDL, wsdl_operation

wsdl = WSDL(
    service_name="CalculatorService",
    namespace="http://example.com/calculator",
    endpoint="/api/soap/calculator"
)

@wsdl_operation(wsdl, name="Add", description="Add two integers")
def add(a: int, b: int) -> int:
    return a + b

@wsdl_operation(wsdl, name="Multiply", description="Multiply two integers")
def multiply(a: int, b: int) -> int:
    return a * b
```

Register the service with the router:

```
from tina4_python.core.router import get, post

@get("/api/soap/calculator")
async def calculator_wsdl(request, response):
    return wsdl.handle_wsdl(request, response)

@post("/api/soap/calculator")
async def calculator_soap(request, response):
    return await wsdl.handle_request(request, response)
```

Visit `http://localhost:7146/api/soap/calculator?wsdl` to see the generated WSDL document. POST a SOAP envelope to the same URL to invoke an operation.

---

3. The @wsdl_operation Decorator

`@wsdl_operation` registers a function as a SOAP operation on the given `WSDL` instance.

Parameter	Type	Description

`wsdl`	`WSDL`	The WSDL instance to register on
`name`	`str`	Operation name as it appears in the WSDL
`description`	`str`	Optional human-readable description

Type annotations on the function parameters become the SOAP message schema. Supported types: `str`, `int`, `float`, `bool`.

```
@wsdl_operation(wsdl, name="GetProductPrice")
def get_product_price(product_id: str, currency: str) -> float:
    prices = {"USD": 79.99, "EUR": 73.99, "GBP": 62.99}
    return prices.get(currency, prices["USD"])
```

---

4. Auto WSDL Generation at ?wsdl

Tina4 generates the WSDL document from your type annotations. When a SOAP client sends a GET request with the `?wsdl` query string, the handler returns the XML document automatically.

```
curl "http://localhost:7146/api/soap/calculator?wsdl"
```

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="CalculatorService"
  targetNamespace="http://example.com/calculator"
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:tns="http://example.com/calculator"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">

  <types>
    <xsd:schema targetNamespace="http://example.com/calculator">
      <xsd:element name="AddRequest">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="a" type="xsd:int"/>
            <xsd:element name="b" type="xsd:int"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
      <xsd:element name="AddResponse">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="result" type="xsd:int"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
      <!-- Multiply types... -->
    </xsd:schema>
  </types>

  <!-- Bindings, ports, and service definition... -->
</definitions>
```

Most SOAP clients (SOAPUI, Java's JAX-WS, .NET's `wsdl.exe`) can import this URL directly and generate client stubs.

---

5. Calling the Service

A raw SOAP request with curl:

```
curl -X POST http://localhost:7146/api/soap/calculator \
  -H "Content-Type: text/xml; charset=utf-8" \
  -H "SOAPAction: \"Add\"" \
  -d '<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
               xmlns:tns="http://example.com/calculator">
  <soap:Body>
    <tns:AddRequest>
      <tns:a>15</tns:a>
      <tns:b>27</tns:b>
    </tns:AddRequest>
  </soap:Body>
</soap:Envelope>'
```

Response:

```
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <AddResponse xmlns="http://example.com/calculator">
      <result>42</result>
    </AddResponse>
  </soap:Body>
</soap:Envelope>
```

---

6. Lifecycle Hooks: on*request and on*result

Inspect or modify the SOAP request before the operation runs, and modify the response before it is serialised.

```
from tina4_python.wsdl import WSDL, wsdl_operation
from tina4_python.debug import Log

wsdl = WSDL(
    service_name="OrderService",
    namespace="http://example.com/orders",
    endpoint="/api/soap/orders"
)

def on_request(operation_name, params):
    Log.info("SOAP request received", operation=operation_name, params=params)
    # Return modified params or None to pass through unchanged
    return params

def on_result(operation_name, result):
    Log.info("SOAP operation completed", operation=operation_name, result=result)
    # Return modified result or None to pass through unchanged
```

```
    return result

wsdl.on_request = on_request
wsdl.on_result = on_result
```

Use `on_request` for:

- Authentication / API key validation from SOAP headers
- Input sanitisation
- Request logging and audit trails

Use `on_result` for:

- Response transformation
- Stripping internal fields before returning
- Result logging

---

7. Complex Types with Dataclasses

For operations that accept or return structured data, use Python dataclasses:

```
from dataclasses import dataclass
from tina4_python.wsdl import WSDL, wsdl_operation

wsdl = WSDL(
    service_name="ProductService",
    namespace="http://example.com/products",
    endpoint="/api/soap/products"
)

@dataclass
class Product:
    id: str
    name: str
    price: float
    in_stock: bool

@wsdl_operation(wsdl, name="GetProduct")
def get_product(product_id: str) -> Product:
    catalog = {
        "KB-001": Product(id="KB-001", name="Wireless Keyboard", price=79.99, in_stock=True),
        "HUB-002": Product(id="HUB-002", name="USB-C Hub", price=49.99, in_stock=False)
    }
    result = catalog.get(product_id)
    if result is None:
        raise ValueError(f"Product {product_id} not found")
    return result
```

Tina4 introspects the dataclass fields and generates the corresponding WSDL complex types automatically.

---

8. Error Handling

Raise a Python exception to return a SOAP fault:

```
@wsdl_operation(wsdl, name="CreateOrder")
def create_order(customer_id: str, product_id: str, quantity: int) -> str:
    if quantity <= 0:
        raise ValueError("Quantity must be greater than zero")

    if not check_stock(product_id, quantity):
        raise RuntimeError(f"Insufficient stock for {product_id}")

    order_id = f"ORD-{customer_id}-{product_id}"
    return order_id
```

Any raised exception becomes a SOAP fault response:

```
<soap:Body>
  <soap:Fault>
    <faultcode>soap:Server</faultcode>
    <faultstring>Quantity must be greater than zero</faultstring>
  </soap:Fault>
</soap:Body>
```

---

9. Exercise: Build a Currency Conversion SOAP Service

Create a SOAP service that converts amounts between currencies.

Requirements

- Create a `WSDL` instance for a `CurrencyService` at `/api/soap/currency`
- Implement two operations:
- `Convert(amount: float, from_currency: str, to_currency: str) -> float`
- `GetRate(from_currency: str, to_currency: str) -> float`
- Add `on_request` logging
- Raise `ValueError` for unknown currency codes

Test with:

```
# Get the WSDL
curl "http://localhost:7146/api/soap/currency?wsdl"

# Convert $100 USD to EUR
curl -X POST http://localhost:7146/api/soap/currency \
  -H "Content-Type: text/xml" \
  -H "SOAPAction: \"Convert\"" \
  -d '<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
                     xmlns:tns="http://example.com/currency">
    <soap:Body>
      <tns:ConvertRequest>
        <tns:amount>100.0</tns:amount>
        <tns:from_currency>USD</tns:from_currency>
        <tns:to_currency>EUR</tns:to_currency>
      </tns:ConvertRequest>
    </soap:Body>
  </soap:Envelope>'
```

---

10. Solution

Create `src/routes/currency_soap.py`:

```
from tina4_python.core.router import get, post
from tina4_python.wsdl import WSDL, wsdl_operation
from tina4_python.debug import Log

RATES = {
    ("USD", "EUR"): 0.92,
    ("USD", "GBP"): 0.79,
    ("USD", "JPY"): 149.50,
    ("EUR", "USD"): 1.09,
    ("EUR", "GBP"): 0.86,
    ("GBP", "USD"): 1.27,
    ("GBP", "EUR"): 1.17,
}
SUPPORTED = {"USD", "EUR", "GBP", "JPY"}

wsdl = WSDL(
    service_name="CurrencyService",
    namespace="http://example.com/currency",
    endpoint="/api/soap/currency"
)

def on_request(operation_name, params):
    Log.info("SOAP currency request", operation=operation_name, params=str(params))
    return params

wsdl.on_request = on_request


@wsdl_operation(wsdl, name="GetRate", description="Get exchange rate between two currencies")
def get_rate(from_currency: str, to_currency: str) -> float:
    from_currency = from_currency.upper()
    to_currency = to_currency.upper()
```

```
    if from_currency not in SUPPORTED:
        raise ValueError(f"Unknown currency: {from_currency}")
    if to_currency not in SUPPORTED:
        raise ValueError(f"Unknown currency: {to_currency}")
    if from_currency == to_currency:
        return 1.0

    rate = RATES.get((from_currency, to_currency))
    if rate is None:
        raise ValueError(f"No rate available for {from_currency} -> {to_currency}")
    return rate


@wsdl_operation(wsdl, name="Convert", description="Convert an amount between currencies")
def convert(amount: float, from_currency: str, to_currency: str) -> float:
    rate = get_rate(from_currency, to_currency)
    return round(amount * rate, 2)


@get("/api/soap/currency")
async def currency_wsdl(request, response):
    return wsdl.handle_wsdl(request, response)


@post("/api/soap/currency")
async def currency_soap(request, response):
    return await wsdl.handle_request(request, response)
```

---

11. Gotchas

1. Missing SOAPAction header

Problem: The SOAP request returns a fault saying the operation was not found.

Fix: Include the `SOAPAction` header in every POST request. Its value must match the operation name, wrapped in quotes: `SOAPAction: "Add"`.

2. Namespace mismatch

Problem: The WSDL is generated with namespace `http://example.com/calculator` but the request body uses a different namespace. The operation is not matched.

Fix: Copy the `targetNamespace` from the WSDL document exactly into the request body's `xmlns:tns` attribute. SOAP is namespace-strict.

3. Type annotations missing

Problem: A parameter with no type annotation causes an error during WSDL generation.

Fix: All parameters and return types must be annotated. Supported types: `str`, `int`, `float`, `bool`, and dataclasses whose fields use those types.

4. Forgetting the ?wsdl route

Problem: The SOAP client cannot import the WSDL because the GET handler is missing.

Fix: Register both a `@get` handler (for `?wsdl`) and a `@post` handler (for SOAP operations) on the same path.

DI Container

1. Stop Constructing Dependencies Inside Your Code

```
@post("/api/orders")
async def create_order(request, response):
    db = Database.get_connection()      # tight coupling
    mailer = Messenger()                # constructed inline
    payments = PaymentGateway(          # hardcoded config
        api_key=os.environ["PAYMENT_KEY"]
    )
    ...
```

Every route constructs its own dependencies. Testing requires monkey-patching or mocking at the module level. Swapping the payment gateway means touching every route that uses it.

A DI container moves construction out of the route. You register services once, at startup. Routes ask the container for what they need by name.

---

2. The Container Class

```
from tina4_python.container import Container

container = Container()
```

The `Container` is a lightweight registry. It does not scan files, read XML, or require annotations on your classes. You register explicitly. You retrieve explicitly.

---

3. register() — Transient Services

A transient service creates a new instance every time `get()` is called:

```
from tina4_python.container import Container

container = Container()

# Register a factory function
container.register("logger", lambda: Logger(level="INFO"))

# Each call creates a fresh instance
logger1 = container.get("logger")
logger2 = container.get("logger")

assert logger1 is not logger2  # Different objects
```

Use transient registration for services that must not share state: request-scoped objects, disposable HTTP clients, or mock implementations in tests.

---

4. singleton() — Cached Services

A singleton creates the instance once and returns the same object on every subsequent call:

```
import os
from tina4_python.container import Container
from tina4_python.api import Api

container = Container()

# Register a singleton factory
container.singleton("payment_gateway", lambda: Api(
    bearer_token=os.environ["PAYMENT_API_KEY"],
    timeout=15
))

# Both calls return the exact same object
gw1 = container.get("payment_gateway")
gw2 = container.get("payment_gateway")

assert gw1 is gw2  # Same object
```

Use singleton registration for: database connections, HTTP clients, configuration objects, external service adapters.

---

5. get() and has()

```
# Retrieve a service
service = container.get("payment_gateway")

# Check before retrieving
if container.has("email_service"):
    mailer = container.get("email_service")
    mailer.send(...)
```

`get()` raises `KeyError` if the service is not registered. `has()` lets you check first, or use it to provide a default:

```
mailer = container.get("email_service") if container.has("email_service") else None
```

---

6. reset()

Clear the singleton cache without removing registrations. Useful in tests to force fresh construction:

```
container.singleton("db", lambda: Database.connect())

# In tests: reset between test cases so each gets a clean DB
container.reset()

db = container.get("db")  # New connection created
```

`reset()` clears all cached singletons. Transient registrations are unaffected (they never cache). Call `reset()` in test teardown to prevent state from leaking between tests.

---

7. Building a Service Container for Your App

Define your container in one place:

```
# src/services.py
import os
from tina4_python.container import Container
from tina4_python.api import Api

container = Container()

# Database connection (singleton -- one connection reused)
container.singleton("db", lambda: Database.get_connection())

# Payment gateway (singleton -- one client, shared config)
container.singleton("payments", lambda: Api(
    bearer_token=os.environ["PAYMENT_API_KEY"],
    timeout=15
))

# Email service (singleton)
container.singleton("mailer", lambda: Api(
    bearer_token=os.environ["SENDGRID_API_KEY"]
))

# Request logger (transient -- new instance per use)
container.register("request_logger", lambda: RequestLogger(
    level=os.environ.get("LOG_LEVEL", "INFO")
))
```

Import `container` wherever you need it:

```
# src/routes/orders.py
from src.services import container

@post("/api/orders")
async def create_order(request, response):
    payments = container.get("payments")
    mailer = container.get("mailer")
    ...
```

---

8. Testing with the Container

Swap implementations for tests without changing any route code:

```
# tests/test_orders.py
import pytest
from src.services import container

class MockPaymentGateway:
    def post(self, url, body):
```

```
        return {
            "http_code": 200,
            "body": {"charge_id": "ch_test_001", "status": "succeeded"},
            "headers": {},
            "error": None
        }

class MockMailer:
    def __init__(self):
        self.sent = []

    def post(self, url, body):
        self.sent.append(body)
        return {"http_code": 200, "body": {"id": "msg_001"}, "headers": {}, "error": None}


@pytest.fixture(autouse=True)
def reset_container():
    # Clear singleton cache before each test
    container.reset()
    yield
    container.reset()


def test_create_order_success(client):
    mock_mailer = MockMailer()

    # Override with test doubles
    container.singleton("payments", lambda: MockPaymentGateway())
    container.singleton("mailer", lambda: mock_mailer)

    resp = client.post("/api/orders", json={
        "customer_id": 1,
        "items": [{"product_id": "KB-001", "quantity": 1, "price": 79.99}],
        "email": "alice@example.com"
    })

    assert resp.status_code == 201
    assert resp.json()["order_id"] is not None
    assert len(mock_mailer.sent) == 1  # Confirmation email was sent
```

The route code never changes. The container is the only seam.

---

9. Full Example: Order Route with DI

```
# src/routes/orders_di.py
from tina4_python.core.router import post
from tina4_python.debug import Log
from src.services import container


@post("/api/di/orders")
async def create_order_di(request, response):
    body = request.body

    if not body.get("customer_id") or not body.get("items"):
        return response({"error": "customer_id and items are required"}, 400)

    payments = container.get("payments")
    mailer = container.get("mailer")

    total = round(sum(item.get("price", 0) * item.get("quantity", 1) for item in body["items"]),

    # Charge payment
    charge_result = payments.post("https://api.payment-provider.com/v1/charges", {
        "amount": int(total * 100),
        "currency": "usd"
    })

    if charge_result["error"] or charge_result["http_code"] not in (200, 201):
        Log.error("Payment failed", customer_id=body["customer_id"], total=total)
        return response({"error": "Payment failed"}, 402)

    charge_id = charge_result["body"]["charge_id"]
    order_id = f"ORD-{body['customer_id']}-{charge_id}"

    # Send confirmation
    mailer.post("https://api.sendgrid.com/v3/mail/send", {
        "to": body.get("email"),
        "subject": f"Order Confirmation {order_id}",
        "body": f"Your order of ${total} has been confirmed."
    })

    Log.info("Order created", order_id=order_id, total=total)

    return response({
        "order_id": order_id,
        "total": total,
        "charge_id": charge_id
    }, 201)
```

---

10. Exercise: Configurable Notification Service

Build a notification service that can switch between email and SMS providers via the container.

Requirements

- Define two notifier classes: `EmailNotifier` and `SmsNotifier`, each with a `send(to, message)` method

- Register the active notifier as `"notifier"` in the container (controlled by a `NOTIFIER` env var)
- Create `POST /api/notify` that retrieves `"notifier"` from the container and sends a message
- Write a test that registers a mock notifier and verifies messages are sent

Test with:

```
# Set NOTIFIER=email or NOTIFIER=sms in .env
curl -X POST http://localhost:7146/api/notify \
  -H "Content-Type: application/json" \
  -d '{"to": "alice@example.com", "message": "Your order has shipped!"}'
```

---

11. Solution

```
# src/notifiers.py
import os

class EmailNotifier:
    def send(self, to, message):
        print(f"[EMAIL] To: {to} | Message: {message}")
        return {"channel": "email", "to": to}

class SmsNotifier:
    def send(self, to, message):
        print(f"[SMS] To: {to} | Message: {message}")
        return {"channel": "sms", "to": to}

def make_notifier():
    channel = os.environ.get("NOTIFIER", "email")
    if channel == "sms":
        return SmsNotifier()
    return EmailNotifier()
```

```
# src/services.py (additions)
from src.notifiers import make_notifier
container.singleton("notifier", make_notifier)
```

```
# src/routes/notify.py
from tina4_python.core.router import post
from src.services import container

@post("/api/notify")
async def notify(request, response):
    body = request.body

    if not body.get("to") or not body.get("message"):
        return response({"error": "to and message are required"}, 400)

    notifier = container.get("notifier")
    result = notifier.send(body["to"], body["message"])

    return response({"sent": True, **result})
```

```
# tests/test_notify.py
from src.services import container
```

```
class MockNotifier:
    def __init__(self):
        self.messages = []
    def send(self, to, message):
        self.messages.append({"to": to, "message": message})
        return {"channel": "mock", "to": to}

def test_notify_sends_message(client):
    mock = MockNotifier()
    container.reset()
    container.singleton("notifier", lambda: mock)

    resp = client.post("/api/notify", json={"to": "bob@example.com", "message": "Hello!"})

    assert resp.status_code == 200
    assert mock.messages[0]["to"] == "bob@example.com"
    container.reset()
```

---

12. Gotchas

1. Mutable singleton shared across requests

Problem: A singleton holds per-request state and leaks it between concurrent requests.

Fix: Use `register()` (transient) for anything that holds request-specific state. Singletons should be stateless or thread-safe.

2. reset() removes singleton cache but not registrations

Problem: After `container.reset()`, calling `container.has("payments")` still returns `True` but the next `get()` creates a new instance.

Fix: This is correct behaviour. `reset()` only clears the cache. The factory function is still registered and will run again on the next `get()`. Use `reset()` deliberately in test teardown.

3. Circular dependencies

Problem: Service A's factory calls `container.get("B")` and B's factory calls `container.get("A")`. Both hang or raise a recursion error.

Fix: Restructure so that dependencies are one-directional. Extract shared logic into a third service that neither A nor B depends back upon.

4. Importing container before .env is loaded

Problem: `container.singleton("payments", lambda: Api(bearer_token=os.environ["PAYMENT_KEY"]))` raises `KeyError` because `.env` has not been loaded yet.

Fix: Tina4 loads `.env` before importing route files. If you initialise the container in a module that is imported before startup, use a lambda (lazy factory) so `os.environ` is not accessed until `get()` is called.

Service Runner

1. Work That Never Stops

Some tasks run on a schedule. Some run indefinitely. A queue worker that drains emails. A cache warmer that refreshes product data every five minutes. These are background services — long-lived processes that run alongside the web server but are not tied to any HTTP request.

Tina4's service runner gives these tasks a consistent start/stop lifecycle. Register services. Start them. Stop them gracefully. They run in background threads so they do not block the web server.

---

2. A Basic Background Service

A service is any class with a `run()` method. Start it with the runner and it executes in the background:

```
import time
from tina4_python.service import ServiceRunner

class HeartbeatService:
    def __init__(self, interval=30):
        self.interval = interval
        self.running = False

    def run(self):
        self.running = True
        while self.running:
            print(f"Heartbeat at {time.strftime('%H:%M:%S')}")
            time.sleep(self.interval)

    def stop(self):
        self.running = False


runner = ServiceRunner()
runner.register("heartbeat", HeartbeatService(interval=30))
runner.start()
```

`runner.start()` launches all registered services in background threads and returns immediately. The web server keeps handling requests. The heartbeat ticks every 30 seconds.

---

3. Start / Stop Lifecycle

Every service implements `run()` and `stop()`. The runner calls `run()` in a thread when started and `stop()` when shutting down.

```
import threading
import time
from tina4_python.service import ServiceRunner

class PollingService:
```

```
    def __init__(self, poll_url, interval=60):
        self.poll_url = poll_url
        self.interval = interval
        self._running = threading.Event()

    def run(self):
        self._running.set()
        while self._running.is_set():
            try:
                self._poll()
            except Exception as exc:
                print(f"Poll failed: {exc}")

            # Wait for interval or until stop() clears the event
            self._running.wait(timeout=self.interval)

    def stop(self):
        self._running.clear()

    def _poll(self):
        from tina4_python.api import Api
        api = Api(timeout=10)
        result = api.get(self.poll_url)
        if result["http_code"] == 200:
            print(f"Poll ok: {result['body']}")
        else:
            print(f"Poll error: {result['http_code']}")
```

Using `threading.Event` instead of a boolean for the stop signal is safer: `wait(timeout=n)` wakes up immediately when the event is cleared, so `stop()` takes effect within milliseconds rather than waiting for the next sleep to expire.

---

4. Queue Worker Service

The most common background service is a queue worker:

```
import time
from tina4_python.service import ServiceRunner
from tina4_python.queue import Queue
from tina4_python.debug import Log


class EmailWorker:
    def __init__(self):
        self.queue = Queue(topic="emails", max_retries=3)
        self._running = False

    def run(self):
        self._running = True
        Log.info("Email worker started")

        while self._running:
            job = self.queue.pop()

            if job is None:
                # No pending jobs -- wait briefly before checking again
```

```
                time.sleep(1)
                continue

            try:
                self._send_email(job.payload)
                job.complete()
                Log.info("Email sent", to=job.payload["to"])
            except Exception as exc:
                job.fail(str(exc))
                Log.warning("Email failed", to=job.payload.get("to"), error=str(exc))

    def stop(self):
        self._running = False
        Log.info("Email worker stopped")

    def _send_email(self, payload):
        # Replace with real email logic
        import time
        time.sleep(0.1)  # Simulate sending
        if payload.get("to") == "bounce@example.com":
            raise RuntimeError("Recipient address rejected")


# Register and start
runner = ServiceRunner()
runner.register("email_worker", EmailWorker())
runner.start()
```

---

5. Scheduled Task Service

Run a function on a fixed schedule:

```
import time
import threading
from tina4_python.service import ServiceRunner
from tina4_python.debug import Log


class ScheduledTask:
    def __init__(self, name, task_fn, interval_seconds):
        self.name = name
        self.task_fn = task_fn
        self.interval = interval_seconds
        self._stop_event = threading.Event()

    def run(self):
        Log.info("Scheduled task started", task=self.name, interval=self.interval)
        while not self._stop_event.is_set():
            try:
                self.task_fn()
            except Exception as exc:
                Log.error("Scheduled task failed", task=self.name, error=str(exc))
            self._stop_event.wait(timeout=self.interval)

    def stop(self):
        self._stop_event.set()
        Log.info("Scheduled task stopped", task=self.name)
```

```
def warm_cache():
    from tina4_python.cache import cache_set
    Log.debug("Cache warming started")
    # Fetch expensive data and pre-populate the cache
    cache_set("featured_products", fetch_featured_products(), ttl=300)
    Log.debug("Cache warming complete")

def fetch_featured_products():
    return [
        {"id": 1, "name": "Wireless Keyboard", "price": 79.99},
        {"id": 2, "name": "USB-C Hub", "price": 49.99}
    ]


runner = ServiceRunner()
runner.register("cache_warmer", ScheduledTask("cache_warmer", warm_cache, interval_seconds=240))
runner.start()
```

---

6. Multiple Services

Register as many services as you need:

```
from tina4_python.service import ServiceRunner

runner = ServiceRunner()
runner.register("email_worker", EmailWorker())
runner.register("sms_worker", SmsWorker())
runner.register("cache_warmer", ScheduledTask("cache_warmer", warm_cache, 240))
runner.register("health_check", PollingService("https://api.internal/health", 30))

runner.start()
print("All services started")
```

All four services start concurrently. Each runs in its own thread.

---

7. Stopping Services Gracefully

Stop all services at once:

```
runner.stop()
```

`stop()` calls `stop()` on each registered service and waits for their threads to finish. Use this in your application shutdown handler:

```
import signal

def shutdown_handler(signum, frame):
    print("Shutting down services...")
    runner.stop()
    print("All services stopped.")

signal.signal(signal.SIGTERM, shutdown_handler)
signal.signal(signal.SIGINT, shutdown_handler)
```

Stop a single service by name:

```
runner.stop_service("cache_warmer")
```

---

8. Integrating with the App

Start the runner in your `src/app.py`:

```
# src/app.py
from tina4_python.tina4 import Tina4
from src.worker_services import runner

app = Tina4()

@app.on_startup
def start_services():
    runner.start()

@app.on_shutdown
def stop_services():
    runner.stop()
```

---

9. Exercise: Cache-Warming Background Service

Build a background service that keeps a product cache warm.

Requirements

- Create a `ProductCacheWarmer` service that:
- Runs every 60 seconds
- Fetches the top 20 products
- Stores them in cache with a 120-second TTL
- Logs a summary each time it runs
- Create a `GET /api/products/featured` route that:
- Returns products from cache (never hits the "database" directly)
- Returns a 503 if the cache is not yet warm
- Register the service with the runner and start it on app startup

Test with:

```
# Before the cache warms (first 60 seconds)
curl http://localhost:7146/api/products/featured
# 503 Service Unavailable -- cache not ready

# After the cache warmer runs
curl http://localhost:7146/api/products/featured
# 200 OK with product list
```

---

10. Solution

```
# src/workers/product_cache_warmer.py
import threading
from tina4_python.cache import cache_set
from tina4_python.debug import Log


FEATURED_PRODUCTS = [
    {"id": i, "name": f"Product {i}", "price": round(9.99 * i, 2)}
    for i in range(1, 21)
]


class ProductCacheWarmer:
    def __init__(self, interval=60):
        self.interval = interval
        self._stop_event = threading.Event()

    def run(self):
        Log.info("ProductCacheWarmer started", interval=self.interval)
        while not self._stop_event.is_set():
            self._warm()
            self._stop_event.wait(timeout=self.interval)

    def stop(self):
        self._stop_event.set()
        Log.info("ProductCacheWarmer stopped")

    def _warm(self):
        products = FEATURED_PRODUCTS[:20]
        cache_set("products:featured", products, ttl=120)
        Log.info("Cache warmed", product_count=len(products))
```

```
# src/routes/featured_products.py
from tina4_python.core.router import get
from tina4_python.cache import cache_get


@get("/api/products/featured")
async def featured_products(request, response):
    products = cache_get("products:featured")

    if products is None:
        return response(
            {"error": "Service warming up, please retry in a moment"},
            503
        )

    return response({"products": products, "count": len(products)})
```

```
# src/app.py (startup integration)
from tina4_python.service import ServiceRunner
from src.workers.product_cache_warmer import ProductCacheWarmer

runner = ServiceRunner()
runner.register("product_cache_warmer", ProductCacheWarmer(interval=60))
```

```
runner.start()
```

---

11. Gotchas

1. Blocking the main thread

Problem: `runner.start()` blocks because a service's `run()` calls `runner.start()` again recursively, or a service does not return from `run()`.

Fix: `run()` must contain the loop internally. `runner.start()` is non-blocking — it launches threads and returns. Each service owns its own loop.

2. No stop() method

Problem: `runner.stop()` raises `AttributeError` because the service class does not implement `stop()`.

Fix: Always implement `stop()`. Set a flag or clear a threading event that the `run()` loop checks.

3. Exceptions crashing the service thread

Problem: An unhandled exception in `run()` kills the thread. The service stops silently.

Fix: Wrap the loop body in `try/except`. Log the error and continue. Only exit the loop on a stop signal.

4. Service starts before .env is loaded

Problem: A service reads an environment variable at construction time, before Tina4 loads `.env`.

Fix: Read environment variables inside `run()` or `_warm()`, not in `__init__`. Or use a factory pattern: construct the service inside an `on_startup` hook, after `.env` is loaded.

MCP Dev Tools

1. What is MCP?

The Model Context Protocol connects AI coding tools to your running application. Claude Code, Cursor, and other assistants speak this protocol. When they connect, they see your database, routes, templates, and files -- not as static text, but as live, queryable tools.

Tina4 ships a built-in MCP server. It starts automatically in dev mode. No configuration needed.

---

2. How It Works

Set `TINA4_DEBUG=true` in your `.env` file and start the server:

```
tina4 serve
```

The console prints:

```
MCP server available at http://localhost:7146/__dev/mcp
```

Tina4 writes the connection details to `.claude/settings.json` automatically. Claude Code discovers the server on its next restart. Cursor reads the same file.

Localhost-Only Security

The built-in MCP server activates only when two conditions hold:

- `TINA4_DEBUG=true`
- The server runs on `localhost`, `127.0.0.1`, or `0.0.0.0`

Deploy to a remote server and the MCP endpoint vanishes -- even with debug enabled. This prevents accidental exposure of database queries and file editing in production.

To enable on a staging server (rare, intentional), set:

```
TINA4_MCP_REMOTE=true
```

---

3. Connecting AI Tools

Claude Code

Tina4 auto-generates `.claude/settings.json`:

```
{
  "mcpServers": {
    "tina4-dev": {
      "url": "http://localhost:7146/__dev/mcp/sse"
    }
  }
}
```

Restart Claude Code. It connects and lists the tools.

Cursor

Copy the same MCP config into Cursor's settings. The SSE URL is identical.

Manual Connection

Any MCP client can connect via:

- **SSE endpoint**: `GET http://localhost:7146/__dev/mcp/sse` -- returns the message endpoint URL
- **Message endpoint**: `POST http://localhost:7146/__dev/mcp/message` -- accepts JSON-RPC 2.0 messages

---

4. Built-in Tools

The MCP server exposes 48 tools organized by category. The exact list lives in `tina4_python/mcp/tools.py` -- the registration block at the bottom of that file is authoritative.

Database

Tool	Description
`database_query`	Execute a read-only SQL query (SELECT)
`database_execute`	Execute arbitrary SQL (INSERT, UPDATE, DELETE, DDL)
`database_tables`	List all tables in the database
`database_columns`	Get column definitions for a specific table

Example -- an AI assistant queries your database directly:

```
Tool: database_query
Arguments: {"sql": "SELECT id, name, email FROM users WHERE active = 1"}
```

The `database_execute` tool handles write operations. It commits automatically after each statement.

Routes

Tool	Description
`route_list`	List all registered routes with methods and auth status
`route_test`	Call a route and return status, body, and headers
`swagger_spec`	Return the full OpenAPI 3.0.3 specification

The `route_test` tool lets an AI assistant verify endpoints without leaving the conversation:

```
Tool: route_test
Arguments: {"method": "GET", "path": "/api/users"}
```

Templates

Tool	Description
`template_render`	Render a Frond template string with provided data

Useful for testing template snippets:

```
Tool: template_render
Arguments: {"template": "Hello {{ name }}!", "data": "{\"name\": \"Alice\"}"}
```

Files

Tool	Description
`file_read`	Read a project file (relative to project root)
`file_write`	Write or update a project file (full overwrite)
`file_patch`	Targeted edit -- replace `old_string` with `new_string` in a file
`file_list`	List files in a directory
`asset_upload`	Upload a file to `src/public/`

File operations are sandboxed. Paths that escape the project directory are rejected. An AI assistant cannot read `/etc/passwd` or write outside your project.

Migrations

Tool	Description
`migration_status`	List pending and completed migrations
`migration_create`	Create a new migration file
`migration_run`	Run all pending migrations

Data and ORM

Tool	Description
`orm_describe`	List all ORM models with fields, types, and relationships
`seed_table`	Seed a table with fake data

Infrastructure

Tool	Description
`queue_status`	Queue size by status (pending, completed, failed)
`session_list`	Active sessions with data
`cache_stats`	Response cache hit/miss statistics

Debugging

Tool	Description
`log_tail`	Read recent log entries
`error_log`	Recent errors and exceptions with stack traces
`env_list`	Environment variables (sensitive values redacted)
`system_info`	Framework version, Python version, project info

The `env_list` tool redacts any variable containing "secret", "password", "token", "key", or "credential" in its name.

Project introspection

Tool	Description
`git_status`	Show current branch, modified/untracked files, and recent commits
`deps_list`	List the project's declared Python dependencies (from `pyproject.toml`)
`project_overview`	One-shot snapshot: dependencies, routes, models, tables, errors, git

Persistent code index

Tool	Description
`index_rebuild`	Refresh the persistent project index (lazy, mtime-based)
`index_search`	Find files by path, symbol, route, or summary -- use FIRST for "where is X"
`index_file`	Full index entry for one file: symbols, routes, imports
`index_overview`	Project shape: files by language, routes, models, recent edits

Framework documentation (prose)

Tool	Description
`docs_list`	List framework documentation files
`docs_search`	Search Tina4 framework docs for a query string -- use before guessing
`docs_section`	Return a full markdown section from a framework doc file

Live API RAG (reflection)

`api_*` tools query the live framework reflection -- actual class/method signatures, file/line locations, with a `framework` vs `user` source tag. Use these for "what's the signature of X?" questions; `docs_*` is for "explain queues conceptually".

Tool	Description
`api_search`	Live API search -- finds framework + user classes/methods by name/summary
`api_class`	Live class reflection -- returns full method list and signatures for an FQN
`api_method`	Live method reflection -- returns signature, params, return type, file, line

Plan management

The `plan_*` family lets an AI assistant cooperate with a markdown plan file in `plan/`. Tina4 uses these to keep a steady record of in-progress refactors and multi-session work.

Tool	Description

`plan_current`	The active plan: title, steps (done/not), next step, progress
`plan_list`	All plans in `plan/` with progress and which one is active
`plan_create`	Create a new markdown plan and make it active
`plan_switch_to`	Make a different plan the active one
`plan_complete_step`	Tick a step as done (call the moment the step finishes)
`plan_add_step`	Append a new unchecked step to the current plan
`plan_note`	Append a timestamped note/breadcrumb to the current plan
`plan_archive`	Move a finished plan to `plan/done/` and clear the current pointer
`plan_read`	Full structured view of any plan by filename
`plan_flesh`	Auto-generate concrete build steps via qwen and append them to a plan

---

5. Protocol Details

The MCP server uses JSON-RPC 2.0 over HTTP. Two endpoints serve the protocol:

SSE Endpoint (GET)

```
GET /__dev/mcp/sse
Content-Type: text/event-stream
```

Returns the message endpoint URL as a server-sent event:

```
event: endpoint
data: http://localhost:7146/__dev/mcp/message
```

Message Endpoint (POST)

```
POST /__dev/mcp/message
Content-Type: application/json
```

Accepts JSON-RPC 2.0 requests:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "tools/list",
  "params": {}
```

```
}
```

Returns JSON-RPC 2.0 responses:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "result": {
    "tools": [
      {"name": "database_query", "description": "Execute a read-only SQL query", "inputSchema":
    ]
  }
}
```

Lifecycle

- Client sends `initialize` -- server responds with capabilities
- Client sends `notifications/initialized` -- server acknowledges
- Client calls `tools/list` -- server returns all registered tools with schemas
- Client calls `tools/call` with tool name and arguments -- server executes and returns results
- Client calls `resources/list` and `resources/read` for data resources

---

6. Troubleshooting

MCP server not starting:

- Check `TINA4_DEBUG=true` in `.env`
- Verify the server is running on localhost (not a remote host)

Claude Code not detecting the server:

- Restart Claude Code after starting the Tina4 server
- Check `.claude/settings.json` exists with the correct URL
- Verify the port matches your server's port

Tools returning errors:

- Database tools require `orm_bind(db)` in `app.py`
- File tools are sandboxed to the project directory
- `database_execute` only works on localhost

Remote server — MCP disabled:

- This is intentional. Set `TINA4_MCP_REMOTE=true` only on staging environments you trust
- Never enable MCP on production servers

Building Custom MCP Servers

1. Beyond Dev Tools

Chapter 27 covered the built-in MCP server that ships with Tina4. It exposes framework internals for AI-assisted development. This chapter goes further: you build your own MCP servers that expose your application's business logic.

A CRM system exposes customer lookup. An accounting system exposes invoice queries. A warehouse system exposes inventory checks. Any domain logic that an AI assistant should access becomes an MCP tool.

---

2. Creating an MCP Server

Import `McpServer` and create an instance on any path:

```
from tina4_python.mcp import McpServer

mcp = McpServer("/api/my-tools", name="My App Tools", version="1.0.0")
```

The server registers HTTP endpoints at:

- `POST /api/my-tools/message` -- JSON-RPC message handler
- `GET /api/my-tools/sse` -- SSE endpoint for client discovery

Register it with the router in `app.py` before `run()`:

```
from tina4_python.core.router import get, post
mcp.register_routes(__import__("tina4_python.core.router", fromlist=["router"]))
```

---

3. Registering Tools with @mcp_tool

The `@mcp_tool` decorator turns a function into an MCP tool. Type hints become the input schema automatically:

```
from tina4_python.mcp import McpServer, mcp_tool

mcp = McpServer("/crm/mcp", name="CRM Tools")

@mcp_tool("lookup_customer", description="Find a customer by email", server=mcp)
def lookup_customer(email: str):
    """Search the customer database by email address."""
    return db.fetch_one("SELECT * FROM customers WHERE email = ?", [email])

@mcp_tool("recent_orders", description="Get recent orders for a customer", server=mcp)
def recent_orders(customer_id: int, limit: int = 10):
    """Fetch the most recent orders for a customer."""
    result = db.fetch(
        "SELECT * FROM orders WHERE customer_id = ? ORDER BY created_at DESC",
        [customer_id], limit=limit
    )
```

```
    return result.to_array()
```

The decorator extracts:

- **Parameter names** from the function signature
- **Types** from type hints (`str` -> `"string"`, `int` -> `"integer"`, etc.)
- **Required vs optional** -- parameters with defaults are optional
- **Description** from the `description` argument or the docstring

An AI assistant sees these tools and their schemas:

```
{
  "name": "lookup_customer",
  "description": "Find a customer by email",
  "inputSchema": {
    "type": "object",
    "properties": {
      "email": {"type": "string"}
    },
    "required": ["email"]
  }
}
```

---

4. Registering Resources with @mcp_resource

Resources are read-only data endpoints. They expose reference data that AI assistants can browse:

```
from tina4_python.mcp import mcp_resource

@mcp_resource("crm://product-catalog", description="All active products", server=mcp)
def product_catalog():
    return db.fetch("SELECT id, name, price, category FROM products WHERE active = 1").to_array(

@mcp_resource("crm://tax-rates", description="Current tax rates by region", server=mcp)
def tax_rates():
    return db.fetch("SELECT region, rate FROM tax_rates").to_array()
```

Resources are accessed via `resources/list` and `resources/read` in the MCP protocol.

---

5. Class-Based MCP Services

Group related tools into a service class. Each method with `@mcp_tool` becomes a tool:

```
from tina4_python.mcp import McpServer, mcp_tool

mcp = McpServer("/accounting/mcp", name="Accounting Tools")

class AccountingService:
    def __init__(self, db):
        self.db = db

    @mcp_tool("invoice_lookup", description="Find an invoice by number", server=mcp)
```

```
    def lookup(self, invoice_no: str):
        return self.db.fetch_one(
            "SELECT * FROM invoices WHERE invoice_no = ?", [invoice_no]
        )

    @mcp_tool("outstanding_balances", description="List all unpaid invoices", server=mcp)
    def balances(self, min_amount: float = 0.0):
        return self.db.fetch(
            "SELECT * FROM invoices WHERE paid = 0 AND total >= ?", [min_amount]
        ).to_array()

    @mcp_tool("monthly_summary", description="Revenue summary for a month", server=mcp)
    def summary(self, year: int, month: int):
        return self.db.fetch_one(
            "SELECT SUM(total) as revenue, COUNT(*) as invoice_count "
            "FROM invoices WHERE strftime('%Y', created_at) = ? "
            "AND strftime('%m', created_at) = ?",
            [str(year), f"{month:02d}"]
        )

# Create the service instance — methods are already registered via decorators
accounting = AccountingService(db)
```

---

6. Securing MCP Endpoints

By default, developer MCP servers are public. Add authentication using the standard Tina4 middleware:

```
from tina4_python.core.router import middleware, secured

# Secure the entire MCP path
@secured()
@middleware(AuthMiddleware)
def register_mcp():
    mcp.register_routes(router)
```

Or check the bearer token inside individual tools:

```
@mcp_tool("sensitive_data", description="Access restricted data", server=mcp)
def sensitive_data(token: str):
    payload = Auth.valid_token_static(token)
    if not payload:
        return {"error": "Unauthorized"}
    return db.fetch("SELECT * FROM sensitive_table").to_array()
```

---

7. Testing MCP Tools

Use `TestClient` to test MCP endpoints without starting a server, or test tool functions directly:

```
# Test the tool function directly
def test_lookup_customer():
    result = lookup_customer("alice@example.com")
    assert result is not None
    assert result["email"] == "alice@example.com"
```

```
# Test via MCP protocol
def test_mcp_tool_call():
    import json
    resp = json.loads(mcp.handle_message({
        "jsonrpc": "2.0",
        "id": 1,
        "method": "tools/call",
        "params": {
            "name": "lookup_customer",
            "arguments": {"email": "alice@example.com"}
        }
    }))
    assert "result" in resp
    content = resp["result"]["content"][0]["text"]
    assert "alice" in content.lower()
```

---

8. Complete Example: CRM MCP Server

Here is a full working example -- a CRM system with customer, order, and product tools:

```
# app.py
from tina4_python.core import run
from tina4_python.orm import orm_bind
from tina4_python.database import Database
from tina4_python.mcp import McpServer, mcp_tool, mcp_resource

db = Database("sqlite:///crm.db")
orm_bind(db)

# Create MCP server
crm_mcp = McpServer("/crm/mcp", name="CRM Assistant", version="1.0.0")

# Tools
@mcp_tool("find_customer", description="Search customers by name or email", server=crm_mcp)
def find_customer(query: str):
    return db.fetch(
        "SELECT * FROM customers WHERE name LIKE ? OR email LIKE ?",
        [f"%{query}%", f"%{query}%"]
    ).to_array()

@mcp_tool("customer_orders", description="Get all orders for a customer", server=crm_mcp)
def customer_orders(customer_id: int):
    return db.fetch(
        "SELECT o.*, GROUP_CONCAT(oi.product_name) as items "
        "FROM orders o LEFT JOIN order_items oi ON o.id = oi.order_id "
        "WHERE o.customer_id = ? GROUP BY o.id ORDER BY o.created_at DESC",
        [customer_id]
    ).to_array()

@mcp_tool("create_note", description="Add a note to a customer record", server=crm_mcp)
def create_note(customer_id: int, note: str):
    db.insert("customer_notes", {"customer_id": customer_id, "note": note})
    return {"success": True}

# Resources
@mcp_resource("crm://products", description="Product catalog", server=crm_mcp)
def products():
```

```
    return db.fetch("SELECT * FROM products WHERE active = 1").to_array()

@mcp_resource("crm://stats", description="CRM statistics", server=crm_mcp)
def stats():
    customers = db.fetch_one("SELECT COUNT(*) as count FROM customers")
    orders = db.fetch_one("SELECT COUNT(*) as count, SUM(total) as revenue FROM orders")
    return {
        "customers": customers["count"],
        "orders": orders["count"],
        "revenue": orders["revenue"],
    }

# Register routes
import tina4_python.core.router as router
crm_mcp.register_routes(router)

if __name__ == "__main__":
    run()
```

Connect Claude Code to `http://localhost:7146/crm/mcp/sse` and ask:

> *"Find all customers named Smith and show their recent orders"*

The AI calls `find_customer` with `query: "Smith"`, then `customer_orders` for each result. No custom API needed. The MCP protocol handles it.

---

9. Best Practices

- **One server per domain** -- CRM tools on `/crm/mcp`, accounting on `/accounting/mcp`
- **Keep tools focused** -- one query per tool, not a Swiss-army-knife tool
- **Use type hints** -- they become the schema. An AI assistant cannot call a tool correctly without knowing the parameter types
- **Return structured data** -- dicts and lists, not formatted strings. Let the AI format for the user
- **Secure production endpoints** -- use `@secured()` or middleware for any MCP server that runs outside localhost
- **Test tools directly** -- call the Python function in your test suite, not just through the MCP protocol

Dev Tools

1. Debugging at 2am

2am. Production monitoring pings you. A 500 error on checkout. You pull up the dev dashboard. The request inspector shows the failing request. The stack trace points to line 47 of `src/routes/checkout.py` -- an `AttributeError` on the shipping address because the user skipped the form field. You add a None check. Push the fix. Back to sleep. Total time: 30 seconds.

Tina4's dev tools are not an afterthought. They ship with the framework from day one. When `TINA4_DEBUG=true`, you get a development dashboard, an error overlay with source code, live reload, a request inspector with replay, a SQL query runner, a queue monitor, and system info -- all without installing extra packages.

---

2. Enabling the Dev Dashboard

Set `TINA4_DEBUG=true` in your `.env`:

```
TINA4_DEBUG=true
```

Restart your server and navigate to:

```
http://localhost:7146/__dev
```

No token or additional environment variables needed. The dashboard runs only when debug mode is on. Set `TINA4_DEBUG=false` in production and the entire dashboard disappears.

---

3. Dashboard Overview

The dev dashboard has several sections. Navigation tabs run along the top.

System Overview

The landing page shows at a glance:

- **Framework version** -- The installed Tina4 Python version
- **Python version** -- The running Python version and loaded packages
- **Uptime** -- How long the server has been running
- **Memory usage** -- Current and peak memory consumption
- **Database status** -- Connection status, database engine, file size (for SQLite)
- **Environment** -- Current `.env` variables (sensitive values masked)
- **Project structure** -- Directory listing of your project with file counts

Check here first when something feels off. Is the database connected? Is the right Python version running? Are the environment variables loaded?

---

4. The Dev Toolbar

Visit any HTML page in your application. A debug toolbar appears at the bottom of the page. Thin bar. Click it to expand.

The toolbar shows:

Field	What It Means
Request	HTTP method and URL of the current request
Status	HTTP response status code
Time	Total request processing time in milliseconds
DB	Number of database queries executed and total query time
Memory	Python memory used for this request
Template	The template rendered and how long rendering took
Session	Session ID and session data summary
Route	Which route handler matched this request

Click any section to expand it. Clicking "DB" shows every SQL query that ran during the request, with the query text, parameters, execution time, and row count.

Disabling the Toolbar

The toolbar hides when `TINA4_DEBUG=false`. You can also hide it for specific routes by returning a response with the `X-Debug-Toolbar: off` header:

```
from tina4_python.core.router import get

@get("/api/data")
async def api_data(request, response):
    return response({"data": "no toolbar"}, headers={
        "X-Debug-Toolbar": "off"
    })
```

This is useful for API endpoints that return JSON -- the toolbar only matters for HTML pages.

---

5. Error Overlay

An unhandled exception occurs. Tina4 does not show a generic "500 Internal Server Error" page. It shows a detailed error overlay:

- **Exception type and message** -- What went wrong, in plain language

- **Stack trace** -- Every function call that led to the error, from entry point to crash
- **Source code** -- The Python code around the line that threw the exception, with the failing line highlighted
- **Request data** -- HTTP method, URL, headers, body, and query parameters of the request that triggered the error
- **Environment** -- Relevant `.env` variables at the time of the error

Example Error

Write this handler:

```
@get("/api/users/{user_id}")
async def get_user(request, response):
    user_id = request.params["user_id"]
    user = get_user_from_db(user_id)
    return response({"name": user.name, "email": user.email})
```

If `get_user_from_db()` returns `None` for a missing user, the error overlay shows:

```
AttributeError: 'NoneType' object has no attribute 'name'

  File: src/routes/users.py, line 5

  3 | async def get_user(request, response):
  4 |     user_id = request.params["user_id"]
  5 |     user = get_user_from_db(user_id)
> 6 |     return response({"name": user.name, "email": user.email})
  7 |

  Request: GET /api/users/999
  Headers: {"Accept": "application/json"}
```

The highlighted line makes the cause obvious. `user` is `None` and the code accesses `.name` on it.

Error Overlay in Production

When `TINA4_DEBUG=false`, the error overlay is disabled. Users see a clean error page. Full error details go to `logs/error.log` for later investigation.

---

6. The Gallery

The gallery is a collection of ready-to-use code examples built into the dev dashboard. Each gallery item includes:

- A description of what it does
- The complete source code (routes, templates, models)
- A "Try It" button that installs the example into your project

Available gallery items:

- **JWT Authentication** -- Complete login/register flow with token management
- **CRUD API** -- Full REST API with ORM model

- **File Upload** -- Multipart form handling with file storage
- **WebSocket Chat** -- Real-time chat with rooms
- **Email Contact Form** -- Form with Messenger integration
- **Dashboard Template** -- Admin dashboard with tina4css

Click "Try It" on any gallery item. Tina4 creates the necessary files in your project. Modify them to fit your needs.

---

7. Live Reload

When `TINA4_DEBUG=true`, Tina4 watches your project files for changes and refreshes the browser automatically. Edit a route file. Save it. The browser refreshes with the new code. No manual restart required.

```
tina4 serve

  Tina4 Python v3.11.12
  HTTP server running at http://0.0.0.0:7146
  File watcher active — press Ctrl+C to stop
```

Live reload watches:

- `src/**` -- Routes, ORM, middleware, templates, SCSS
- `migrations/` -- SQL migrations
- `.env` -- Environment variables

How It Works

The `tina4` Rust CLI is the sole file watcher for the whole Tina4 stack — Python, PHP, Ruby, and Node.js all share it. There is no framework-side watcher (the old `dev_reload.py` was removed in 3.11.x).

- The CLI watches `src/`, `migrations/`, `.env` using the `notify` crate. Events are filtered to real source changes — metadata/access events, `__pycache__`, `.git`, `node_modules`, `logs`, `.log`/`.db*`/`.pyc`/`.swp` files are ignored. A real mtime check also defeats overlayfs / polling-mode spurious events (Podman, distrobox).
- On a real change, the CLI POSTs `/__dev/api/reload` to the framework. The server keeps running.
- The framework bumps an in-memory reload counter and broadcasts `{type: "reload"}` over WebSocket at `/__dev_reload`. `GET /__dev/api/mtime` returns the counter for browsers using the polling fallback.
- The dev-toolbar script in your page reloads the browser. SCSS/CSS changes are signalled as `type: "css"` and swap the stylesheet without a full reload.

The reload happens in under a second. Edit code. Switch to the browser. The changes are already there.

If you run without the Rust CLI (e.g. Docker), set `TINA4_OVERRIDE_CLIENT=true` — there is no automatic reload in that mode.

---

8. Hot-Patching with jurigged

For faster iteration, Tina4 supports hot-patching via jurigged. Hot-patching updates function definitions in the running server without restarting it:

- No connection drops (WebSocket clients stay connected)
- No cache loss (in-memory caches stay warm)
- No session interruption
- Sub-second updates

Install jurigged:

```
uv add --dev jurigged
```

Start the server with hot-patching:

```
tina4 serve --hot
```

```
  Tina4 Python v3.0.0
  HTTP server running at http://0.0.0.0:7146
  Hot-patching enabled (jurigged)
```

Edit any route handler and save. The function updates in place -- without restarting the server. The next request uses the new code.

When Hot-Patching Cannot Help

Hot-patching works for function body changes. It does not work for:

- Adding or removing route decorators (requires restart)
- Changing module-level variables (requires restart)
- Modifying class definitions (may require restart)
- Changing `.env` (requires restart)

For these changes, the server does a full reload.

---

9. Request Inspector

The request inspector records every HTTP request to your application. For each request:

- **Timestamp** -- When the request arrived
- **Method** -- GET, POST, PUT, DELETE

- **URL** -- The full URL including query parameters
- **Status** -- The HTTP status code returned (color-coded: green for 2xx, yellow for 4xx, red for 5xx)
- **Time** -- How long the request took to process
- **Request ID** -- A unique identifier for correlating logs

Click on any request to see its full details.

Request Details Panel

- **Headers** -- All request headers (Accept, Content-Type, Authorization)
- **Body** -- The request body (for POST/PUT/PATCH), formatted as JSON if applicable
- **Query parameters** -- Parsed URL query parameters
- **Route match** -- Which route definition matched this request
- **Middleware** -- Which middleware ran and how long each took
- **Database queries** -- Every SQL query executed during this request, with timing
- **Template renders** -- Which templates rendered and how long each took
- **Response headers** -- The response headers sent back
- **Response body** -- The first 1000 characters of the response body

Filtering Requests

The inspector supports filtering:

- **By status**: Click the status code badges at the top (e.g., show only 5xx errors)
- **By method**: Filter by GET, POST, PUT, DELETE
- **By path**: Search for a URL pattern (e.g., `/api/` for API requests only)
- **By time range**: Show requests from the last 5 minutes, 1 hour, or all time

Request Replay

Click "Replay" on any request to re-send it. The inspector fires the same method, URL, headers, and body. You reproduce an error without constructing the curl command by hand.

This is the fastest path from "what happened?" to "I can see it happen again." Find a failing request. Hit Replay. Watch the error overlay show the stack trace. Fix the code. Replay again. Green status code. Done.

No more `print()` statements scattered through your code. The inspector shows what happened. Every request. Every detail. With one-click replay.

---

10. SQL Query Runner

The dev dashboard includes a SQL query runner. Execute queries against your database:

- **Execute queries** -- Run any SQL statement
- **See results** -- Formatted table with column headers and row numbers
- **View query timing** -- How long each query took
- **Browse tables** -- A sidebar lists all tables with their column definitions
- **Export results** -- Download as CSV

```
SELECT * FROM products WHERE category = 'Electronics' ORDER BY price DESC;
```

The results display in a table with column headers, row numbers, and data type indicators. Copy results, export as CSV, or run another query.

Safety

The query runner is read-write in development. You can run INSERT, UPDATE, and DELETE statements. Be careful -- there is no undo. In shared environments, consider a read-only database connection for the dev dashboard.

The query runner only works when `TINA4_DEBUG=true`. It is disabled in production.

---

11. Queue Monitor

If your application uses background job queues (Chapter 12 -- Queues), the dev dashboard includes a queue monitor. The monitor shows five categories:

- **Pending jobs** -- Jobs waiting to be processed
- **Active jobs** -- Jobs currently being processed
- **Completed jobs** -- Recently completed jobs with timing
- **Failed jobs** -- Jobs that threw exceptions, with error details
- **Dead-letter queue** -- Jobs that failed too many times

For each job, you see:

- The job function name
- The payload (serialized arguments)
- When it was enqueued
- When it started processing (if active)
- The error message and stack trace (if failed)
- How many times it has been retried

You can also take action:

- **Retry a failed job** -- Click "Retry" to move it back to the pending queue
- **Delete a job** -- Remove it from any queue
- **Pause/resume the queue** -- Stop processing without losing jobs

The queue monitor gives you a window into your background work. A job fails. You see the error. You fix the code. You hit Retry. The job processes. No log file hunting. No guesswork about what payload caused the failure.

---

12. System Info

The System Info tab shows detailed information about your environment:

- **Python Configuration** -- Version, installed packages, virtual environment path, memory limit
- **Database Info** -- Engine, version, connection details, table sizes, index information
- **Server Info** -- OS, hostname, server software, document root
- **Tina4 Config** -- All loaded `.env` variables (sensitive values masked), auto-discovered routes, registered middleware, ORM models
- **Disk Usage** -- Size of your project directory, data directory, logs directory

This panel solves the "it works on my machine" problem. Compare the System Info output between two environments. Spot the difference. The wrong Python version. A missing package. A misconfigured environment variable. The answer is in the panel.

---

13. Exercise: Debug a Failing Route

Your colleague wrote a route that fails. Use the dev tools to find and fix the bug.

Setup

Create `src/routes/buggy.py`:

```
from tina4_python.core.router import get, post
from tina4_python.cache import cache_get, cache_set

@get("/api/buggy/users")
async def buggy_users(request, response):
    users = cache_get("all_users")
    if users:
        return response({"users": users, "source": "cache"})

    # Bug 1: This query has an error
    db = Database.get_connection()
    users = db.fetch_all("SELCT * FROM users ORDER BY name")

    cache_set("all_users", users, ttl=300)
    return response({"users": users, "source": "database"})


@post("/api/buggy/users")
async def buggy_create_user(request, response):
    body = request.body

    # Bug 2: Missing validation
    user = User()
```

```
    user.name = body["name"]
    user.email = body["email"]
    user.save()

    # Bug 3: Wrong status code
    return response({"message": "User created", "user": user.to_dict()})
```

Requirements

- Open the dev dashboard at `http://localhost:7146/__dev`
- Hit the `GET /api/buggy/users` endpoint and find Bug 1 using the error overlay
- Hit the `POST /api/buggy/users` endpoint without a body and find Bug 2 using the request inspector
- Fix all three bugs:
- Bug 1: Fix the SQL syntax error
- Bug 2: Add validation for required fields
- Bug 3: Return 201 instead of 200
- Use Request Replay to verify each fix

Solution

```
from tina4_python.core.router import get, post
from tina4_python.cache import cache_get, cache_set

@get("/api/buggy/users")
async def buggy_users(request, response):
    users = cache_get("all_users")
    if users:
        return response({"users": users, "source": "cache"})

    db = Database.get_connection()
    users = db.fetch_all("SELECT * FROM users ORDER BY name")  # Fixed: SELCT -> SELECT

    cache_set("all_users", users, ttl=300)
    return response({"users": users, "source": "database"})


@post("/api/buggy/users")
async def buggy_create_user(request, response):
    body = request.body

    # Fixed: Added validation
    if not body.get("name") or not body.get("email"):
        return response({"error": "Name and email are required"}, 400)

    user = User()
    user.name = body["name"]
    user.email = body["email"]
    user.save()

    return response({"message": "User created", "user": user.to_dict()}, 201)  # Fixed: 201
```

The dev tools made each bug visible. The error overlay showed the SQL syntax error with the exact line. The request inspector showed the missing body causing a `KeyError`. Request Replay

let you re-send the fixed requests without leaving the dashboard.

---

14. Gotchas

1. Dev Dashboard Accessible on Network

Problem: Anyone on your network can access the dev dashboard.

Cause: `TINA4_DEBUG=true` makes the dashboard available at `/__dev`.

Fix: In production, set `TINA4_DEBUG=false` to disable the dashboard. In shared development environments, restrict network access.

2. Live Reload Causes Connection Drops

Problem: WebSocket connections drop every time you save a file.

Cause: The server restarts on file change, which closes all active connections.

Fix: Use hot-patching (`--hot` flag) instead of live reload for WebSocket development. Hot-patching updates function bodies without restarting the server, so connections stay open.

3. Error Overlay Shows in Production

Problem: Users see Python stack traces when errors occur.

Cause: `TINA4_DEBUG=true` is set in the production environment.

Fix: Set `TINA4_DEBUG=false` in production. The error overlay is replaced by a clean error page. Full details go to `logs/error.log`.

4. Request Inspector Slows Down the Server

Problem: The server becomes slower as it runs longer.

Cause: The request inspector stores every request in memory. After thousands of requests, memory usage grows.

Fix: The inspector limits storage to the last 1000 requests. If you notice slowness, clear the inspector from the dashboard. In production, the inspector is disabled.

5. SQL Runner Executes Destructive Queries

Problem: Someone ran `DROP TABLE users` in the SQL query runner.

Cause: The SQL runner executes any valid SQL query. There is no confirmation step.

Fix: The SQL runner only works when `TINA4_DEBUG=true`. Never leave debug mode on in production. For sensitive development databases, use a read-only database connection for the SQL runner.

6. Hot-Patching Does Not Pick Up New Routes

Problem: You added a new route decorator but it does not respond to requests.

Cause: Hot-patching (jurigged) can update function bodies but cannot register new decorators. Adding `@get("/new-route")` requires the decorator to execute, which only happens at import time.

Fix: Restart the server when adding or removing routes. Hot-patching modifies existing route behavior. It does not add new routes.

7. Template Changes Not Reflected

Problem: You edited a template but the page still shows the old version.

Cause: Template caching is enabled (`TINA4_TEMPLATE_CACHE_TTL=true`). The compiled template serves from cache.

Fix: In development, set `TINA4_TEMPLATE_CACHE_TTL=false` (this is the default when `TINA4_DEBUG=true`). If you enabled template caching manually, disable it for development.

8. Queue Monitor Shows No Jobs

Problem: The Queue Monitor tab is empty even though your application uses queues.

Cause: The queue system is not running. Jobs are enqueued but no worker processes them.

Fix: Start a queue worker: `tina4 queue work`. The queue monitor reflects the state of the queue storage (database or Redis). If no worker is running, jobs sit in "Pending" and never move to "Active" or "Completed."

Tina4 CLI

1. Getting a New Developer Up to Speed

Monday morning. A new developer joins your team. You hand them the repo URL. By 10am they have a running project, a new database model, CRUD routes, a migration, and a deployment to staging. All from the command line. No documentation scavenger hunt. No boilerplate copy-paste.

The Tina4 CLI is a single Rust binary. It manages all four Tina4 frameworks (PHP, Python, Ruby, Node.js). The commands are identical across languages. Learn the CLI for Python. You know it for PHP.

---

2. tina4 init -- Project Scaffolding

You saw this in Chapter 1. Now the details.

```
tina4 init my-project
```

```
Creating Tina4 project in ./my-project ...
  Detected language: Python (pyproject.toml)
  Created .env
  Created .env.example
  Created .gitignore
  Created src/routes/
  Created src/orm/
  Created migrations/
  Created src/seeds/
  Created src/templates/
  Created src/templates/errors/
  Created src/public/
  Created src/public/js/
  Created src/public/css/
  Created src/public/scss/
  Created src/public/images/
  Created src/public/icons/
  Created src/locales/
  Created data/
  Created logs/
  Created secrets/
  Created tests/

Project created! Next steps:
  cd my-project
  uv sync
  tina4 serve
```

Language Detection

The CLI detects the language from existing files:

File Present	Language
`composer.json`	PHP

`pyproject.toml` or `requirements.txt`	Python
`Gemfile`	Ruby
`package.json`	Node.js

If no language-specific file exists, the CLI asks:

```
tina4 init my-project
```

```
No language detected. Which language?
  1. PHP
  2. Python
  3. Ruby
  4. Node.js
> 2

Creating Tina4 Python project in ./my-project ...
```

Explicit Language Selection

Skip the prompt by specifying the language:

```
tina4 init python my-project
```

This creates a Python project with `pyproject.toml`, `app.py`, and the full directory structure.

Init into an Existing Directory

Already have a project? Add Tina4 structure:

```
cd existing-project
tina4 init .
```

The CLI creates only files and directories that do not exist. It never overwrites.

---

3. tina4 serve -- Dev Server

```
tina4 serve
```

```
  Tina4 Python v3.0.0
  HTTP server running at http://0.0.0.0:7146
  WebSocket server running at ws://0.0.0.0:7146
  Live reload enabled
  Press Ctrl+C to stop
```

`tina4 serve` detects the language and starts the appropriate server. For Python, it spawns the Tina4 Python runtime with SCSS compilation, file watching, and live reload all wired in.

Options

```
tina4 serve --port 8080        # Custom port
tina4 serve --host 127.0.0.1   # Bind to localhost only
tina4 serve --production       # Production mode (no live reload, debug off)
```

Bypassing the CLI

The framework refuses to start without the Rust CLI. For sandboxed environments (Docker images that already wrap the framework, CI runners, APM agents that wrap the Python process directly) set `TINA4_OVERRIDE_CLIENT=true` in `.env` and only then run:

```
TINA4_OVERRIDE_CLIENT=true uv run python app.py
```

This bypasses SCSS compilation and live reload — use it only when the Rust CLI is genuinely unavailable.

---

4. tina4 generate model -- ORM Scaffolding

The `generate model` command creates an ORM model file and a matching migration. One command produces both.

```
tina4 generate model Product
```

```
Created src/orm/product.py
Created migrations/20260322120000_create_products_table.sql
```

The generated model:

```
from tina4_python.orm import ORM

class Product(ORM):
    table_name = "products"
    id: int
    created_at: str
```

The generated migration:

```
-- UP
CREATE TABLE products (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    created_at TEXT DEFAULT CURRENT_TIMESTAMP
);

-- DOWN
DROP TABLE IF EXISTS products;
```

The model is ready to use. Add your fields and run migrations.

Adding Fields

Specify fields on the command line:

```
tina4 generate model Product --fields "name:string,price:float,category:string,in_stock:bool"
```

The generated model includes all the fields:

```
from tina4_python.orm import ORM

class Product(ORM):
    table_name = "products"
    id: int
    name: str
    price: float
```

```
    category: str
    in_stock: bool
    created_at: str
```

And the migration:

```
-- UP
CREATE TABLE products (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    name TEXT NOT NULL DEFAULT '',
    price REAL NOT NULL DEFAULT 0,
    category TEXT NOT NULL DEFAULT '',
    in_stock INTEGER NOT NULL DEFAULT 0,
    created_at TEXT DEFAULT CURRENT_TIMESTAMP
);

-- DOWN
DROP TABLE IF EXISTS products;
```

Field Types

CLI Type	Python Type	SQLite Column
`string`	`str`	`TEXT`
`int`	`int`	`INTEGER`
`float`	`float`	`REAL`
`bool`	`bool`	`INTEGER`
`text`	`str`	`TEXT`
`date`	`str`	`TEXT`

Options

Flag	Description	Example
`--fields`	Comma-separated field definitions	`--fields "name:string,price:float"`
`--auto-crud`	Enable auto-CRUD on the model	`--auto-crud`
`--soft-delete`	Add soft delete support	`--soft-delete`
`--no-migration`	Skip migration generation	`--no-migration`
`--with-route`	Also generate a CRUD route file	`--with-route`

---

5. tina4 generate route -- CRUD Route Scaffolding

The `generate route` command creates a complete CRUD route file with all five REST endpoints. It reads the model's properties and builds routes with proper imports and response handling.

```
tina4 generate route products
```

```
Created src/routes/products.py
```

The generated route file:

```
from tina4_python.core.router import get, post, put, delete

@get("/api/products")
async def list_products(request, response):
    page = int(request.params.get("page", 1))
    per_page = int(request.params.get("per_page", 20))
    offset = (page - 1) * per_page

    products, total = Product.where("1=1", [], limit=per_page, offset=offset)
    results = [p.to_dict() for p in products]

    return response({
        "data": results,
        "page": page,
        "per_page": per_page,
        "count": len(results)
    })


@get("/api/products/{product_id}")
async def get_product(request, response):
    product = Product.find(request.params["product_id"])

    if product is None:
        return response({"error": "Product not found"}, 404)

    return response(product.to_dict())


@post("/api/products")
async def create_product(request, response):
    body = request.body

    product = Product()
    product.name = body.get("name", "")
    product.price = float(body.get("price", 0))
    product.category = body.get("category", "")
    product.in_stock = bool(body.get("in_stock", False))
    product.save()

    return response(product.to_dict(), 201)


@put("/api/products/{product_id}")
async def update_product(request, response):
    product = Product.find(request.params["product_id"])
```

```
    if product is None:
        return response({"error": "Product not found"}, 404)

    body = request.body
    if "name" in body:
        product.name = body["name"]
    if "price" in body:
        product.price = float(body["price"])
    if "category" in body:
        product.category = body["category"]
    if "in_stock" in body:
        product.in_stock = bool(body["in_stock"])
    product.save()

    return response(product.to_dict())


@delete("/api/products/{product_id}")
async def delete_product(request, response):
    product = Product.find(request.params["product_id"])

    if product is None:
        return response({"error": "Product not found"}, 404)

    product.delete()
    return response(None, 204)
```

The generator reads the model's properties and creates routes with type casting and None checks. Customize the generated code immediately. It is regular Python, not magic.

Options

Flag	Description	Example
`--prefix`	Custom route prefix (default: `/api`)	`--prefix /api/v2`
`--middleware`	Add middleware to all routes	`--middleware auth_middleware`

---

6. tina4 generate migration -- Migration Scaffolding

The `generate migration` command creates a timestamped migration file with `UP` and `DOWN` sections. The timestamp ensures migrations run in order.

```
tina4 generate migration add_category_to_products
```

```
Created migrations/20260322120500_add_category_to_products.sql
```

The generated file:

```
-- UP
-- Add your forward migration SQL here

-- DOWN
-- Add your rollback migration SQL here
```

Fill in the SQL:

```
-- UP
ALTER TABLE products ADD COLUMN category TEXT DEFAULT '';

-- DOWN
ALTER TABLE products DROP COLUMN category;
```

The timestamp prefix (`20260322120500`) ensures migrations run in order. Each migration runs once. The framework tracks which ones have been applied.

When to Generate a Migration

Generate a new migration when you need to change the database schema after the initial model migration. Adding a column. Creating an index. Renaming a table. Each change gets its own migration file with a unique timestamp.

---

7. tina4 generate middleware -- Middleware Scaffolding

The `generate middleware` command creates a middleware function with the correct signature and a placeholder for your logic.

```
tina4 generate middleware rate_limit
```

```
Created src/middleware/rate_limit.py
```

The generated file:

```
async def rate_limit(request, response, next_handler):
    # Add your middleware logic here

    # Continue to the next middleware or route handler
    return await next_handler(request, response)

    # Or return early to block the request:
    # return response({"error": "Rate limit exceeded"}, 429)
```

The middleware is a named function. Reference it in route definitions:

```
@get("/api/data", middleware=["rate_limit"])
async def protected_data(request, response):
    return response({"data": "protected"})
```

---

8. Generate All at Once

Combine flags to generate multiple files in a single command:

```
tina4 generate model Product --with-route --with-migration
```

```
Created src/orm/product.py
Created src/routes/products.py
Created migrations/20260322120000_create_products_table.sql
```

Model. CRUD routes. Migration. All wired together. Ready to use.

---

9. tina4 doctor -- Health Check

The `doctor` command checks your project for common issues:

```
tina4 doctor
```

```
Tina4 Doctor -- Checking your project...

  [OK] Python 3.12.0 detected
  [OK] uv package manager found
  [OK] tina4_python package installed (v3.0.0)
  [OK] .env file exists
  [OK] Database connection: sqlite:///data/app.db
  [OK] Database is accessible
  [OK] src/routes/ directory exists (3 route files)
  [OK] src/orm/ directory exists (2 model files)
  [OK] src/templates/ directory exists (5 templates)
  [OK] src/public/ directory exists (static files served)
  [OK] tests/ directory exists (4 test files)
  [WARN] No migrations found in migrations/
  [OK] AI context: Claude Code detected, CLAUDE.md present
  [OK] .gitignore includes .env, data/, logs/

  12 checks passed, 1 warning, 0 errors
```

Doctor checks:

- Python version and package manager
- Tina4 package installation and version
- `.env` file existence and critical variables
- Database connectivity
- Directory structure
- Missing files or configurations
- AI tool context files
- Git configuration

The warnings give actionable advice. If your database is not configured, it tells you exactly what to add to `.env`.

---

10. tina4 test -- Running Tests

```
tina4 test
```

```
Running tests...

  ProductTest
    [PASS] test_create_product
    [PASS] test_load_product

  2 tests, 2 passed, 0 failed (0.12s)
```

This runs all tests in the `tests/` directory. See Chapter 18 for full testing documentation.

Test Options

```
tina4 test --file tests/test_product.py           # Specific file
tina4 test --file tests/test_product.py --method test_create  # Specific method
tina4 test --verbose                                # Show assertion details
```

---

11. tina4 routes -- Route Listing

See all registered routes in your project:

```
tina4 routes
```

```
Registered Routes:

  Method  Path                        Handler                     Middleware
  ------  --------------------------  --------------------------  ----------
  GET     /health                     health_check                -
  GET     /api/products               list_products               ResponseCache:300
  GET     /api/products/{product_id}  get_product                 -
  POST    /api/products               create_product              auth_middleware
  PUT     /api/products/{product_id}  update_product              auth_middleware
  DELETE  /api/products/{product_id}  delete_product              auth_middleware
  GET     /api/auth/login             -                           -
  POST    /api/auth/login             login                       -
  GET     /admin                      admin_dashboard             -

  9 routes registered
```

This is useful for verifying that your routes are registered and for finding the handler function for a specific URL.

Filtering

```
tina4 routes --method POST          # Filter by HTTP method
tina4 routes --filter products      # Filter by path pattern
tina4 routes --middleware auth      # Filter by middleware
```

When debugging routing issues, check here first. If a route does not match, `tina4 routes` shows whether it was registered and what middleware is attached.

---

12. tina4 migrate -- Database Migrations

Run pending migrations:

```
tina4 migrate
```

```
Running migrations...
  [UP] 20260322000100_create_users_table.sql
  [UP] 20260322000200_create_products_table.sql
  [UP] 20260322000300_add_category_to_products.sql

  3 migrations applied
```

Rollback

```
tina4 migrate --rollback
```

```
Rolling back last migration...
  [DOWN] 20260322000300_add_category_to_products.sql

  1 migration rolled back
```

Migration Table Auto-Upgrade

If your project was created with an earlier version of Tina4, the `tina4_migration` tracking table may use the older v2 schema. Running `tina4 migrate` detects the old layout and adds the missing `migration_id`, `batch`, and `executed_at` columns, backfilling existing data. No manual intervention needed.

Status

```
tina4 migrate --status
```

```
Migration Status:

  Status    Migration
  --------  ----------------------------------------
  Applied   20260322000100_create_users_table.sql
  Applied   20260322000200_create_products_table.sql
  Applied   20260322000300_add_category_to_products.sql
  Pending   20260322000400_create_orders_table.sql

  3 applied, 1 pending
```

---

13. Exercise: Scaffold a Feature in 5 Commands

Scaffold a complete "Customer" feature from scratch using only CLI commands.

Requirements

Starting from an existing Tina4 Python project, run 5 commands to create:

- A Customer ORM model with name, email, phone, and company fields
- A CRUD route file with all five REST endpoints
- A migration that creates the customers table
- Run the migration to create the table
- Run the doctor to verify everything

Expected Commands

```
# 1. Generate the model with route and migration
tina4 generate model Customer --with-route --with-migration

# 2. Edit the model to add fields (manual step)
# Add fields to src/orm/customer.py

# 3. Edit the migration to add columns (manual step)
# Add columns to the migration file

# 4. Run the migration
tina4 migrate

# 5. Run the doctor to verify everything is set up
tina4 doctor
```

Solution

Command 1: Generate everything:

```
tina4 generate model Customer --with-route --with-migration
```

Command 2: Edit `src/orm/customer.py`:

```
from tina4_python.orm import ORM

class Customer(ORM):
    table_name = "customers"

    id: int
    name: str
    email: str
    phone: str
    company: str
    created_at: str
```

Command 3: Edit the migration file:

```
-- UP
CREATE TABLE customers (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    name TEXT NOT NULL,
    email TEXT NOT NULL UNIQUE,
    phone TEXT,
    company TEXT,
    created_at TEXT DEFAULT CURRENT_TIMESTAMP
);

CREATE INDEX idx_customers_email ON customers(email);

-- DOWN
DROP TABLE IF EXISTS customers;
```

Command 4: Run the migration:

```
tina4 migrate
```

```
Running migrations...
  [UP] 20260322120000_create_customers_table.sql
```

```
  1 migration applied
```

Command 5: Verify with doctor:

```
tina4 doctor
```

```
  [OK] src/orm/ directory exists (3 model files)
  [OK] src/routes/ directory exists (4 route files)
  [OK] Database connection: sqlite:///data/app.db
  ...
```

Now test the API:

```
curl -X POST http://localhost:7146/api/customers \
  -H "Content-Type: application/json" \
  -d '{"name": "Alice Corp", "email": "alice@corp.com", "phone": "+1-555-0100", "company": "Alic
```

```
{
  "id": 1,
  "name": "Alice Corp",
  "email": "alice@corp.com",
  "phone": "+1-555-0100",
  "company": "Alice Corp",
  "created_at": "2026-03-22 12:00:00"
}
```

From zero to a working CRUD API. Five commands. Under two minutes.

---

14. Gotchas

1. tina4 Command Not Found

Problem: Running `tina4` gives "command not found".

Cause: The Tina4 CLI is not installed or not in your PATH.

Fix: Install the CLI: `curl -fsSL https://tina4.com/install.sh | sh`. Verify with `tina4 --version`. If installed but not found, add the installation directory to your PATH:

```
echo 'export PATH="$HOME/.tina4/bin:$PATH"' >> ~/.zshrc
source ~/.zshrc
```

2. Wrong Language Detected

Problem: `tina4 init` creates a PHP project instead of Python.

Cause: A `composer.json` file exists in the directory from a previous project.

Fix: Use explicit language selection: `tina4 init python my-project`. Or delete the conflicting language file before running `init`.

3. Generated Files Overwrite Existing Code

Problem: Running `tina4 generate route products` overwrites your custom route file.

Cause: The generate command creates files at fixed paths. If the file exists, it is overwritten.

Fix: The CLI warns you before overwriting. Check if the file exists first. If you need to regenerate, rename the existing file: `mv src/routes/products.py src/routes/products_backup.py`.

4. Migration Order Issues

Problem: A migration fails because it references a table that does not exist yet.

Cause: Migration files run in alphabetical (timestamp) order. If migration B depends on a table created by migration A, but A has a later timestamp, B runs first and fails.

Fix: Use consistent timestamps. The `tina4 generate migration` command uses the current timestamp. Generating migrations in order ensures correct execution order. If you need to fix ordering, rename the migration files to adjust their timestamps.

5. tina4 serve Uses Wrong Port

Problem: `tina4 serve` starts on port 7146 but you need port 8080.

Cause: The default port is 7146 unless overridden.

Fix: Set it in `.env`: `TINA4_PORT=8080`. Or pass it as a flag: `tina4 serve --port 8080`. The `.env` value takes precedence over the default. The command-line flag overrides everything.

6. Doctor Shows False Warnings

Problem: `tina4 doctor` warns about missing migrations, but your project does not use migrations.

Cause: Doctor checks for common conventions. It warns about missing migrations regardless of your approach.

Fix: These are warnings, not errors. Ignore warnings that do not apply to your project. Doctor is a guide, not a gatekeeper.

7. Generate Creates Python 3.12+ Syntax

Problem: Generated code uses `type: int` syntax that does not work in Python 3.10.

Cause: The generator targets the latest Python version supported by Tina4.

Fix: Use Python 3.12 or later with Tina4 Python. Check your Python version with `python --version`. If you must use an older version, modify the generated code to use compatible syntax.

8. Model Name Must Be PascalCase

Problem: `tina4 generate model order_item` creates a model class named `order_item` which is not valid Python convention.

Cause: The CLI uses the argument as-is for the class name.

Fix: Use PascalCase for model names: `tina4 generate model OrderItem`. The CLI converts it to snake_case for the table name (`order_items`).

---

15. Documentation

```
tina4 docs      # Download framework-specific book chapters to .tina4-docs/
tina4 books     # Download the complete Tina4 book (all languages) to tina4-book/
```

`tina4 docs` detects your project language and downloads only the relevant chapters. The documentation is available in Markdown format, optimised for AI tools and local reference.

---

16. Test Port (Dual-Port Development)

When `TINA4_DEBUG=true`, Tina4 automatically starts a second HTTP server on `port + 1000`:

- **Main port** (e.g. 7146) — hot-reload enabled, for AI dev tools
- **Test port** (e.g. 8145) — stable, no hot-reload, for user testing

This prevents the browser from refreshing mid-test when AI tools edit files.

Setting	Effect
`TINA4_NO_AI_PORT=true`	Disables the test port entirely
`TINA4_NO_RELOAD=true`	Disables hot-reload on the main port too
`--no-reload`	CLI flag equivalent of TINA4*NO*RELOAD

Vibe Coding with AI

Why Tina4 is Built for AI-Assisted Development

Tell your AI assistant: "Add a product catalog with search, pagination, and category filtering." In most frameworks, the AI needs to guess which packages to install, which config files to create, which naming conventions to follow. It hallucinates half of it.

With Tina4, the AI reads one file -- `CLAUDE.md` -- and knows everything. Every method signature. Every import path. Every convention. It generates correct, runnable code on the first try because there is only one way to do things in Tina4.

This is not accidental. Tina4 was designed from the ground up for AI-assisted development.

The Zero-Dependency Advantage

When an AI generates code for Django, it might suggest `from django.core.cache import cache` or `from django.views.decorators.cache import cache_page`. Both exist. Both work differently. The AI guesses.

Tina4 eliminates the guesswork. One cache. One queue. One ORM. One template engine. No ambiguity. No alternatives. No "it depends on which package you installed." The AI knows.

```
# There is only one way to cache in Tina4
from tina4_python.cache import cache_get
products = cache_get("products")

# There is only one way to queue in Tina4
from tina4_python.queue import Queue
queue = Queue(topic="emails")
queue.push({"to": "user@test.com"})

# There is only one way to send email in Tina4
from tina4_python.messenger import Messenger
mailer = Messenger()
mailer.send(to="user@test.com", subject="Welcome", body="Hello!")
```

Zero dependencies means zero confusion for AI.

CLAUDE.md -- The AI's Instruction Manual

Every Tina4 project includes a `CLAUDE.md` file that tells AI assistants exactly how the framework works. It contains:

- Every method signature with parameters and return types
- The project structure convention (src/routes/, src/orm/, src/templates/)
- The .env variable reference

- Code style rules (routes return `response()`, ORM uses `to_dict()`)
- Database connection string formats
- Common gotchas and how to avoid them

When you open a Tina4 project in Claude Code, Cursor, or GitHub Copilot -- the AI reads this file first and immediately understands how to write correct Tina4 code.

Skills -- Deep Framework Knowledge

Beyond CLAUDE.md, Tina4 ships with three AI skill files:

tina4-maintainer

For framework maintenance -- porting features between languages, reviewing PRs, running benchmarks, checking parity.

tina4-developer

For application development -- creating routes, defining models, writing templates, setting up auth, configuring queues.

tina4-js

For frontend development -- tina4-js signals, Tina4Element components, reactive templates, WebSocket client.

These skills are automatically installed when AI tools are detected:

```
# In your project root
from tina4_python.ai import install_selected, install_all

install_selected(".", "all")    # Detects + installs context for every supported tool
# install_all(".")                # Or: install context for ALL known tools, even those not dete
```

The Convention Advantage

AI thrives on convention. Every Tina4 project follows the same structure. The AI never asks "where should I put this?"

```
src/
  routes/hello.py       -> AI knows: this is a route file
  orm/product.py        -> AI knows: this is an ORM model
  templates/home.html   -> AI knows: this is a template
  middleware/auth.py     -> AI knows: this is middleware
```

Tell the AI "add a User model" and it creates `src/orm/user.py` with the correct field definitions, the correct class structure, and the correct method stubs. Every time.

Real-World Vibe Coding Session

Here is an actual conversation with Claude Code in a Tina4 Python project:

You: "Add a blog with posts and comments. Posts belong to users. Comments belong to posts. I want a REST API and a template page that lists posts."

Claude Code generates:

- `src/orm/post.py` -- Model with relationships to user and comments
- `src/orm/comment.py` -- Model with relationship to post
- `src/routes/api/posts.py` -- Full CRUD API (list, get, create, update, delete)
- `src/routes/api/comments.py` -- Nested comments API
- `src/routes/blog.py` -- Template route for the blog page
- `src/templates/blog.html` -- Page with tina4css cards for each post
- `migrations/20260322_create_posts_and_comments.sql` -- Database schema

All correct. All runnable. First try.

You: "Add pagination to the posts list"

Claude adds `request.params.get("page", 1)` handling and `to_dict()` with limit/offset -- because CLAUDE.md told it exactly how Tina4 pagination works.

You: "Cache the post list for 5 minutes"

Claude adds `cache_get(f"posts_page_{page}")` with `cache_set(f"posts_page_{page}", data, ttl=300)` -- because it knows the cache API.

You: "Send an email when a new comment is posted"

Claude adds `mailer = Messenger(); mailer.send(...)` -- because it knows Messenger reads from .env.

No research. No Stack Overflow. No package debates. Describe the intent. The AI builds it.

Python-Specific Patterns the AI Knows

The CLAUDE.md file teaches AI assistants the correct Python patterns for Tina4:

Route Decorators

```
from tina4_python.core.router import get, post, put, delete, template

@get("/api/products")
async def list_products(request, response):
    products = Product().select("*")
    return response({"products": [p.to_dict() for p in products]})

@post("/api/products")
async def create_product(request, response):
    body = request.body
    product = Product()
    product.name = body["name"]
    product.price = body["price"]
    product.save()
    return response(product.to_dict(), 201)
```

Template Rendering

```
@get("/products")
async def products_page(request, response):
    products = Product.all(order_by="name ASC")
    return response.render("products.html", {"products": [p.to_dict() for p in products]})
```

Middleware Attachment

```
from tina4_python.core.router import get, middleware
from src.app.middleware import AuthMiddleware

@middleware(AuthMiddleware)
@get("/api/profile")
async def get_profile(request, response):
    user = User.find(request.user_id)
    return response(user.to_dict()) if user else response({"error": "User not found"}, 404)
```

`@middleware(...)` is a separate decorator stacked above the route decorator -- it is not a keyword argument on `@get`/`@post`.

WebSocket Handlers

```
from tina4_python.websocket import websocket_route

@websocket_route("/ws/updates")
async def updates_handler(connection):
    async for message in connection:
        await connection.broadcast(message)
```

The AI generates all of these patterns correctly because CLAUDE.md specifies:

- Handlers are always `async def`
- Route handlers always receive `(request, response)`
- WebSocket handlers receive a single `connection` object -- iterate it with `async for` to receive frames
- Responses use `response()` with the data as the first argument
- Templates render via `response.render("name.html", {...})`

- Middleware stacks above the route decorator via `@middleware(MiddlewareClass)`

Supported AI Tools

Tina4 auto-detects and installs context for eight tools. The exact list is in `tina4_python/ai/__init__.py` (`AI_TOOLS`):

Tool	Detection (context file)	Context installed
Claude Code	`CLAUDE.md`	CLAUDE.md + `.claude/` skills
Cursor	`.cursorules`	`.cursorules` (+ `.cursor/`)
GitHub Copilot	`.github/copilot-instructions.md`	`.github/copilot-instructions.md`
Windsurf	`.windsurfrules`	`.windsurfrules`
Aider	`CONVENTIONS.md`	`CONVENTIONS.md`
Cline	`.clinerules`	`.clinerules`
OpenAI Codex	`AGENTS.md`	`AGENTS.md`
Google Antigravity	`.antigravity/context.md`	`.antigravity/context.md`

Run `tina4 ai` to see which tools are detected in your project and install context files via the menu.

The AI Chat in Dev Dashboard

The dev dashboard at `/__dev` includes an AI chat tab. By default it talks to a local **qwen2.5-coder** model served via Ollama, grounded by Tina4's built-in RAG index of the framework source. Nothing leaves your machine.

Configure it in `.env`:

```
TINA4_AI_URL=http://localhost:11434      # Ollama HTTP endpoint
TINA4_AI_MODEL=qwen2.5-coder             # Default coding model
TINA4_RAG_URL=http://localhost:11434     # RAG embedding endpoint (defaults to TINA4_AI_URL)
```

If you prefer a remote provider, point `TINA4_AI_URL` at any OpenAI-compatible endpoint (LM Studio, vLLM, an OpenAI-compatible proxy, etc.) and set `TINA4_AI_MODEL` to the model name that endpoint serves.

The AI has full context of your Tina4 project -- routes, models, templates, configuration. Ask it:

- "Why is my /api/users route returning 500?"

- "How do I add WebSocket support?"
- "Write a migration to add an 'avatar' column to users"

The "Ask about this error" button on the error overlay sends the full stack trace and source code to the AI for instant debugging.

Tips for Effective Vibe Coding with Tina4

- **Be specific about what you want** -- "Add a product API with search by name and price range" works better than "add products"
- **Let the AI use tina4 generate** -- "Run tina4 generate model Product then add price and category fields" gives the AI a starting point
- **Reference the .env** -- "Configure the queue to use RabbitMQ" -- the AI knows to update .env with `TINA4_QUEUE_BACKEND=rabbitmq`
- **Ask for tests** -- "Write tests for the Product API" -- the AI uses Tina4's inline testing framework
- **Ask for the gallery** -- "Deploy the auth gallery example" -- the AI clicks Try It and you have a working JWT demo
- **Review, don't rewrite** -- The AI's first attempt is usually 90% correct. Tweak the details instead of starting over.

Exercise

Open your Tina4 Python project in Claude Code (or your preferred AI tool) and try these prompts:

- "Add a Contact model with name, email, phone, and message fields"
- "Create a contact form page at /contact with an HTML template using tina4css"
- "Add an API endpoint that accepts the form submission and saves it to the database"
- "Send an email notification when a new contact is submitted"
- "Add rate limiting to the contact form -- max 3 submissions per minute"

Each prompt should generate correct, runnable code on the first try. If it does not, check that CLAUDE.md is in your project root.

Gotchas

1. No CLAUDE.md?

Problem: The AI generates incorrect import paths or uses wrong API patterns.

Fix: Run `tina4 ai` (or `from tina4_python.ai import install_all; install_all(".")`) to generate the CLAUDE.md file. This gives the AI complete framework knowledge.

2. AI Suggests Wrong Import Paths

Problem: The AI writes `from tina4 import Router` instead of `from tina4_python.core.router import get`.

Fix: The CLAUDE.md might be outdated. Regenerate it with `install_all(".", force=True)`. Check that the file includes the correct import paths for Python.

3. AI Hallucinates a Package

Problem: The AI suggests `pip install tina4-cache-redis` or similar packages that do not exist.

Fix: Remind it: "Tina4 has zero dependencies. Use only built-in features." Everything -- caching, queues, email, WebSocket, GraphQL -- is part of the framework.

4. AI Creates Files in Wrong Location

Problem: The AI creates routes in `routes/` instead of `src/routes/`.

Fix: Tell it: "Follow the src/routes, src/orm, src/templates convention." The CLAUDE.md file specifies this, but some AI tools need a reminder.

5. AI Code Does Not Run

Problem: The generated code has syntax errors or does not follow Tina4 patterns.

Fix: Check that route handlers use `async def` and return `response()`. Check that ORM models extend `ORM`. Check that templates use `template()` for rendering. These are the most common mistakes.

6. AI Uses Synchronous Code

Problem: The AI generates `def handler(request, response)` instead of `async def handler(request, response)`.

Fix: All Tina4 Python route handlers must be `async def`. Tell the AI: "All handlers are async in Tina4 Python."

7. AI Suggests Flask/Django Patterns

Problem: The AI writes `@app.route("/products")` or `return JsonResponse(data)`.

Fix: The AI is falling back to its general Python web framework knowledge. Point it to CLAUDE.md: "Read the CLAUDE.md file for Tina4 Python patterns. Do not use Flask or Django patterns."

The Philosophy

Tina4 was not made compatible with AI coding tools. It was designed for them.

Convention over configuration: the AI knows where things go. Zero dependencies: the AI never chooses between packages. A single CLAUDE.md: the AI has complete framework knowledge. Identical APIs across 4 languages: the AI's knowledge transfers instantly.

You describe the intent. The AI writes the code. You review and refine. That is the workflow Tina4 was built for.

The Intelligent Native Application 4ramework. Built for AI. Built for you.

Environment Variables

> **■■ *BREAKING CHANGE — Tina4 v3.12.0***
>
> *Every framework env var now requires the `TINA4_` prefix. The legacy un-prefixed names (`DATABASE_URL`, `SECRET`, `SMTP_HOST`, `HOST_NAME`, etc.) no longer work. Setting them at startup makes the framework refuse to boot with a list of renames.*
>
> *Run `tina4 env --migrate` to rewrite your existing `.env` automatically, or rename manually using the table below. The runtime guard prints the same mapping if it detects legacy names.*
>
> ***Conventional names stay un-prefixed:*** `PORT`, `HOST`, `NODE_ENV`, `RACK_ENV`, `RUBY_ENV`, `ENVIRONMENT`. *These are runtime/PaaS conventions, not framework config.*

Tina4 Python is configured through environment variables, read from `.env` at the project root. Every variable has a sensible default — most projects set three or four values and leave the rest alone.

This chapter lists every variable the Python framework reads, grouped by subsystem. Start with the minimum-config examples at the end, then come back here when you need to tune something specific.

Core Server

Variable	Default	Description
`HOST`	`0.0.0.0`	Bind address. `0.0.0.0` listens on every interface. `127.0.0.1` restricts to localhost.
`TINA4_HOST`	*(inherits `HOST`)*	Explicit Tina4-specific bind address override. Takes precedence over `HOST` when both are set.
`PORT`	`7146`	HTTP server port. The Rust CLI prefers `TINA4_PORT` but falls back to `PORT`.
`TINA4_PORT`	*(inherits `PORT`)*	Explicit Tina4-specific port override. Takes precedence over `PORT` when both are set.
`TINA4_HOST_NAME`	`localhost:7146`	Fully-qualified host used in generated absolute URLs (Swagger, OAuth redirects, emails).
`TINA4_DEBUG`	`false`	Master debug toggle. Enables Swagger UI, dev dashboard, live reload, template dump filter, error overlay. Never set to `true` in production.
`TINA4_ENV`	`development`	Runtime environment label. Values like `development`, `staging`, `production` control dev-only features.
`TINA4_ENV_FILE`	`.env`	Alternative `.env` path. Read at boot before any other framework config — point at `.env.staging` or `.env.production` to switch the whole config tree.

`TINA4_SUPPRESS`	`false`	Suppresses the framework startup banner. Useful in CI runs or systemd units where stdout is parsed by another process.
`TINA4_HEALTH_PATH`	`/__health`	URL for the built-in liveness/readiness endpoint. The legacy `/health` path is kept as an alias.
`TINA4_NO_BROWSER`	`false`	Stops `tina4 serve` from opening your browser on every restart. Recommended during active development.
`TINA4_OPEN_BROWSER`	`true`	Alternative flag — set to `false` to prevent the browser opening on start.
`TINA4_NO_RELOAD`	`false`	Disables the dev hot-reload signal from the Rust CLI. Use when you want a stable server for debugging.
`TINA4_DEV_POLL_INTERVAL`	`1.0`	Seconds between dev-mode mtime polls. Lower for faster reload, higher to reduce CPU.
`TINA4_PUBLIC_DIR`	`src/public`	Directory served as static files under `/`.

Secrets and Authentication

Variable	Default	Description
`TINA4_SECRET`	`tina4-default-secret`	JWT signing secret. Must be long, random, and unique per environment. **Never commit.**
`TINA4_TOKEN_LIMIT`	`60`	JWT token lifetime in minutes.
`TINA4_TOKEN_EXPIRES_IN`	*(inherits `TOKEN_LIMIT`)*	Alias for `TINA4_TOKEN_LIMIT`.
`TINA4_API_KEY`	*(empty)*	Static API key used by `Auth.validate_api_key()` as a fallback to JWT.
`TINA4_API_KEY`	*(empty)*	Legacy alias for `TINA4_API_KEY`.

Database

Variable	Default	Description
`TINA4_DATABASE_URL`	`sqlite:///data/app.db`	Connection URL. Scheme selects the driver: `sqlite`, `postgres`, `mysql`, `firebird`.
`TINA4_DATABASE_USERNAME`	*(empty)*	Overrides the username embedded in `TINA4_DATABASE_URL`.
`TINA4_DATABASE_PASSWORD`	*(empty)*	Overrides the password embedded in `TINA4_DATABASE_URL`.
`TINA4_DATABASE_FIREBIRD_PATH`	*(empty)*	Overrides the database path/alias parsed from `TINA4_DATABASE_URL` for Firebird. Useful for Windows backslash paths and split-config setups.
`TINA4_DB_CACHE`	`false`	Enables in-memory query-result caching for read queries.
`TINA4_DB_CACHE_TTL`	`60`	Query cache TTL in seconds when `TINA4_DB_CACHE=true`.
`TINA4_DB_POOL`	`0`	Default `pool` size for `Database(url)` when the caller doesn't pass `pool=` explicitly. `0` disables pooling and uses a single connection. Set to a positive integer (e.g. `4`) to enable round-robin connection pooling.
`TINA4_ORM_PLURAL_TABLE_NAMES`	`true`	When `true`, the ORM pluralises class names into table names (`User` → `users`). Set `false` to keep them singular.

CORS

Variable	Default	Description
`TINA4_CORS_ORIGINS`	`*`	Comma-separated allowed origins. Lock down to real domains in production.
`TINA4_CORS_MAX_AGE`	`600`	Preflight cache lifetime in seconds.

Routing

Variable	Default	Description
`TINA4_TRAILING_SLASH_REDIRECT`	`false`	When truthy, requests to `/foo/` are 301-redirected to `/foo` so search engines and clients only see one canonical URL per route. The root `/` is exempt.

Security Headers

Variable	Default	Description
`TINA4_CSP`	`default-src 'self'`	`Content-Security-Policy` header.
`TINA4_CSRF`	`true`	CSRF token validation on POST/PUT/PATCH/DELETE.
`TINA4_HSTS`	*(empty/off)*	`Strict-Transport-Security` max-age in seconds. Set `31536000` in production with HTTPS.
`TINA4_FRAME_OPTIONS`	`DENY`	`X-Frame-Options` header.
`TINA4_REFERRER_POLICY`	`strict-origin-when-cross-origin`	`Referrer-Policy` header.

Rate Limiting

Variable	Default	Description
`TINA4_RATE_LIMIT`	`100`	Maximum requests per window per IP. Set `0` to disable.
`TINA4_RATE_WINDOW`	`60`	Rate-limit window in seconds.

Sessions

Variable	Default	Description
`TINA4_SESSION_BACKEND`	`file`	Storage backend. Options: `file`, `redis`, `valkey`, `mongo`, `database`.
`TINA4_SESSION_TTL`	`1800`	Session expiry in seconds (30 minutes).
`TINA4_SESSION_SAMESITE`	`Lax`	SameSite cookie attribute. Options: `Strict`, `Lax`, `None`.
`TINA4_SESSION_PATH`	`data/sessions`	Filesystem path for the file backend.
`TINA4_SESSION_NAME`	`tina4_session`	Name of the session cookie sent to the browser.
`TINA4_SESSION_HTTPONLY`	`true`	Sets the `HttpOnly` cookie attribute so JavaScript on the page cannot read the session ID.
`TINA4_SESSION_SECURE`	`false`	Sets the `Secure` cookie attribute so the session cookie is only sent over HTTPS. Turn on in production.

Redis/Valkey session backend

Variable	Default	Description
`TINA4_SESSION_REDIS_HOST`	`localhost`	Redis host.
`TINA4_SESSION_REDIS_PORT`	`6379`	Redis port.
`TINA4_SESSION_REDIS_PASSWORD`	*(none)*	Redis auth password.
`TINA4_SESSION_REDIS_DB`	`0`	Redis database number.
`TINA4_SESSION_VALKEY_HOST`	`localhost`	Valkey host.
`TINA4_SESSION_VALKEY_PORT`	`6379`	Valkey port.
`TINA4_SESSION_VALKEY_PASSWORD`	*(none)*	Valkey auth password.
`TINA4_SESSION_VALKEY_DB`	`0`	Valkey database number.

MongoDB session backend

Variable	Default	Description
`TINA4_SESSION_MONGO_URL`	`mongodb://localhost:27017`	MongoDB connection string.
`TINA4_SESSION_MONGO_DB`	`tina4`	MongoDB database name.
`TINA4_SESSION_MONGO_COLLECTION`	`sessions`	MongoDB collection name.

Templates

Variable	Default	Description
`TINA4_TEMPLATE_CACHE_TTL`	`0`	Frond template compile-cache TTL in seconds. `0` keeps compiled templates in memory permanently — set to a positive value if you want the engine to recompile after N seconds (rarely needed; `tina4 serve` invalidates the cache automatically on file change).

Cache

Variable	Default	Description
`TINA4_CACHE_BACKEND`	`memory`	Response cache backend. Options: `memory`, `file`, `redis`.
`TINA4_CACHE_DIR`	`data/cache`	Cache directory for the file backend.
`TINA4_CACHE_TTL`	`60`	Default cache TTL in seconds.
`TINA4_CACHE_MAX_ENTRIES`	`1000`	Maximum cache entries. Oldest entries evicted first.
`TINA4_CACHE_URL`	`redis://localhost:6379`	Connection URL for remote cache backends.

Queues

Variable	Default	Description
`TINA4_QUEUE_BACKEND`	`lite`	Queue backend. Options: `lite` (file-based), `kafka`, `rabbitmq`, `mongo`, `database`.
`TINA4_QUEUE_PATH`	`data/queue`	Filesystem path for the lite backend.
`TINA4_QUEUE_URL`	*(none)*	Connection URL for remote backends.

Kafka queue backend

Variable	Default	Description
`TINA4_KAFKA_BROKERS`	`localhost:9092`	Comma-separated broker list.
`TINA4_KAFKA_GROUP_ID`	`tina4_consumer_group`	Kafka consumer group ID.

RabbitMQ queue backend

Variable	Default	Description
`TINA4_RABBITMQ_HOST`	`localhost`	RabbitMQ host.
`TINA4_RABBITMQ_PORT`	`5672`	RabbitMQ port.
`TINA4_RABBITMQ_USERNAME`	`guest`	RabbitMQ username.
`TINA4_RABBITMQ_PASSWORD`	`guest`	RabbitMQ password.
`TINA4_RABBITMQ_VHOST`	`/`	RabbitMQ virtual host.

MongoDB queue backend

Variable	Default	Description
`TINA4_MONGO_URI`	*(none)*	Full MongoDB connection string. Overrides host/port when set.
`TINA4_MONGO_HOST`	`localhost`	MongoDB host.
`TINA4_MONGO_PORT`	`27017`	MongoDB port.
`TINA4_MONGO_USERNAME`	*(none)*	MongoDB username.
`TINA4_MONGO_PASSWORD`	*(none)*	MongoDB password.
`TINA4_MONGO_DB`	`tina4`	MongoDB database name.
`TINA4_MONGO_COLLECTION`	`tina4_queue`	MongoDB collection name for jobs.

WebSocket

Variable	Default	Description
`TINA4_WS_BACKPLANE`	*(none)*	Backplane type. Set `redis` for multi-instance broadcasts.
`TINA4_WS_MAX_CONNECTIONS`	`1000`	Maximum concurrent WebSocket connections.
`TINA4_WS_MAX_FRAME_SIZE`	`65536`	Maximum WebSocket frame size in bytes.

Email

Variable	Default	Description
`TINA4_MAIL_HOST`	*(none)*	SMTP server hostname.
`TINA4_MAIL_PORT`	`587`	SMTP server port.
`TINA4_MAIL_USERNAME`	*(none)*	SMTP authentication username.
`TINA4_MAIL_PASSWORD`	*(none)*	SMTP authentication password.
`TINA4_MAIL_FROM`	*(none)*	Default sender email address.
`TINA4_MAIL_FROM_NAME`	*(none)*	Default sender display name.
`TINA4_MAIL_ENCRYPTION`	`tls`	Connection encryption. Options: `tls`, `ssl`, `none`.
`TINA4_MAIL_IMAP_HOST`	*(none)*	IMAP server for inbound mail.
`TINA4_MAIL_IMAP_PORT`	`993`	IMAP server port.
`TINA4_MAIL_IMAP_ENCRYPTION`	`tls`	IMAP transport encryption. Options: `tls`, `starttls`, `none`. Invalid values fall back to `tls` so a typo can't accidentally disable encryption.
`TINA4_MAILBOX_DIR`	`data/mailbox`	Dev mailbox directory. All outbound mail lands here when `TINA4_DEBUG=true`.

> `TINA4_MAIL_HOST`, `TINA4_MAIL_PORT`, `TINA4_MAIL_USERNAME`, `TINA4_MAIL_PASSWORD`, `TINA4_MAIL_FROM`, `TINA4_MAIL_FROM_NAME`, `TINA4_MAIL_IMAP_HOST`, `TINA4_MAIL_IMAP_PORT` *are accepted as legacy aliases. New projects should use the* `TINA4_MAIL_*` *names.*

Logging

Logs default to stdout in `text` format. Set `TINA4_LOG_OUTPUT=file` plus `TINA4_LOG_FILE=app.log` to write to disk; the framework rotates at `TINA4_LOG_ROTATE_SIZE` bytes and keeps `TINA4_LOG_ROTATE_KEEP` backups (`app.log.1` … `app.log.N`). Switch `TINA4_LOG_FORMAT=json` for one structured record per line — perfect for shipping to Loki, Datadog, or any JSON-aware log aggregator.

Variable	Default	Description
`TINA4_LOG_LEVEL`	`ERROR`	Minimum log level written to files. Options: `ALL`, `DEBUG`, `INFO`, `WARNING`, `ERROR`.
`TINA4_LOG_FILE`	*(empty — stdout only)*	Path to a log file. Empty leaves logs on stdout. Relative paths are resolved against `TINA4_LOG_DIR`; absolute paths are used verbatim.
`TINA4_LOG_DIR`	`logs`	Directory for log files. Joined with `TINA4_LOG_FILE` when the latter is a relative path.
`TINA4_LOG_FORMAT`	`text`	Output format. `text` writes the human-readable `[INFO] message` form; `json` writes one structured JSON record per line.
`TINA4_LOG_OUTPUT`	`stdout`	Where logs go. Options: `stdout`, `file`, `both`.
`TINA4_LOG_CRITICAL`	`false`	Enables the `Log.critical(...)` level above `error`. When off, calls to `Log.critical()` are silent no-ops.
`TINA4_LOG_ROTATE_SIZE`	`10485760`	Bytes per file before rotation (default 10 MB). `0` disables rotation entirely.
`TINA4_LOG_ROTATE_KEEP`	`5`	Number of rotated files to keep (`app.log.1` … `app.log.N`). Older files are deleted on the next rotation.
`TINA4_LOG_MAX_SIZE`	`10485760`	Legacy alias for `TINA4_LOG_ROTATE_SIZE`. Per-file log size limit in bytes (10 MB). Rotated when exceeded.

`TINA4_LOG_KEEP`	`5`	Legacy alias for `TINA4_LOG_ROTATE_KEEP`. Number of rotated log files to retain.

Uploads

Variable	Default	Description
`TINA4_MAX_UPLOAD_SIZE`	`10485760`	Maximum multipart upload size in bytes (10 MB).

Localisation

Variable	Default	Description
`TINA4_LOCALE`	`en`	Default locale for the I18n module.
`TINA4_LOCALE_DIR`	`src/locale`	Directory containing locale JSON files.

Services (background tasks)

Variable	Default	Description
`TINA4_SERVICE_DIR`	`src/services`	Directory scanned for service classes.

AI and MCP Tooling

The dashboard AI chat and the framework's RAG-based code search both default to a **local qwen2.5-coder model served via Ollama**. Nothing leaves your machine unless you point `TINA4_AI_URL` at a remote endpoint.

Variable	Default	Description

`TINA4_AI_URL`	`http://localhost:11434`	OpenAI-compatible HTTP endpoint for the chat/completion model. Ollama by default; can point at any compatible provider (vLLM, LM Studio, an OpenAI proxy).
`TINA4_AI_MODEL`	`qwen2.5-coder`	Model identifier the endpoint should serve.
`TINA4_RAG_URL`	*(inherits* `TINA4_AI_URL`*)*	Embedding endpoint for the framework RAG index. Override only if embeddings live on a different host.
`TINA4_AI_MODEL`	`nomic-embed-text`	Embedding model used to index `tina4_python/` and `src/`.
`TINA4_NO_AI_PORT`	`false`	Disables the MCP port listener in dev mode.
`TINA4_MCP_REMOTE`	`false`	Allow the MCP server to bind on non-localhost interfaces. **Never enable in production.**
`TINA4_OVERRIDE_CLIENT`	`false`	Allow the framework to start without the Rust CLI (`tina4 serve`). Used in Docker images and CI runners; bypasses SCSS compilation, the file watcher, and live reload.

Swagger / OpenAPI

Variable	Default	Description
`TINA4_SWAGGER_TITLE`	`Tina4 API`	OpenAPI spec title.
`TINA4_SWAGGER_DESCRIPTION`	*(empty)*	OpenAPI spec description.
`TINA4_SWAGGER_VERSION`	`1.0.0`	OpenAPI spec version.
`TINA4_SWAGGER_ENABLED`	*(inherits `TINA4_DEBUG`)*	Toggle the Swagger UI on or off independently of debug mode. Set `true` in production to keep API docs available without enabling the rest of the dev surface.
`TINA4_SWAGGER_CONTACT_EMAIL`	*(empty)*	Contact email rendered in the OpenAPI spec's `info.contact.email` field.
`TINA4_SWAGGER_LICENSE`	*(empty)*	License name in the OpenAPI spec's `info.license.name` field (e.g. `MIT`, `Apache-2.0`).

GraphQL

Variable	Default	Description
`TINA4_GRAPHQL_ENDPOINT`	`/graphql`	URL path the GraphQL handler is mounted on. POST serves queries, GET serves the GraphiQL IDE.
`TINA4_GRAPHQL_AUTO_SCHEMA`	`true`	When `true`, the framework auto-builds the GraphQL schema from every registered ORM model on boot. Set `false` if you want to define the schema manually with `gql.schema.add_type(...)`.

MCP (Model Context Protocol)

Variable	Default	Description
`TINA4_MCP`	*(inherits `TINA4_DEBUG`)*	Toggle the built-in MCP dev-tools server. Set explicitly to keep the MCP endpoint exposed in a debug-disabled deployment (e.g. for a remote AI assistant).
`TINA4_MCP_PORT`	*(framework port + 2000)*	TCP port for the MCP server. The default offset keeps it clear of both the main server (default `7145`) and the AI test port (`+1000`).

Minimal `.env` for Development

```
TINA4_DEBUG=true
TINA4_LOG_LEVEL=DEBUG
TINA4_NO_BROWSER=true
```

Debug mode lights up the Swagger UI, the dev dashboard, detailed error pages, and live reload. Keeping the browser flag on stops a new tab opening every time you save a file.

Minimal `.env` for Production

```
TINA4_SECRET=your-long-random-secret-here
TINA4_DATABASE_URL=postgresql://user:password@db-host:5432/myapp
TINA4_CORS_ORIGINS=https://myapp.com,https://www.myapp.com
TINA4_HSTS=31536000
TINA4_MAIL_HOST=smtp.example.com
TINA4_MAIL_PORT=587
TINA4_MAIL_USERNAME=noreply@myapp.com
TINA4_MAIL_PASSWORD=your-smtp-password
TINA4_MAIL_FROM=noreply@myapp.com
```

No `TINA4_DEBUG`. It defaults to `false`, which is what you want in production. Set a real secret, a real database, locked-down CORS origins, HSTS, and SMTP credentials if you send email. Everything else has a production-appropriate default.

Deployment

1. From Development to Production

The app works on `localhost:7146`. Now it needs to run around the clock on a real server. Handle thousands of concurrent users. Survive restarts. Hold steady on memory. The gap between "works on my machine" and "works in production" is where projects stumble.

This chapter covers everything for a production deployment: environment configuration, ASGI server setup, Docker packaging, health checks, graceful shutdown, SSL/TLS, scaling, and monitoring.

When you run `tina4 init`, the framework generates a production-ready `Dockerfile` and `.dockerignore` in your project root. The Dockerfile uses a multi-stage build: the first stage installs dependencies and the second stage copies only runtime artifacts into a slim image. You do not need to write a Dockerfile from scratch -- the generated one is a solid starting point.

---

2. Production .env Configuration

Development defaults optimize for debugging. Production defaults optimize for performance and security. The first deployment step: configure `.env` for production.

Create a production `.env`:

```
# Core
TINA4_DEBUG=false
TINA4_LOG_LEVEL=WARNING
TINA4_PORT=7146

# Database
TINA4_DATABASE_URL=sqlite:///data/app.db

# Security
CORS_ORIGINS=https://yourdomain.com
JWT_SECRET=your-long-random-secret-at-least-32-characters
TINA4_RATE_LIMIT=120

# Performance
TINA4_TEMPLATE_CACHE_TTL=true
```

Key Differences from Development

Setting	Development	Production	Why
`TINA4_DEBUG`	`true`	`false`	Hides stack traces, disables toolbar
`TINA4_LOG_LEVEL`	`ALL`	`WARNING`	Reduces log noise
`CORS_ORIGINS`	`*`	Your domain	Prevents cross-origin abuse

Sensitive Values

Production secrets never go into version control. The `.env` file is gitignored by default. For deployment, use environment variables from your hosting platform, CI/CD secrets, or a secrets manager.

```
# Docker: pass env vars at runtime
docker run -e JWT_SECRET=your-secret -e TINA4_DATABASE_URL=sqlite:///data/app.db my-app

# Fly.io: set secrets
fly secrets set JWT_SECRET=your-secret

# Railway: use the dashboard or CLI
railway variables set JWT_SECRET=your-secret
```

---

3. ASGI Server Auto-Detection

Tina4 Python auto-detects ASGI servers at startup. ASGI servers deliver production-grade performance:

- Multiple worker processes
- Async request handling
- Graceful shutdown
- Better error recovery

How Auto-Detection Works

When you run `tina4 serve` or `tina4 serve --production`, Tina4 checks for installed ASGI servers:

- If `uvicorn` is installed, the framework uses uvicorn with optimal settings
- If `hypercorn` is installed, the framework uses hypercorn
- If neither is installed, the framework falls back to the built-in development server

```
# Install uvicorn for production
uv add uvicorn

# With uvicorn installed
tina4 serve

  Tina4 Python v3.0.0
  Server: uvicorn (4 workers)
  Running at http://0.0.0.0:7146

# Without uvicorn
tina4 serve

  Tina4 Python v3.0.0
  Server: built-in (development)
  Running at http://0.0.0.0:7146
  WARNING: Do not use the built-in server in production
```

Uvicorn Configuration

Fine-tune uvicorn through environment variables:

Or pass options directly:

```
uvicorn app:app --workers 4 --host 0.0.0.0 --port 7146
```

How Many Workers?

Start with `(2 * CPU cores) + 1`:

```
import multiprocessing
workers = (2 * multiprocessing.cpu_count()) + 1
```

CPU Cores	Workers	Use Case
1	3	Small VPS, hobbyist
2	5	Small production app
4	9	Medium production app
8	17	High-traffic app

---

4. Docker Deployment

Docker is the most portable deployment path. Your app runs the same way on your laptop, in CI, and on the production server.

Official Base Image

Tina4 Python provides an official Docker Hub base image: `tina4stack/tina4-python:v3`. It is a lean, Alpine-based image (~56MB) with Python 3.13, SQLite, and the Tina4 framework pre-installed. Your app Dockerfile extends it and adds only your application code.

The base image includes these environment variables pre-configured:

- `TINA4_OVERRIDE_CLIENT=true` — bypasses the CLI guard for Docker
- `TINA4_DEBUG=false` — production mode by default
- `PYTHONUNBUFFERED=1` — ensures logs appear in `docker logs`
- `TINA4_NO_BROWSER=true` — prevents browser auto-open attempts
- `HOST=0.0.0.0` and `PORT=7146` — server binds to all interfaces

Dockerfile

Create `Dockerfile`:

```
FROM tina4stack/tina4-python:v3
WORKDIR /app

# Copy application code
COPY app.py .
```

```
COPY .env .
COPY migrations/ migrations/
COPY src/ src/

# Create data directories for SQLite, sessions, queue, and mailbox
RUN mkdir -p data data/sessions data/queue data/mailbox

EXPOSE 7146
CMD ["python", "app.py"]
```

That is the entire Dockerfile. The base image handles Python, dependencies, and framework setup.

.dockerignore

Create `.dockerignore`:

```
.venv
__pycache__
*.pyc
data/
tests/
.tina4/
.DS_Store
*.db
*.db-wal
*.db-shm
logs/
.git
.env.development
.pytest_cache
htmlcov
.claude
```

Building and Running

```
# Build the image
docker build -t my-tina4-app .

# Run the container
docker run -d \
  --name my-app \
  -p 7146:7146 \
  -e JWT_SECRET=your-production-secret \
  -e TINA4_DATABASE_URL=sqlite:///data/app.db \
  -v $(pwd)/data:/app/data \
  my-tina4-app
```

Adding Database Drivers

The base image ships with SQLite only. To use PostgreSQL, MySQL, MSSQL, or Firebird, install the driver in your Dockerfile.

PostgreSQL:

```
FROM tina4stack/tina4-python:v3
WORKDIR /app
RUN python -m pip install --no-cache-dir psycopg2-binary
COPY app.py .
COPY .env .
```

```
COPY migrations/ migrations/
COPY src/ src/
RUN mkdir -p data data/sessions data/queue data/mailbox
EXPOSE 7146
CMD ["python", "app.py"]
```

MySQL:

```
FROM tina4stack/tina4-python:v3
WORKDIR /app
RUN apk add --no-cache mariadb-connector-c-dev && \
    python -m pip install --no-cache-dir mysqlclient
COPY app.py .
COPY .env .
COPY migrations/ migrations/
COPY src/ src/
RUN mkdir -p data data/sessions data/queue data/mailbox
EXPOSE 7146
CMD ["python", "app.py"]
```

MSSQL:

```
FROM tina4stack/tina4-python:v3
WORKDIR /app
RUN apk add --no-cache unixodbc-dev freetds-dev && \
    python -m pip install --no-cache-dir pymssql
COPY app.py .
COPY .env .
COPY migrations/ migrations/
COPY src/ src/
RUN mkdir -p data data/sessions data/queue data/mailbox
EXPOSE 7146
CMD ["python", "app.py"]
```

Firebird:

```
FROM tina4stack/tina4-python:v3
WORKDIR /app
# Pure Python driver — no system dependencies needed
RUN python -m pip install --no-cache-dir firebird-driver
COPY app.py .
COPY .env .
COPY migrations/ migrations/
COPY src/ src/
RUN mkdir -p data data/sessions data/queue data/mailbox
EXPOSE 7146
CMD ["python", "app.py"]
```

Docker Compose

For a complete setup with supporting services, use Docker Compose.

Create `docker-compose.yml`:

```
services:
  app:
    build: .
    ports:
      - "7146:7146"
    environment:
      - TINA4_DEBUG=false
```

```
      - TINA4_LOG_LEVEL=WARNING
      - JWT_SECRET=${JWT_SECRET}
      - TINA4_DATABASE_URL=sqlite:///data/app.db
    volumes:
      - app-data:/app/data
    restart: unless-stopped
    healthcheck:
      test: ["CMD", "python", "-c", "import urllib.request; urllib.request.urlopen('http://local
      interval: 30s
      timeout: 5s
      retries: 3

volumes:
  app-data:
```

```
# Start everything
docker compose up -d

# View logs
docker compose logs -f app

# Stop everything
docker compose down
```

---

5. Health Checks

Production deployments need a health check endpoint. Load balancers, container orchestrators, and monitoring tools all rely on it to verify the app is running.

Create a health check route:

```
from datetime import datetime, timezone
from tina4_python.core.router import get

@get("/health")
async def health_check(request, response):
    db = Database.get_connection()
    db_ok = False

    try:
        db.fetch_one("SELECT 1")
        db_ok = True
    except Exception:
        pass

    status = "ok" if db_ok else "degraded"
    status_code = 200 if db_ok else 503

    return response({
        "status": status,
        "version": "1.0.0",
        "database": "connected" if db_ok else "disconnected",
        "timestamp": datetime.now(timezone.utc).isoformat()
    }, status_code)
```

This endpoint:

- Returns `200` when everything is healthy
- Returns `503` when the database is down (so the load balancer stops routing traffic)
- Includes version information for deployment tracking
- Runs fast (no heavy queries, no authentication)

---

6. Graceful Shutdown

Deploy a new version. The old process must stop. Graceful shutdown finishes active requests before terminating.

Tina4 handles this when it receives a `SIGTERM` signal (the standard shutdown signal from Docker, Kubernetes, and systemd):

- Stop accepting new connections
- Wait for active requests to complete (up to 30 seconds)
- Close database connections
- Flush logs
- Exit

Configuring Shutdown Timeout

```
TINA4_SHUTDOWN_TIMEOUT=30
```

If active requests do not finish within the timeout, the server terminates them. Set this based on your longest expected request time. For most applications, 30 seconds is generous.

Docker Stop Grace Period

Docker sends `SIGTERM`, waits for the grace period, then sends `SIGKILL`. Match the Docker grace period to your shutdown timeout:

```
services:
  app:
    stop_grace_period: 30s
```

---

7. Log Rotation

In production, logs grow without limit unless rotated. Tina4 writes logs to `logs/app.log` and `logs/error.log`.

Using logrotate (Linux)

Create `/etc/logrotate.d/tina4`:

```
/app/logs/*.log {
    daily
    rotate 14
    compress
```

```
    delaycompress
    missingok
    notifempty
    create 0640 www-data www-data
    postrotate
        kill -USR1 $(cat /app/tina4.pid) 2>/dev/null || true
    endscript
}
```

This rotates logs daily, keeps 14 days of history, and compresses old logs. The `USR1` signal tells Tina4 to reopen its log files after rotation.

Docker Logging

Docker captures stdout/stderr automatically. Configure log rotation in the Docker daemon or compose file:

```
services:
  app:
    logging:
      driver: json-file
      options:
        max-size: "10m"
        max-file: "5"
```

This keeps at most 50MB of logs (5 files of 10MB each).

---

8. Reverse Proxy with Nginx

In production, Tina4 runs behind Nginx. Nginx handles:

- SSL/TLS termination (HTTPS)
- Static file serving (faster than Python)
- Request buffering
- Rate limiting
- WebSocket proxying

Create `/etc/nginx/sites-available/my-app`:

```
server {
    listen 80;
    server_name yourdomain.com;
    return 301 https://$host$request_uri;
}

server {
    listen 443 ssl;
    server_name yourdomain.com;

    ssl_certificate /etc/letsencrypt/live/yourdomain.com/fullchain.pem;
    ssl_certificate_key /etc/letsencrypt/live/yourdomain.com/privkey.pem;

    # Static files (served by Nginx, faster than Python)
    location /css/ {
```

```
        alias /app/src/public/css/;
        expires 30d;
        add_header Cache-Control "public, immutable";
    }

    location /js/ {
        alias /app/src/public/js/;
        expires 30d;
        add_header Cache-Control "public, immutable";
    }

    location /images/ {
        alias /app/src/public/images/;
        expires 30d;
        add_header Cache-Control "public, immutable";
    }

    # WebSocket upgrade
    location /ws/ {
        proxy_pass http://127.0.0.1:7146;
        proxy_http_version 1.1;
        proxy_set_header Upgrade $http_upgrade;
        proxy_set_header Connection "upgrade";
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_read_timeout 86400;
    }

    # Application
    location / {
        proxy_pass http://127.0.0.1:7146;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header X-Forwarded-Proto $scheme;
    }
}
```

---

9. SSL/TLS with Let's Encrypt

HTTPS is non-negotiable in production. Let's Encrypt provides free SSL certificates with automatic renewal.

With Nginx and Certbot

Install Certbot:

```
sudo apt install certbot python3-certbot-nginx
```

Obtain a certificate:

```
sudo certbot --nginx -d yourdomain.com
```

Certbot modifies your Nginx configuration to include SSL settings and sets up automatic renewal. Verify auto-renewal works:

```
sudo certbot renew --dry-run
```

Certbot renews certificates 30 days before expiry. The renewal runs via a systemd timer or cron job that Certbot creates during installation.

With Docker (Using Traefik as Reverse Proxy)

Traefik handles SSL termination and automatic certificate provisioning. Add it to your Docker Compose setup:

```
services:
  reverse-proxy:
    image: traefik:v3.0
    command:
      - "--providers.docker=true"
      - "--entrypoints.web.address=:80"
      - "--entrypoints.websecure.address=:443"
      - "--certificatesresolvers.letsencrypt.acme.tlschallenge=true"
      - "--certificatesresolvers.letsencrypt.acme.email=you@example.com"
      - "--certificatesresolvers.letsencrypt.acme.storage=/letsencrypt/acme.json"
    ports:
      - "80:80"
      - "443:443"
    volumes:
      - /var/run/docker.sock:/var/run/docker.sock:ro
      - letsencrypt:/letsencrypt

  app:
    build: .
    labels:
      - "traefik.http.routers.app.rule=Host(`yourdomain.com`)"
      - "traefik.http.routers.app.tls=true"
      - "traefik.http.routers.app.tls.certresolver=letsencrypt"
    environment:
      - TINA4_DEBUG=false
      - JWT_SECRET=${JWT_SECRET}

volumes:
  letsencrypt:
```

Traefik detects your app container through Docker labels, provisions a certificate from Let's Encrypt, and handles all HTTPS traffic. No Nginx configuration needed.

Certificate Monitoring

Certificates expire. Even with auto-renewal, things go wrong -- DNS changes, firewall rules blocking port 80 for ACME challenges, or a crashed renewal service. Set up monitoring:

```
# Check certificate expiry manually
echo | openssl s_client -connect yourdomain.com:443 2>/dev/null | openssl x509 -noout -dates

# Verify Certbot timer is active
sudo systemctl list-timers | grep certbot
```

Use an external monitoring service (Uptime Robot, Better Uptime) that checks certificate expiry and alerts you 14 days before it expires.

---

10. Scaling

A single server handles many applications. When traffic outgrows one server, you scale.

Multiple Workers

Uvicorn runs multiple worker processes by default. Configure the count in `.env`:

Start with the number of CPU cores on your server. For I/O-heavy applications (database queries, external API calls), double or quadruple the core count. CPU-bound work benefits less from extra workers.

Load Balancing with Nginx

When you run multiple Tina4 instances, Nginx distributes traffic across them:

```
upstream tina4_backend {
    server 127.0.0.1:7146;
    server 127.0.0.1:7246;
    server 127.0.0.1:7346;
    server 127.0.0.1:7446;
}

server {
    listen 80;
    server_name yourdomain.com;

    location / {
        proxy_pass http://tina4_backend;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header X-Forwarded-Proto $scheme;
    }
}
```

Start four instances on different ports. The framework refuses to start without the Rust CLI unless `TINA4_OVERRIDE_CLIENT=true`, so for a multi-process setup either set that override or run each instance through `tina4 serve --port`:

```
TINA4_PORT=7146 tina4 serve --port 7146 &
TINA4_PORT=7246 tina4 serve --port 7246 &
TINA4_PORT=7346 tina4 serve --port 7346 &
TINA4_PORT=7446 tina4 serve --port 7446 &
```

Nginx distributes requests in round-robin order by default. If a backend goes down, Nginx routes traffic to the remaining instances.

Docker Scaling

With Docker Compose, scale horizontally with a single command:

```
docker compose up -d --scale app=4
```

This starts four containers. Place a load balancer (Traefik, Nginx, or a cloud load balancer) in front of them:

```
services:
```

```
  reverse-proxy:
    image: traefik:v3.0
    command:
      - "--providers.docker=true"
      - "--entrypoints.web.address=:80"
    ports:
      - "80:80"
    volumes:
      - /var/run/docker.sock:/var/run/docker.sock:ro

  app:
    build: .
    labels:
      - "traefik.http.routers.app.rule=Host(`yourdomain.com`)"
    environment:
      - TINA4_DEBUG=false
      - TINA4_DATABASE_URL=sqlite:///data/app.db
    volumes:
      - app-data:/app/data

volumes:
  app-data:
```

Scaling Considerations

Scaling introduces shared-state problems. When four instances serve requests, each must agree on the state of the world.

Sessions: Store sessions in Redis, not in-memory. Otherwise, a user who logs in on instance 1 appears logged out on instance 2.

Database: SQLite handles one writer at a time. Under high load with multiple instances, switch to PostgreSQL or MySQL. If you must use SQLite, enable WAL mode.

File uploads: Store uploaded files in shared storage (S3, a mounted volume) -- not the local filesystem of a single container.

Cache: Use Redis as the cache backend so all instances share the same cache.

---

11. Monitoring

Your app runs in production. You need to know when it breaks, slows down, or runs out of resources.

Log Aggregation

Switch to JSON-formatted logs for production. Structured logs feed into aggregation services:

```
TINA4_LOG_FORMAT=json
```

Example JSON log entry:

```
{
  "timestamp": "2026-03-22T14:30:00.123Z",
  "level": "WARNING",
  "message": "Rate limit exceeded",
```

```
  "request_id": "req-abc123",
  "ip": "203.0.113.42",
  "path": "/api/products",
  "method": "GET"
}
```

Services that ingest JSON logs:

Service	Strengths
Grafana Loki	Open source, pairs with Grafana dashboards
Elastic Stack (ELK)	Full-text search across logs
Datadog	Managed service, correlates logs with metrics
AWS CloudWatch	Native for AWS deployments

Docker makes log aggregation straightforward. Configure the logging driver to send container output to your chosen service:

```
services:
  app:
    logging:
      driver: "fluentd"
      options:
        fluentd-address: "localhost:24224"
        tag: "tina4.app"
```

Uptime Monitoring

Point an external monitoring service at your health endpoint:

```
https://yourdomain.com/health
```

Services like Uptime Robot, Pingdom, or Better Uptime ping this endpoint every 30-60 seconds. When it stops responding or returns a non-200 status, you receive an alert via email, SMS, or Slack.

The health endpoint from section 5 serves double duty: container orchestrators use it for restart decisions, and uptime monitors use it for alerting.

Application Performance Monitoring (APM)

Uptime monitoring tells you the app is running. APM tells you how well it performs. APM agents track:

- Request latency (average, p95, p99)
- Database query performance (slow queries, connection pool usage)
- Error rates (which endpoints fail, how often)
- Memory and CPU usage over time

Since Tina4 Python runs on standard Python, any Python APM agent works:

- **Datadog APM**: `uv add ddtrace` and run with `TINA4_OVERRIDE_CLIENT=true ddtrace-run uv run tina4 serve` (the override is required because APM wrappers spawn Python directly rather than going through the Rust CLI)
- **New Relic**: `uv add newrelic` and run with `TINA4_OVERRIDE_CLIENT=true newrelic-admin run-program uv run tina4 serve`
- **Elastic APM**: `uv add elastic-apm` and configure in your app startup, then `TINA4_OVERRIDE_CLIENT=true tina4 serve --production`

A basic monitoring stack for a small team: Uptime Robot for availability alerts (free tier covers it), JSON logs shipped to Grafana Loki for debugging, and `docker stats` for resource usage. Add APM when your application serves enough traffic to warrant the cost.

---

12. Exercise: Docker Deploy

Deploy a Tina4 Python application using Docker.

Requirements

- Create a `Dockerfile` that:
- Uses `tina4stack/tina4-python:v3` as the base image
- Copies the application code
- Creates data directories
- Exposes port 7146
- Create a `docker-compose.yml` that:
- Builds and runs the app
- Mounts volumes for data persistence
- Sets environment variables for production
- Create a `/health` endpoint that checks database connectivity
- Build, run, and verify:

```
# Build
docker compose build

# Start
docker compose up -d

# Test health
curl http://localhost:7146/health

# Test the app
curl http://localhost:7146/api/products

# View logs
docker compose logs -f app

# Stop
```

```
docker compose down
```

Solution

The Dockerfile and docker-compose.yml are shown in section 4. The health check route is shown in section 5. Combine them in your project, then:

```
docker compose up -d --build
```

```
[+] Building 4.2s
[+] Running 1/1
  Container my-app   Started
```

```
curl http://localhost:7146/health
```

```
{
  "status": "ok",
  "version": "1.0.0",
  "database": "connected",
  "timestamp": "2026-03-22T14:30:00+00:00"
}
```

The app runs in production mode with persistent data volumes, automatic restarts, and health monitoring.

---

13. Gotchas

1. .env Not Loaded in Docker

Problem: Environment variables from `.env` are not available in the container.

Cause: Docker does not read `.env` files automatically. The `.env` file belongs in `.dockerignore` (never ship secrets in the image).

Fix: Pass environment variables via `docker run -e`, the `environment` section in `docker-compose.yml`, or an `env_file` directive. For secrets, use Docker secrets or your platform's secret management.

2. SQLite Database Lost on Container Restart

Problem: All data disappears when the container restarts.

Cause: The SQLite database file sits inside the container. When the container is recreated, the file is gone.

Fix: Mount a volume for the data directory: `-v $(pwd)/data:/app/data`. In Docker Compose, use a named volume: `volumes: [app-data:/app/data]`.

3. WebSocket Connections Drop Behind Nginx

Problem: WebSocket connections fail or drop immediately behind Nginx.

Cause: Nginx does not proxy WebSocket by default. It treats the upgrade request as a regular HTTP request.

Fix: Add WebSocket proxy headers in your Nginx config:

```
proxy_http_version 1.1;
proxy_set_header Upgrade $http_upgrade;
proxy_set_header Connection "upgrade";
proxy_read_timeout 86400;
```

4. Built-in Server Used in Production

Problem: The server logs show "WARNING: Do not use the built-in server in production".

Cause: No ASGI server (uvicorn or hypercorn) is installed. The built-in server is single-threaded and not designed for production load.

Fix: Install uvicorn: `uv add uvicorn`. Tina4 auto-detects it and uses it with multiple workers.

5. Static Files Slow

Problem: CSS, JS, and images load slowly in production.

Cause: Python serves static files. Every static file request goes through the Python process.

Fix: Serve static files from Nginx (see the Nginx config in section 8). Nginx serves static files from memory-mapped files, which is orders of magnitude faster than Python.

6. Memory Usage Grows Over Time

Problem: The container's memory usage climbs until it crashes with OOM (out of memory).

Cause: Memory leaks in the application -- unclosed database connections, growing caches without TTL, accumulating request data in global variables.

Fix: Set TTLs on all cache entries. Close database connections properly. Avoid storing request data in module-level variables. Use `docker stats` to monitor memory usage. Set memory limits in Docker Compose: `deploy: {resources: {limits: {memory: 512M}}}`.

7. Container Starts Before Database Is Ready

Problem: The app crashes on startup because the database is not ready.

Cause: Docker Compose starts services in parallel. The app container starts before the database container finishes initializing.

Fix: For SQLite, this is not an issue (the file is created automatically). For PostgreSQL or MySQL, use a startup script that waits for the database, or use Docker Compose healthcheck on the database service with `depends_on: {db: {condition: service_healthy}}`.

8. SSL Certificate Not Renewing

Problem: Your HTTPS certificate expires and the site goes down.

Cause: The auto-renewal process (Certbot or Traefik) failed. Common reasons: DNS changes, firewall blocking port 80 for ACME challenges, or the renewal service crashed.

Fix: Monitor certificate expiry with an external service. Check renewal logs and verify the renewal timer is active:

```
sudo certbot renew --dry-run
sudo systemctl list-timers | grep certbot
```

9. Scaled Instances Have Different State

Problem: Users see inconsistent data across requests when running multiple app instances.

Cause: In-memory sessions and cache are not shared between instances. A user who logs in on instance 1 appears logged out when the load balancer routes the next request to instance 2.

Fix: Store sessions in Redis. Use Redis as the cache backend. Store uploaded files in shared storage (S3 or a mounted volume). All instances must read from and write to the same data stores.

Building a Complete App

1. Putting It All Together

Twenty chapters of building blocks. Routing. Templates. Databases. ORM. Authentication. Middleware. Queues. WebSocket. Caching. Frontend. GraphQL. Testing. Dev tools. CLI scaffolding. Deployment. Now all of it works together in one application.

TaskFlow -- a task management system with:

- User registration and JWT authentication
- Task creation, assignment, and tracking
- A dashboard with real-time updates
- Email notifications when tasks are assigned
- Caching for dashboard performance
- A full test suite
- Docker deployment

Not a toy project. A production-ready application. Every major Tina4 feature in one codebase.

---

2. Planning the App

Models

Model	Table	Fields
User	users	id, name, email, password*hash, role, created*at
Task	tasks	id, title, description, status, priority, created*by, assigned*to, due*date, completed*at, created*at, updated*at

Relationships

- A User has many Tasks (created by them)
- A User has many Tasks (assigned to them)
- A Task belongs to a User (creator)
- A Task belongs to a User (assignee)

Routes

Method	Path	Description	Auth
POST	/api/auth/register	Register a new user	public
POST	/api/auth/login	Login, get JWT	public
GET	/api/profile	Get current user profile	secured
GET	/api/tasks	List tasks (with filters)	secured
GET	/api/tasks/{id}	Get a single task	secured
POST	/api/tasks	Create a task	secured
PUT	/api/tasks/{id}	Update a task	secured
DELETE	/api/tasks/{id}	Delete a task	secured
GET	/api/dashboard/stats	Dashboard statistics	secured
GET	/admin	Dashboard HTML page	public

Templates

- `base.html` -- Base layout with sidebar and topbar
- `dashboard.html` -- Dashboard with stats, task list, quick actions
- `login.html` -- Login page
- `register.html` -- Registration page

---

3. Step 1: Init Project and Set Up Database

```
tina4 init python taskflow
cd taskflow
uv sync
```

Update `.env`:

```
TINA4_DEBUG=true
TINA4_SECRET=taskflow-dev-secret-change-in-production
TINA4_TOKEN_EXPIRES_IN=1440
```

Create Migrations

Create `migrations/20260322000100_create_users_table.sql`:

```
-- UP
CREATE TABLE users (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    name TEXT NOT NULL,
    email TEXT NOT NULL UNIQUE,
```

```
    password_hash TEXT NOT NULL,
    role TEXT NOT NULL DEFAULT 'user',
    created_at TEXT DEFAULT CURRENT_TIMESTAMP
);

CREATE INDEX idx_users_email ON users(email);

-- DOWN
DROP TABLE IF EXISTS users;
```

Create `migrations/20260322000200_create_tasks_table.sql`:

```
-- UP
CREATE TABLE tasks (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    title TEXT NOT NULL,
    description TEXT,
    status TEXT NOT NULL DEFAULT 'pending',
    priority TEXT NOT NULL DEFAULT 'medium',
    created_by INTEGER NOT NULL,
    assigned_to INTEGER,
    due_date TEXT,
    completed_at TEXT,
    created_at TEXT DEFAULT CURRENT_TIMESTAMP,
    updated_at TEXT DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (created_by) REFERENCES users(id),
    FOREIGN KEY (assigned_to) REFERENCES users(id)
);

CREATE INDEX idx_tasks_status ON tasks(status);
CREATE INDEX idx_tasks_assigned_to ON tasks(assigned_to);
CREATE INDEX idx_tasks_created_by ON tasks(created_by);

-- DOWN
DROP TABLE IF EXISTS tasks;
```

Run migrations:

```
tina4 migrate
```

```
Running migrations...
  [UP] 20260322000100_create_users_table.sql
  [UP] 20260322000200_create_tasks_table.sql

  2 migrations applied
```

Verify the database:

```
curl http://localhost:7146/health
```

```
{"status": "ok", "database": "connected"}
```

---

4. Step 2: ORM Models

Create `src/orm/user.py`:

```
from tina4_python.orm import ORM
import hashlib
```

```
import hmac
import os

class User(ORM):
    table_name = "users"
    auto_crud = True

    id: int
    name: str
    email: str
    password_hash: str
    role: str
    created_at: str

    def set_password(self, password):
        salt = os.urandom(32)
        key = hashlib.pbkdf2_hmac("sha256", password.encode(), salt, 100000)
        self.password_hash = salt.hex() + ":" + key.hex()

    def verify_password(self, password):
        if not self.password_hash:
            return False
        salt_hex, key_hex = self.password_hash.split(":")
        salt = bytes.fromhex(salt_hex)
        key = bytes.fromhex(key_hex)
        new_key = hashlib.pbkdf2_hmac("sha256", password.encode(), salt, 100000)
        return hmac.compare_digest(key, new_key)

    def safe_dict(self):
        data = self.to_dict()
        data.pop("password_hash", None)
        return data
```

Create `src/orm/task.py`:

```
from tina4_python.orm import ORM

class Task(ORM):
    table_name = "tasks"
    auto_crud = True

    id: int
    title: str
    description: str
    status: str
    priority: str
    created_by: int
    assigned_to: int
    due_date: str
    completed_at: str
    created_at: str
    updated_at: str

    STATUSES = ["pending", "in_progress", "completed", "cancelled"]
    PRIORITIES = ["low", "medium", "high", "urgent"]
```

---

5. Step 3: Authentication Routes

Create `src/routes/auth.py`:

```
from tina4_python.core.router import post, get
from tina4_python.auth import Auth

@post("/api/auth/register")
async def register(request, response):
    body = request.body

    if not body.get("name") or not body.get("email") or not body.get("password"):
        return response({"error": "Name, email, and password are required"}, 400)

    # Check for existing user
    results, count = User.where("email = ?", [body["email"]])
    if results:
        return response({"error": "Email already registered"}, 409)

    user = User()
    user.name = body["name"]
    user.email = body["email"]
    user.set_password(body["password"])
    user.role = "user"
    user.save()

    return response({
        "message": "Registration successful",
        "id": user.id,
        "name": user.name,
        "email": user.email
    }, 201)


@post("/api/auth/login")
async def login(request, response):
    body = request.body

    if not body.get("email") or not body.get("password"):
        return response({"error": "Email and password are required"}, 400)

    results, count = User.where("email = ?", [body["email"]])

    if not results:
        return response({"error": "Invalid credentials"}, 401)

    user = results[0]
    if not user.verify_password(body["password"]):
        return response({"error": "Invalid credentials"}, 401)

    token = Auth.get_token({
        "user_id": user.id,
        "email": user.email,
        "role": user.role
    })

    return response({
        "token": token,
        "user": user.safe_dict()
```

```
    })
```

Test Registration and Login

```
# Start the server
tina4 serve
```

```
# Register a user
curl -X POST http://localhost:7146/api/auth/register \
  -H "Content-Type: application/json" \
  -d '{"name": "Alice Johnson", "email": "alice@example.com", "password": "securepass123"}'
```

```
{
  "message": "Registration successful",
  "id": 1,
  "name": "Alice Johnson",
  "email": "alice@example.com"
}
```

```
# Login
curl -X POST http://localhost:7146/api/auth/login \
  -H "Content-Type: application/json" \
  -d '{"email": "alice@example.com", "password": "securepass123"}'
```

```
{
  "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...",
  "user": {
    "id": 1,
    "name": "Alice Johnson",
    "email": "alice@example.com",
    "role": "user"
  }
}
```

Authentication works. Save the token for the next steps.

---

6. Step 4: Auth Middleware

Create `src/middleware/auth.py`:

```
from tina4_python.auth import Auth

async def auth_middleware(request, response, next_handler):
    auth_header = request.headers.get("Authorization", "")

    if not auth_header.startswith("Bearer "):
        return response({"error": "Authentication required"}, 401)

    token = auth_header.replace("Bearer ", "")

    payload = Auth.valid_token(token)
    if payload is None:
        return response({"error": "Invalid or expired token"}, 401)

    request.user = payload
    request.user_id = payload["user_id"]

    return await next_handler(request, response)
```

Verify the middleware blocks unauthenticated requests:

```
curl http://localhost:7146/api/tasks
```

```
{"error": "Authentication required"}
```

The middleware stands guard. No token, no access.

---

7. Step 5: Task CRUD Routes

Create `src/routes/tasks.py`:

```
from datetime import datetime, timezone
from tina4_python.core.router import get, post, put, delete
from tina4_python.cache import cache_get, cache_set, cache_delete

@get("/api/tasks", middleware=["auth_middleware"])
async def list_tasks(request, response):
    status = request.params.get("status")
    assigned_to = request.params.get("assigned_to")
    page = int(request.params.get("page", 1))
    limit = int(request.params.get("limit", 20))
    offset = (page - 1) * limit

    where = "1=1"
    params = []

    if status:
        where += " AND status = ?"
        params.append(status)
    if assigned_to:
        where += " AND assigned_to = ?"
        params.append(int(assigned_to))

    tasks, total = Task.where(where, params, limit=limit, offset=offset)

    return response({
        "tasks": [t.to_dict() for t in tasks],
        "page": page,
        "limit": limit
    })


@get("/api/tasks/{task_id}", middleware=["auth_middleware"])
async def get_task(request, response):
    task_id = request.params["task_id"]

    task = Task.find(task_id)

    if task is None:
        return response({"error": "Task not found"}, 404)

    # Load creator and assignee names
    creator = User.find(task.created_by)

    result = task.to_dict()
    result["creator_name"] = creator.name if creator else "Unknown"
```

```
    if task.assigned_to:
        assignee = User.find(task.assigned_to)
        result["assignee_name"] = assignee.name if assignee else "Unassigned"

    return response(result)


@post("/api/tasks", middleware=["auth_middleware"])
async def create_task(request, response):
    body = request.body

    if not body.get("title"):
        return response({"error": "Title is required"}, 400)

    task = Task()
    task.title = body["title"]
    task.description = body.get("description", "")
    task.status = body.get("status", "pending")
    task.priority = body.get("priority", "medium")
    task.created_by = request.user_id
    task.assigned_to = body.get("assigned_to")
    task.due_date = body.get("due_date")
    task.save()

    # Invalidate dashboard cache
    cache_delete("dashboard:stats")

    # Send email notification if assigned
    if task.assigned_to:
        await notify_assignee(task)

    # Push WebSocket update
    await push_task_update("created", task)

    return response(task.to_dict(), 201)


@put("/api/tasks/{task_id}", middleware=["auth_middleware"])
async def update_task(request, response):
    task_id = request.params["task_id"]
    body = request.body

    task = Task.find(task_id)

    if task is None:
        return response({"error": "Task not found"}, 404)

    if "title" in body:
        task.title = body["title"]
    if "description" in body:
        task.description = body["description"]
    if "status" in body:
        if body["status"] not in Task.STATUSES:
            return response({"error": f"Invalid status. Must be one of: {Task.STATUSES}"}, 400)
        task.status = body["status"]
        if body["status"] == "completed":
            task.completed_at = datetime.now(timezone.utc).isoformat()
    if "priority" in body:
        if body["priority"] not in Task.PRIORITIES:
```

```
            return response({"error": f"Invalid priority. Must be one of: {Task.PRIORITIES}"}, 4
        task.priority = body["priority"]
    if "assigned_to" in body:
        old_assignee = task.assigned_to
        task.assigned_to = body["assigned_to"]
        if body["assigned_to"] != old_assignee and body["assigned_to"]:
            await notify_assignee(task)
    if "due_date" in body:
        task.due_date = body["due_date"]

    task.updated_at = datetime.now(timezone.utc).isoformat()
    task.save()

    # Invalidate dashboard cache
    cache_delete("dashboard:stats")

    # Push WebSocket update
    await push_task_update("updated", task)

    return response(task.to_dict())


@delete("/api/tasks/{task_id}", middleware=["auth_middleware"])
async def delete_task(request, response):
    task_id = request.params["task_id"]

    task = Task.find(task_id)

    if task is None:
        return response({"error": "Task not found"}, 404)

    task.delete()

    cache_delete("dashboard:stats")

    return response(None, 204)
```

Test the Task API

```
# Set your token from the login step
TOKEN="eyJhbGciOiJIUzI1NiIs..."

# Create a task
curl -X POST http://localhost:7146/api/tasks \
  -H "Content-Type: application/json" \
  -H "Authorization: Bearer $TOKEN" \
  -d '{"title": "Design database schema", "priority": "high"}'
```

```
{
  "id": 1,
  "title": "Design database schema",
  "priority": "high",
  "status": "pending",
  "created_by": 1,
  "created_at": "2026-03-22 10:00:00"
}
```

```
# Create more tasks
curl -X POST http://localhost:7146/api/tasks \
  -H "Content-Type: application/json" \
```

```
  -H "Authorization: Bearer $TOKEN" \
  -d '{"title": "Build API endpoints", "priority": "high"}'

curl -X POST http://localhost:7146/api/tasks \
  -H "Content-Type: application/json" \
  -H "Authorization: Bearer $TOKEN" \
  -d '{"title": "Write documentation", "priority": "medium", "due_date": "2026-04-01"}'

# List all tasks
curl http://localhost:7146/api/tasks \
  -H "Authorization: Bearer $TOKEN"

# Update a task status
curl -X PUT http://localhost:7146/api/tasks/1 \
  -H "Content-Type: application/json" \
  -H "Authorization: Bearer $TOKEN" \
  -d '{"status": "completed"}'

# Delete a task
curl -X DELETE http://localhost:7146/api/tasks/3 \
  -H "Authorization: Bearer $TOKEN"
```

Every endpoint responds. Create. Read. Update. Delete. The CRUD cycle works.

---

8. Step 6: Dashboard Stats with Caching

Create `src/routes/dashboard.py`:

```
from tina4_python.core.router import get, template
from tina4_python.cache import cache_get, cache_set

@get("/api/dashboard/stats", middleware=["auth_middleware"])
async def dashboard_stats(request, response):
    cached = cache_get("dashboard:stats")
    if cached:
        return response({**cached, "source": "cache"})

    db = Database.get_connection()

    total = db.fetch_one("SELECT COUNT(*) as count FROM tasks")
    pending = db.fetch_one("SELECT COUNT(*) as count FROM tasks WHERE status = 'pending'")
    in_progress = db.fetch_one("SELECT COUNT(*) as count FROM tasks WHERE status = 'in_progress'
    completed = db.fetch_one("SELECT COUNT(*) as count FROM tasks WHERE status = 'completed'")
    overdue = db.fetch_one(
        "SELECT COUNT(*) as count FROM tasks WHERE due_date < date('now') AND status != 'complet
    )
    users = db.fetch_one("SELECT COUNT(*) as count FROM users")

    recent = db.fetch_all(
        "SELECT t.*, u.name as assignee_name FROM tasks t "
        "LEFT JOIN users u ON t.assigned_to = u.id "
        "ORDER BY t.created_at DESC LIMIT 10"
    )

    stats = {
        "total_tasks": total["count"],
        "pending": pending["count"],
```

```
        "in_progress": in_progress["count"],
        "completed": completed["count"],
        "overdue": overdue["count"],
        "total_users": users["count"],
        "recent_tasks": recent
    }

    cache_set("dashboard:stats", stats, ttl=30)

    return response({**stats, "source": "database"})


@get("/admin")
async def admin_dashboard(request, response):
    return response(template("dashboard.html"))
```

Verify the stats endpoint:

```
curl http://localhost:7146/api/dashboard/stats \
  -H "Authorization: Bearer $TOKEN"

{
  "total_tasks": 2,
  "pending": 1,
  "in_progress": 0,
  "completed": 1,
  "overdue": 0,
  "total_users": 1,
  "recent_tasks": [...],
  "source": "database"
}
```

Hit it again. The second response comes from cache:

```
curl http://localhost:7146/api/dashboard/stats \
  -H "Authorization: Bearer $TOKEN"

{
  "total_tasks": 2,
  "pending": 1,
  "completed": 1,
  "source": "cache"
}
```

The cache holds for 30 seconds. Creating or updating a task invalidates it.

---

9. Step 7: WebSocket Real-Time Updates

Create `src/routes/task_ws.py`:

```
import json
from tina4_python.core.router import websocket, push_to_websocket

@websocket("/ws/tasks")
async def task_updates(connection, event, data):
    if event == "open":
        await connection.send(json.dumps({
            "type": "connected",
```

```
            "message": "Listening for task updates"
        }))

    if event == "message":
        message = json.loads(data)
        if message.get("type") == "ping":
            await connection.send(json.dumps({"type": "pong"}))


async def push_task_update(action, task):
    """Push a task update to all connected WebSocket clients."""
    await push_to_websocket("/ws/tasks", json.dumps({
        "type": "task_update",
        "action": action,
        "task": task.to_dict()
    }))
```

Any user creates, updates, or deletes a task. All connected dashboard users see the change.

---

10. Step 8: Email Notifications

Create `src/routes/notifications.py`:

```
from tina4_python.messenger import Messenger
from tina4_python.core.router import template

async def notify_assignee(task):
    """Send email notification when a task is assigned."""
    assignee = User.find(task.assigned_to)

    if assignee is None:
        return

    creator = User.find(task.created_by)

    html_body = template("emails/task-assigned.html",
        assignee_name=assignee.name,
        task_title=task.title,
        task_description=task.description or "No description",
        task_priority=task.priority,
        task_due_date=task.due_date or "No due date",
        creator_name=creator.name if creator else "Unknown",
        task_url=f"http://localhost:7146/admin#task-{task.id}"
    )

    mailer = Messenger()
    mailer.send(
        to=assignee.email,
        subject=f"Task assigned: {task.title}",
        body=html_body,
        text_body=f"Hi {assignee.name},\n\n"
                  f"{creator.name} assigned you a task: {task.title}\n"
                  f"Priority: {task.priority}\n"
                  f"Due: {task.due_date or 'No due date'}\n\n"
                  f"Description:\n{task.description or 'No description'}"
    )
```

Create `src/templates/emails/task-assigned.html`:

```
<!DOCTYPE html>
<html>
<head><meta charset="UTF-8"></head>
<body>
    <div class="container">
        <h2>New Task Assigned</h2>
        <p>Hi {{ assignee_name }},</p>
        <p><strong>{{ creator_name }}</strong> assigned you a new task:</p>

        <div class="card">
            <div class="card-body">
                <h3>{{ task_title }}</h3>
                <p>{{ task_description }}</p>
                <table class="table">
                    <tr><td><strong>Priority:</strong></td><td>{{ task_priority }}</td></tr>
                    <tr><td><strong>Due:</strong></td><td>{{ task_due_date }}</td></tr>
                </table>
            </div>
        </div>

        <p><a href="{{ task_url }}">View Task</a></p>
    </div>
</body>
</html>
```

---

11. Step 9: Dashboard Template

Create `src/templates/dashboard.html`:

```
{% extends "base.html" %}

{% block title %}TaskFlow Dashboard{% endblock %}

{% block content %}
<h1>Dashboard</h1>

<div id="stats" class="row mb-4">
    <div class="col-md-2"><div class="card"><div class="card-body">
        <h6 class="text-muted">Total</h6><h2 id="stat-total">--</h2>
    </div></div></div>
    <div class="col-md-2"><div class="card"><div class="card-body">
        <h6 class="text-muted">Pending</h6><h2 id="stat-pending">--</h2>
    </div></div></div>
    <div class="col-md-2"><div class="card"><div class="card-body">
        <h6 class="text-muted">In Progress</h6><h2 id="stat-progress">--</h2>
    </div></div></div>
    <div class="col-md-2"><div class="card"><div class="card-body">
        <h6 class="text-muted">Completed</h6><h2 id="stat-completed">--</h2>
    </div></div></div>
    <div class="col-md-2"><div class="card"><div class="card-body">
        <h6 class="text-muted">Overdue</h6><h2 id="stat-overdue" class="text-danger">--</h2>
    </div></div></div>
    <div class="col-md-2"><div class="card"><div class="card-body">
        <h6 class="text-muted">Users</h6><h2 id="stat-users">--</h2>
    </div></div></div>
```

```
</div>

<div class="row">
    <div class="col-md-8">
        <div class="card">
            <div class="card-header d-flex justify-content-between">
                <span>Recent Tasks</span>
                <button class="btn btn-sm btn-primary" data-toggle="modal" data-target="#newTask
                    New Task
                </button>
            </div>
            <div class="card-body">
                <table class="table table-striped" id="task-table">
                    <thead>
                        <tr><th>Title</th><th>Assignee</th><th>Priority</th><th>Status</th></tr>
                    </thead>
                    <tbody id="task-list"></tbody>
                </table>
            </div>
        </div>
    </div>
    <div class="col-md-4">
        <div class="card">
            <div class="card-header">Live Updates</div>
            <div class="card-body" id="live-updates">
                <p class="text-muted">Connecting...</p>
            </div>
        </div>
    </div>
</div>

<script src="/js/frond.min.js"></script>
<script>
    var token = localStorage.getItem("token");
    if (token) frond.setToken(token);

    // Load dashboard stats
    frond.get("/api/dashboard/stats", function (data) {
        document.getElementById("stat-total").textContent = data.total_tasks;
        document.getElementById("stat-pending").textContent = data.pending;
        document.getElementById("stat-progress").textContent = data.in_progress;
        document.getElementById("stat-completed").textContent = data.completed;
        document.getElementById("stat-overdue").textContent = data.overdue;
        document.getElementById("stat-users").textContent = data.total_users;

        var tbody = document.getElementById("task-list");
        tbody.innerHTML = "";
        (data.recent_tasks || []).forEach(function (task) {
            var tr = document.createElement("tr");
            var badgeColor = {pending: "secondary", in_progress: "warning",
                              completed: "success", cancelled: "danger"};
            tr.innerHTML = "<td>" + task.title + "</td>" +
                "<td>" + (task.assignee_name || "Unassigned") + "</td>" +
                "<td>" + task.priority + "</td>" +
                "<td><span class='badge bg-" + (badgeColor[task.status] || "secondary") +
                "'>" + task.status + "</span></td>";
            tbody.appendChild(tr);
        });
    });
```

```
    // WebSocket for live updates
    var ws = frond.ws("/ws/tasks");
    var updates = document.getElementById("live-updates");

    ws.on("open", function () {
        updates.innerHTML = "<p class='text-success'>Connected - listening for updates</p>";
    });

    ws.on("message", function (raw) {
        var msg = JSON.parse(raw);
        if (msg.type === "task_update") {
            var div = document.createElement("div");
            div.innerHTML = "<strong>" + msg.action + ":</strong> " + msg.task.title;
            updates.prepend(div);

            // Refresh stats
            frond.get("/api/dashboard/stats", function () {});
        }
    });
</script>
{% endblock %}
```

Verify the dashboard:

```
curl http://localhost:7146/admin
```

The server returns the rendered HTML. Open `http://localhost:7146/admin` in your browser to see the full dashboard with stats, task list, and live update panel.

---

12. Step 10: Tests

Create `tests/test_taskflow.py`:

```
import uuid
import json
from tina4_python.test import Test, assert_equal, assert_true, assert_not_none

class TaskFlowTest(Test):

    def _register_and_login(self):
        email = f"test-{uuid.uuid4().hex[:8]}@example.com"
        self.post("/api/auth/register", {
            "name": "Test User",
            "email": email,
            "password": "TestPassword123!"
        })
        login_resp = self.post("/api/auth/login", {
            "email": email,
            "password": "TestPassword123!"
        })
        return json.loads(login_resp.body)["token"]

    def test_register(self):
        resp = self.post("/api/auth/register", {
            "name": "Alice",
            "email": f"alice-{uuid.uuid4().hex[:8]}@example.com",
            "password": "SecurePass123!"
```

```
        })
        assert_equal(resp.status_code, 201, "Registration should succeed")

    def test_login(self):
        email = f"login-{uuid.uuid4().hex[:8]}@example.com"
        self.post("/api/auth/register", {
            "name": "Bob",
            "email": email,
            "password": "SecurePass123!"
        })
        resp = self.post("/api/auth/login", {
            "email": email,
            "password": "SecurePass123!"
        })
        assert_equal(resp.status_code, 200, "Login should succeed")
        body = json.loads(resp.body)
        assert_not_none(body.get("token"), "Should return token")

    def test_create_task(self):
        token = self._register_and_login()
        resp = self.post("/api/tasks", {
            "title": "Test Task",
            "description": "A test task",
            "priority": "high"
        }, headers={"Authorization": f"Bearer {token}"})

        assert_equal(resp.status_code, 201, "Task creation should succeed")
        body = json.loads(resp.body)
        assert_equal(body["title"], "Test Task", "Title should match")
        assert_equal(body["priority"], "high", "Priority should match")

    def test_list_tasks(self):
        token = self._register_and_login()
        self.post("/api/tasks", {"title": "List Task 1"}, headers={"Authorization": f"Bearer {to
        self.post("/api/tasks", {"title": "List Task 2"}, headers={"Authorization": f"Bearer {to

        resp = self.get("/api/tasks", headers={"Authorization": f"Bearer {token}"})
        assert_equal(resp.status_code, 200, "Should return 200")
        body = json.loads(resp.body)
        assert_true(len(body["tasks"]) >= 2, "Should have at least 2 tasks")

    def test_update_task_status(self):
        token = self._register_and_login()
        create_resp = self.post("/api/tasks", {"title": "Status Task"},
                                headers={"Authorization": f"Bearer {token}"})
        task_id = json.loads(create_resp.body)["id"]

        resp = self.put(f"/api/tasks/{task_id}", {"status": "completed"},
                        headers={"Authorization": f"Bearer {token}"})
        assert_equal(resp.status_code, 200, "Update should succeed")
        body = json.loads(resp.body)
        assert_equal(body["status"], "completed", "Status should be updated")
        assert_not_none(body.get("completed_at"), "Should have completion timestamp")

    def test_delete_task(self):
        token = self._register_and_login()
        create_resp = self.post("/api/tasks", {"title": "Delete Me"},
                                headers={"Authorization": f"Bearer {token}"})
        task_id = json.loads(create_resp.body)["id"]
```

```
        resp = self.delete(f"/api/tasks/{task_id}",
                           headers={"Authorization": f"Bearer {token}"})
        assert_equal(resp.status_code, 204, "Delete should succeed")

    def test_dashboard_stats(self):
        token = self._register_and_login()
        self.post("/api/tasks", {"title": "Stats Task"},
                  headers={"Authorization": f"Bearer {token}"})

        resp = self.get("/api/dashboard/stats",
                        headers={"Authorization": f"Bearer {token}"})
        assert_equal(resp.status_code, 200, "Should return stats")
        body = json.loads(resp.body)
        assert_true(body["total_tasks"] >= 1, "Should have at least 1 task")

    def test_unauthorized_access(self):
        resp = self.get("/api/tasks")
        assert_equal(resp.status_code, 401, "Should reject unauthenticated request")
```

Run the tests:

```
tina4 test

Running tests...

  TaskFlowTest
    [PASS] test_register
    [PASS] test_login
    [PASS] test_create_task
    [PASS] test_list_tasks
    [PASS] test_update_task_status
    [PASS] test_delete_task
    [PASS] test_dashboard_stats
    [PASS] test_unauthorized_access

  8 tests, 8 passed, 0 failed (0.84s)
```

All green. The application works.

---

13. Step 11: Docker Deployment

Create `Dockerfile`:

```
FROM python:3.12-slim

COPY --from=ghcr.io/astral-sh/uv:latest /uv /usr/local/bin/uv

WORKDIR /app

COPY pyproject.toml uv.lock ./
RUN uv sync --frozen --no-dev

COPY . .
RUN mkdir -p data logs

EXPOSE 7146

HEALTHCHECK --interval=30s --timeout=5s --retries=3 \
```

```
  CMD curl -f http://localhost:7146/health || exit 1

CMD ["uv", "run", "python", "app.py"]
```

Create `docker-compose.yml`:

```
services:
  app:
    build: .
    ports:
      - "7146:7146"
    environment:
      - TINA4_DEBUG=false
      - JWT_SECRET=${JWT_SECRET:-change-me-in-production}
      - TINA4_DATABASE_URL=sqlite:///data/app.db
      - TINA4_CACHE_BACKEND=redis
      - TINA4_CACHE_URL=redis
    volumes:
      - taskflow-data:/app/data
      - taskflow-logs:/app/logs
    depends_on:
      - redis
    restart: unless-stopped

  redis:
    image: redis:7-alpine
    restart: unless-stopped

volumes:
  taskflow-data:
  taskflow-logs:
```

Deploy:

```
docker compose up -d --build
```

Verify:

```
curl http://localhost:7146/health
```

```
{"status": "ok", "version": "1.0.0", "database": "connected"}
```

TaskFlow runs in production. Authenticated APIs. Real-time WebSocket updates. Email notifications. Cached dashboard stats. Tested. Dockerized.

---

14. The Complete Project Structure

```
taskflow/
■■■ .env
■■■ .env.example
■■■ .gitignore
■■■ pyproject.toml
■■■ uv.lock
■■■ app.py
■■■ Dockerfile
■■■ docker-compose.yml
■■■ src/
■   ■■■ routes/
■   ■   ■■■ auth.py               # Registration, login
■   ■   ■■■ tasks.py              # Task CRUD
■   ■   ■■■ dashboard.py          # Dashboard stats + page
■   ■   ■■■ notifications.py      # Email notification helpers
■   ■   ■■■ task_ws.py            # WebSocket event handlers
■   ■■■ orm/
■   ■   ■■■ user.py               # User model with auth methods
■   ■   ■■■ task.py               # Task model with relationships
■   ■■■ middleware/
■   ■   ■■■ auth.py               # JWT auth middleware
■   ■■■ migrations/
■   ■   ■■■ 20260322000100_create_users_table.sql
■   ■   ■■■ 20260322000200_create_tasks_table.sql
■   ■■■ templates/
■   ■   ■■■ base.html             # Base layout
■   ■   ■■■ dashboard.html        # Dashboard page
■   ■   ■■■ emails/
■   ■   ■   ■■■ task-assigned.html # Assignment notification
■   ■   ■■■ errors/
■   ■       ■■■ 404.html
■   ■       ■■■ 500.html
■   ■■■ public/
■   ■   ■■■ css/
■   ■   ■   ■■■ tina4.css
■   ■   ■■■ js/
■   ■       ■■■ tina4.min.js
■   ■       ■■■ frond.min.js
■   ■■■ locales/
■       ■■■ en.json
■■■ data/
■   ■■■ app.db
■■■ logs/
■■■ secrets/
■■■ tests/
    ■■■ test_taskflow.py
```

Every file has a purpose. Every directory follows the convention. A new developer looks at this structure and knows where to find things.

---

15. What You Built

This chapter used every major concept from the book:

Feature	Chapter
Route decorators (`@get`, `@post`, `@put`, `@delete`)	Chapter 2
Request/response handling	Chapter 3
Jinja templates	Chapter 4
Database queries	Chapter 5
ORM models (User, Task)	Chapter 6
JWT authentication	Chapter 8
Auth middleware	Chapter 10
Email notifications (Messenger)	Chapter 16
Cache (dashboard stats)	Chapter 11
Frontend (tina4css dashboard)	Chapter 17
WebSocket (live updates)	Chapter 23
Testing (full test suite)	Chapter 18
Docker deployment	Chapter 33

---

16. What to Build Next

TaskFlow is a solid foundation. Here are ideas for extending it.

Features:

- **Task comments** -- Add a Comment model with a `task_id` foreign key. Display comments on the task detail page.
- **File attachments** -- Let users upload files to tasks. Store them in `data/uploads/` and serve them via a route.
- **Team management** -- Add a Team model. Users belong to teams. Tasks are scoped to teams.
- **Task labels/tags** -- Many-to-many relationship between tasks and labels for categorization.
- **Due date reminders** -- Use the queue system to schedule reminder emails 24 hours before a task's due date.
- **Activity log** -- Record every change to a task (who changed what, when) for audit trails.
- **Search** -- Full-text search across task titles and descriptions.
- **Calendar view** -- Render tasks on a calendar based on their due dates.

- **Mobile API** -- The API already works for mobile apps. Add push notification support via Firebase Cloud Messaging.

Technical improvements:

- **Rate limiting per user** -- Replace the global rate limiter with per-user limits.
- **Database upgrade** -- Switch from SQLite to PostgreSQL for better concurrency.
- **CI/CD pipeline** -- Add GitHub Actions to run tests on every push.
- **API documentation** -- Generate OpenAPI/Swagger docs from your route definitions.
- **Internationalization** -- Add `src/locales/` files for multiple languages.

---

17. Closing Thoughts -- The Tina4 Philosophy

You built a complete application. User auth. CRUD. Real-time updates. Email. Caching. Tests. Deployment. Your project has one dependency: `tina4_python`.

No separate ORM package. No template engine package. No authentication library. No WebSocket server. No caching library. No testing framework. No CLI tool. No CSS framework. No JavaScript helpers. All built in.

One framework. Zero extra dependencies. Everything you need.

The same patterns work in PHP, Ruby, and Node.js. Same project structure. Same CLI commands. Same `.env` variables. Same template syntax. Learn Tina4 once. Use it everywhere.

Build things. Ship them. Keep it simple.

Release Notes

v3.12.4 (2026-05-06)

Documentation-truth release. The `audit-truth.py` CI gate (introduced post-3.12.3) flagged 39 env vars referenced in docs that no framework actually read. This release closes that gap: 25 of them now exist in code, the other 14 are deleted from docs (11 hallucinations + 6 clustering vars deferred to tina4#2). Both audit gates (CLI drift + env-var drift) are now strict in CI.

25 new env vars across all 4 frameworks

Server: `TINA4_HOST`, `TINA4_SUPPRESS`, `TINA4_ENV_FILE`. Health: `TINA4_HEALTH_PATH` (default `/__health`, with `/health` kept as a legacy alias), `TINA4_TRAILING_SLASH_REDIRECT`. Sessions: `TINA4_SESSION_HTTPONLY`, `TINA4_SESSION_NAME`, `TINA4_SESSION_SECURE`. Templates: `TINA4_TEMPLATE_CACHE_TTL` (`0` = permanent). GraphQL: `TINA4_GRAPHQL_AUTO_SCHEMA`, `TINA4_GRAPHQL_ENDPOINT`. Mail: `TINA4_MAIL_IMAP_ENCRYPTION` (`tls`/`starttls`/`none`). MCP: `TINA4_MCP`, `TINA4_MCP_PORT`. Swagger: `TINA4_SWAGGER_ENABLED`, `TINA4_SWAGGER_CONTACT_EMAIL`, `TINA4_SWAGGER_LICENSE`. Database: `TINA4_DB_POOL` (env override on the existing `Database(url, pool=N)` constructor argument).

Logging — env-driven file output + rotation

Six new vars give you full control over logging without touching code:

Var	Default	What it does
`TINA4_LOG_FILE`	*(empty — stdout only)*	Path to a log file. Empty leaves you on stdout.
`TINA4_LOG_DIR`	`logs`	Directory for log files (joined with `_LOG_FILE` if relative).
`TINA4_LOG_FORMAT`	`text`	`text` or `json`. JSON mode emits one structured record per line.
`TINA4_LOG_OUTPUT`	`stdout`	`stdout`, `file`, or `both`. Strict — `stdout` means stdout only.
`TINA4_LOG_CRITICAL`	`false`	Enables a `Log.critical()` level above `error`. Off = no-op.
`TINA4_LOG_ROTATE_SIZE`	`10485760` (10 MB)	Rotate when the file exceeds this many bytes. `0` disables rotation.

`TINA4_LOG_ROTATE_KEEP`	`5`	Number of rotated files to retain (`app.log.1` … `app.log.N`). Older ones are deleted.

Implementation uses each language's stdlib — Python's `logging.handlers.RotatingFileHandler`, Ruby's `Logger.new(path, shift_age, shift_size)`, and a roll-your-own atomic-rename pattern in PHP and Node. Zero new dependencies in any framework.

Documentation-truth CI gate now strict on both axes

The `audit-truth.py` script now blocks merges to `main` of `tina4-documentation` whenever a doc references a `tina4 <command>` or `TINA4_*` env var that doesn't exist in source. Previously CLI drift was strict; env drift was warn-only. Today both are strict.

Tests added

- Python: +53 tests in `tests/test_env_vars.py` (2395 → 2448)
- PHP: +59 tests in `tests/EnvVarTest.php` (2172 → 2231)
- Ruby: +51 examples in `spec/env_vars_spec.rb` (2696 → 2747)
- Node: +59 tests in `test/envVars.test.ts` (3204 → 3263)

Cross-framework total: 10,689 tests passing, +222 from 3.12.3.

Upgrade path

Drop in. No breaking changes — every new env var is opt-in with a sensible default. If you were setting any of the 17 deleted vars in your `.env`, the boot guard will warn (then ignore) — clean them out at your leisure.

v3.12.3 (2026-05-05)

Cross-framework parity sweep. Two minor breaking changes in the Ruby and PHP public API that bring all four frameworks onto the same shape.

Breaking changes (Ruby + PHP only)

Ruby Container — predicate now uses `?` suffix.

```
# before (3.12.2 and earlier)
Tina4::Container.has(:mailer)        # outdated

# after (3.12.3)
Tina4::Container.has?(:mailer)       # idiomatic Ruby predicate
```

This brings Ruby in line with Python (`has()`), PHP (`has()`), and Node (`has()`) while still respecting Ruby's `?`-suffix idiom for predicates returning bool. The pre-existing `resolve` → `get` rename happened earlier; only the predicate was lagging.

ResponseCache public surface — middleware-only across all four frameworks.

The cache has always been middleware. Two of the four frameworks (PHP, Ruby) historically exposed lookup/store as public methods, which let users couple to internals. The public API is now consistent across all four: use the middleware on a route, and read stats with module-level helpers.

```
# Ruby — module-level helpers (parity with Python)
Tina4.cache_stats   # → { hits:, misses:, size:, backend:, keys: }
Tina4.clear_cache   # flush all entries

# PHP — static methods on the class
\Tina4\Middleware\ResponseCache::cacheStats();
\Tina4\Middleware\ResponseCache::clearCache();
```

Internal methods that used to be public (`get`, `lookup`, `store`, `cache_response`) are now private. Tests that needed them retain access via `_internal*` test seams marked `@internal`.

Doc parity — CLAUDE.md and book chapter 33

• **CLAUDE.md**: every framework's "Key Method Stubs" section now covers the same surface area Python documents — Queue, QueryBuilder, Frond, Api, Background Tasks, ResponseCache, etc. PHP added 4 sections; Ruby added 5; Node added 13.

• **Book chapter 33**: env var tables are now grounded in source. Each framework's chapter 33 lists every `TINA4_*` var its source actually reads. Found and fixed several gaps — Ruby was missing `TINA4_CACHE_*`, `TINA4_QUEUE_*`, `TINA4_KAFKA_*`, `TINA4_RABBITMQ_*`, `TINA4_MONGO_*`, `TINA4_WS_BACKPLANE`, and the entire `TINA4_SESSION_VALKEY_*` block.

Other fixes

• **Ruby** `lib/tina4/ai.rb` — subprocess output is now force-encoded to UTF-8 before `String#strip`, fixing `Encoding::CompatibilityError` that crashed 4 ai specs on systems with non-ASCII pip output.

• **Node** `test/serverParity.test.ts` — sets `TINA4_OVERRIDE_CLIENT=true` so `start()` actually runs, plus emits the `N passed, M failed` summary line the runner expects. The test was effectively a no-op before; now it's recorded properly.

Genuine gaps surfaced by the parity audit (follow-up, not blocking 3.12.3)

The chapter 33 audit flagged env vars Python documents that no other framework actually reads — Ruby/PHP/Node lack `TINA4_OPEN_BROWSER`, `TINA4_DEV_POLL_INTERVAL`, `TINA4_PUBLIC_DIR`, `TINA4_TOKEN_EXPIRES_IN` alias, plus a few framework-specific gaps (Ruby has no Mongo session backend; Node `TINA4_CSRF` defaults to `false` vs Python's `true`). Tracked for a future patch.

Upgrade path

Symptom	Fix
Ruby: `NoMethodError: undefined method 'has' for Tina4::Container`	Replace `has(:key)` with `has?(:key)`
PHP: `BadMethodCallException` calling `$cache->lookup(...)`	Use the middleware: `[ResponseCache::class, 'beforeCache']` / `[..., 'afterCache']`. Or call `_internalLookup` if you really need direct access (test code only — `@internal`).
Ruby: `NoMethodError: undefined method 'get' for ResponseCache instance`	Use `Tina4.cache_stats` / `Tina4.clear_cache` for stats. Lookup goes through the middleware.

No `.env` changes from 3.12.2.

v3.12.2 (2026-05-05)

Quality-of-life patch. Two related portability fixes — no breaking changes from 3.12.1.

Firebird URL auto-detect

Firebird is the awkward one in the stack. Every other engine has a server-side database name (`postgres://host:port/dbname`), but Firebird wants either an absolute file path on the server, a Windows drive-letter path, or an alias. The classic URI form needs a double slash to keep the leading `/` of an absolute path through the URL parser — unintuitive to anyone used to the way postgres / mysql / mssql encode the database name.

The framework now accepts five equivalent forms and normalises all of them transparently:

URL path you write	Resolved Firebird identifier
`//abs/path/db.fdb` (classic double-slash)	`/abs/path/db.fdb`
`/abs/path/db.fdb` (single-slash, intuitive)	`/abs/path/db.fdb`
`/C:/Data/db.fdb` (Windows drive letter)	`C:/Data/db.fdb`
`/C%3A/Data/db.fdb` (URL-encoded colon)	`C:/Data/db.fdb`
`/employee` (Firebird alias)	`employee`

For ops setups that keep server URL and DB location in separate config layers — or for Windows backslash paths that fight URL encoding — set `TINA4_DATABASE_FIREBIRD_PATH`. The env override wins over whatever path is in the URL.

```
TINA4_DATABASE_FIREBIRD_PATH=C:\firebird\data\app.fdb
TINA4_DATABASE_URL=firebird://SYSDBA:masterkey@localhost:3050/ignored
```

Shipped to all 4 frameworks. 11 regression tests per framework (8 unit + 3 live).

Bug fix specific to PHP — `mysqli` localhost+port quirk

PHP's `mysqli` has a long-standing quirk where `host == "localhost"` triggers a Unix socket lookup and IGNORES the port argument entirely. Connecting to `mysql://...:53306` against a Docker container fails with "No such file or directory" — `mysqli` is hunting for `/tmp/mysql.sock` instead of opening a TCP connection. `MySQLAdapter::rewriteHostForTcp()` now rewrites `localhost` to `127.0.0.1` when a non-zero port is specified, forcing the TCP code path. Bare `mysql:///db` (no port) is preserved so existing socket-based setups keep working.

Other fixes

- **chore(python):** `pyproject.toml` had drifted to `3.10.41` while `__init__.py` read `3.12.1`. Synced both to 3.12.2 so `uv build` and runtime introspection now agree.
- **chore(claude.md, all 4):** stale framework version banners in `CLAUDE.md` headers updated.

No `.env` changes from 3.12.1, no migration needed. Existing 3.12.1 installs upgrade by changing one version number.

v3.12.1 (2026-05-04)

CI-only patch — no framework code changes from 3.12.0.

- **fix(ci, all 4):** every `publish.yml` workflow now declares `permissions: contents: write` on the publish job. Without this, `softprops/action-gh-release` 403'd against the default `GITHUB_TOKEN` on repos whose default Workflow permissions setting was read-only (Ruby and Node hit this every release; PHP and Python worked by luck of repo settings). The explicit declaration makes the workflow self-sufficient.
- **chore(ci):** bumped `softprops/action-gh-release` from `@v1` (unmaintained) to `@v2`.

No `.env` changes, no API changes, no migration needed. Existing 3.12.0 installs can upgrade without touching anything else.

The version-bump itself is the test: a successful 3.12.1 release proves the workflow fix works on Ruby and Node where 3.12.0 needed manual `gh release create`.

v3.12.0 (2026-05-04)

> ■■ ***Breaking change — read before upgrading.*** *Every framework env var now uses the `TINA4_` prefix. Existing `.env` files set with `DATABASE_URL`, `SECRET`, `SMTP_HOST`, `HOST_NAME`, etc. will cause the framework to refuse to boot. Run `tina4 env --migrate` to rewrite, or follow the rename table below.*

Why this release

Tina4's env vars had grown inconsistent. Some had the `TINA4_` prefix (`TINA4_DEBUG`, `TINA4_LOCALE`, `TINA4_CACHE_BACKEND`), others didn't (`DATABASE_URL`, `SECRET`, `SMTP_HOST`). Newcomers had to guess which convention applied to which feature. Existing tools and PaaS dashboards collided with un-prefixed names like `SECRET` and `API_KEY` that other libraries also read. Documentation drifted — 91 env-var names appeared in the docs that didn't exist in any framework, and 22 framework-specific env vars in the code didn't match the names users were told to set.

This release closes all three gaps with a single hard rename. No deprecation period, no fallback chain. The framework refuses to boot if it detects a legacy name in the environment, prints a list of every var to rename, and tells you which command to run.

What changed

• **22 env vars renamed** to `TINA4_*` form. See the migration table below.

• `tina4 env --migrate` **CLI** added to all four frameworks. Reads your `.env`, rewrites it in place, leaves a `.env.bak` backup, prints a diff. Idempotent.

• **Boot-time guard** scans `os.environ` (or the language equivalent) for the 22 legacy names. If any are present, prints the rename map and exits with code 2. Bypass with `TINA4_ALLOW_LEGACY_ENV=true` for migration scripts that need both names set during transition.

• **All 4 framework books rewritten.** Chapter 33 (Environment Variables) is now a clean canonical list — every var prefixed, descriptions current, legacy names removed.

• **Doc-vs-code drift closed.** Of the 91 stale env vars previously documented, 61 were renames (corrected), 32 were never implemented (removed). The `audit-links.py` CI gate stays at 0 broken links / 0 broken anchors.

• **Frond bundle** rebuilt at v2.1.3 — `frond.min.js` footer now shows the version explicitly so users can verify what they have.

Bug fixes shipped alongside the rename

• **#38 PostgreSQL UUID-PK transaction abort** — the post-INSERT `lastval()` probe is now wrapped in a SAVEPOINT, so UUID-PK INSERTs no longer poison the outer transaction with `InFailedSqlTransaction`. Live regression test against PostgreSQL 16. (Affects all 4 frameworks where the PG adapter does this probe.)

• **#39 Landing page + template auto-routing** —

• Auto-routing now scans `src/templates/pages/` only. Partials, layouts, base.twig, errors/, components/, and `_*` files never auto-serve from a URL.

• `TINA4_TEMPLATE_ROUTING=off` kills the feature entirely.

• `src/public/index.html` auto-serves at `/` (and `/foo/` serves `src/public/foo/index.html`) — SPA hosting Just Works.

• The framework landing page only renders when `TINA4_DEBUG=true`. Production never shows it; framework version, dev-admin link, and gallery don't leak to real users.

• The malformed `HTTP/1.1 404 OK` status line is fixed — every status code now uses its canonical RFC 7231/9110 reason phrase.

• **#37 frond.form.submit redirect handling** — verified shipped at v2.1.x; `xhr.responseURL` change triggers `window.location` navigation correctly.

• **#36 Session file handler** — re-verified safeguards (lazy save, WebSocket skip, probabilistic GC, new-and-empty skip) all still in place.

Migration — every renamed var

Legacy name	New name
`DATABASE_URL`	`TINA4_DATABASE_URL`
`DATABASE_USERNAME`	`TINA4_DATABASE_USERNAME`
`DATABASE_PASSWORD`	`TINA4_DATABASE_PASSWORD`
`DB_URL`	`TINA4_DATABASE_URL` (alias dropped)
`SECRET`	`TINA4_SECRET`
`API_KEY`	`TINA4_API_KEY`
`JWT_ALGORITHM`	`TINA4_JWT_ALGORITHM`
`SMTP_HOST`	`TINA4_MAIL_HOST`
`SMTP_PORT`	`TINA4_MAIL_PORT`
`SMTP_USERNAME`	`TINA4_MAIL_USERNAME`
`SMTP_PASSWORD`	`TINA4_MAIL_PASSWORD`
`SMTP_FROM`	`TINA4_MAIL_FROM`
`SMTP_FROM_NAME`	`TINA4_MAIL_FROM_NAME`
`IMAP_HOST`	`TINA4_MAIL_IMAP_HOST`
`IMAP_PORT`	`TINA4_MAIL_IMAP_PORT`
`IMAP_USER`	`TINA4_MAIL_IMAP_USERNAME`
`IMAP_PASS`	`TINA4_MAIL_IMAP_PASSWORD`
`HOST_NAME`	`TINA4_HOST_NAME`
`SWAGGER_TITLE`	`TINA4_SWAGGER_TITLE`
`SWAGGER_DESCRIPTION`	`TINA4_SWAGGER_DESCRIPTION`
`SWAGGER_VERSION`	`TINA4_SWAGGER_VERSION`
`ORM_PLURAL_TABLE_NAMES`	`TINA4_ORM_PLURAL_TABLE_NAMES`

Names that stay un-prefixed (not framework config)

`PORT`, `HOST`, `NODE_ENV`, `RACK_ENV`, `RUBY_ENV`, `ENVIRONMENT` — these are runtime / PaaS conventions, not framework config. Heroku, Railway, Vercel, and friends set them; we keep reading them.

How to upgrade

- **Backup your `.env`:** `cp .env .env.bak.pre-v3.12`
- **Run the migration:** `tina4 env --migrate` — rewrites your `.env` in place.
- **Update PaaS dashboards:** Heroku, Railway, Vercel, Render, Fly.io etc — rename the same vars in your provider's env-var UI.
- **Restart your app.** The boot guard verifies nothing legacy remains.

If your app uses `SECRET`, `DATABASE_URL`, or any other listed name in places besides `.env` (e.g. your CI pipeline's `env:` blocks), update those too — the boot guard checks `os.environ`, not just `.env`.

Parity

All 4 frameworks aligned at **3.12.0**:

- tina4-python 3.11.32 → 3.12.0
- tina4-php 3.11.32 → 3.12.0
- tina4-ruby 3.11.32 → 3.12.0
- tina4-nodejs 3.11.32 → 3.12.0

Coordinated release across PyPI, Packagist, RubyGems, npm.

v3.11.32 (2026-04-25)

Critical fix — pool + transactions are now actually atomic. Plus a coordinated parity release that aligns all four frameworks at the same version after months of drift.

Before this release, creating a `Database` with `pool > 0` silently broke transactions. The pool's round-robin checkout rotated to a different adapter on every call — so `start_transaction()` pinned its flag on adapter A, the executes autocommitted on adapters B and C, and the final `commit()` / `rollback()` landed on adapter D, which had nothing to commit. Result: `rollback()` was a no-op, writes leaked through, and no error or log surfaced the problem.

The fix pins one adapter to the calling context for the lifetime of a transaction. Each language uses its own primitive:

- **Python** — `threading.local()` on the `Database` instance
- **Ruby** — `Thread.current[:tina4_pinned_adapter_<obj_id>]`
- **Node.js** — `AsyncLocalStorage` from `node:async_hooks` (async-safe across overlapping awaits)
- **PHP** — per-instance property (PHP-FPM is one process per request; threading.local is unnecessary)

While pinned, every database call routes to the same adapter. `commit()` and `rollback()` release the pin so subsequent calls round-robin again.

- **fix (database / all 4):** adapter pinning across transaction scope in `Database._get_adapter()` (and language equivalents). Every backend is affected — SQLite, PostgreSQL, MySQL, MSSQL, Firebird. Firebird exposed it loudest because of its honest "commit-empty-txn is a real no-op" semantics; the others mostly hid the bug behind eager autocommits but still lost rollback atomicity.
- **tests (all 4):** new regression suite — three INSERTs followed by `rollback()` under `pool=4` now leaves zero rows (was leaking three). Three INSERTs followed by `commit()` persists exactly three. Pin-release after commit/rollback verified. `pool=0` regression test added so single-connection mode stays unaffected.
- **parity / version alignment:** all 4 frameworks bumped to 3.11.32 — closes the cross-framework version drift that had built up (PHP at 3.11.31, Python at 3.11.24, Ruby and Node at 3.11.19). A single coordinated release across all four registries: PyPI, Packagist, RubyGems, npm.

No migration needed. Code using `pool=0` (the default for every adapter except where explicitly raised) is unaffected. Code using `pool>0` will now actually honour transactions instead of silently dropping them.

If you've been seeing intermittent "writes vanished" or "rollback didn't help" reports on a pooled `Database`, this release is the cause and the cure.

v3.11.13 (2026-04-16)

Issue-driven release. Everything reported in the open tina4-book issues either was fixed in this version or is already fixed in 3.11.12; this release consolidates the remaining bits and corrects documentation drift.

- **feat (router / all 4):** Explicit typed-parameter system shared across Python, PHP, Ruby, Node. Adds `alpha`, `alnum`, `slug`, `uuid`, and explicit `string` types in addition to the existing `int`/`integer`, `float`/`number`, `path`/`.*`. **Unknown type names now throw at registration** — `{name:str}`, `{id:inetger}`, etc. raise with a clear message listing the valid types instead of silently falling through to the default matcher. Fixes tina4-book#125. +45 new tests across the four suites.
- **fix (gallery / python+php+ruby):** Gallery Try-It / View buttons now open the deployed example in a new tab (`window.open(url, '_blank')`) instead of navigating away from the gallery home. Fixes tina4-book#115.
- **fix (ruby gemspec):** `sqlite3` promoted from `add_development_dependency` to `add_dependency`. Matches the "zero-config SQLite on first run" promise. Fixes tina4-book#100.
- **docs (tina4-book):** PHP Chapter 2 updated — correct port (7145), `->noAuth()` on write-method examples, and an explicit callout explaining the secure-by-default policy for POST/PUT/PATCH/DELETE. Addresses tina4-book#87, #94, #123.
- **docs (tina4-book):** Python `@template` decorator ordering corrected (must sit BELOW the route decorator) in book chapters 04 and 10; Python `request->query` vs

`request->params` distinction in PHP chapter 1.

• **tests (python):** Session-handler tests updated to reflect the real default TTL of 3600s (were stale at 1800s).

• **verified already fixed in earlier 3.11.x releases** — closed comments posted on all of these:

• #79 dotted numeric index (`{{ items.0.name }}`)

• #80 `truncate` filter

• #82 `{{ parent() }}` / `{{ super() }}` across all 4 frameworks

• #83 Ruby dashboard — WEBrick is runtime dep

• #89 `load_dotenv` rename, `DatabaseResult` methods, SQLite WAL locking

• #91 Ruby `request.params` symbol + string keys via `IndifferentHash`

• #93 Ruby `/docs/*` and bare `/*` wildcard routes

• #97 Frond ternary operator

• **parity:** All 4 frameworks bumped to 3.11.13.

v3.11.12 (2026-04-16)

Breaking: `sqlite:///X` URLs are now relative to the project root (cwd), matching the documented convention. For absolute paths use four slashes (`sqlite:////abs/path.db`) or a Windows drive letter (`sqlite:///C:/Users/app.db`).

Before this release, `TINA4_DATABASE_URL=sqlite:///data/app.db` was interpreted differently by every framework. Python/Node/Ruby tried to open `/data/app.db` (absolute) which crashed on macOS with `OSError: [Errno 30] Read-only file system: '/data'`. PHP did the same under the hood. All four frameworks now agree: three slashes = relative, four slashes = absolute.

• **fix (all 4):** `sqlite:///X` resolves under cwd; parent directory auto-created only when inside cwd. Absolute paths are trusted and never mkdir'd at root.

• **fix (python):** `_ensure_folders` no longer creates a bogus `src/migrations/` directory. The migration runner always looks at `migrations/` at the project root — there is only one correct location.

• **parity (php, ruby, node):** Same `sqlite:///X` parsing as Python. Dedicated `resolve_path` / `resolveSqlitePath` helpers in each framework so adapters consistently handle `:memory:`, `./` forms, Windows drive letters.

• **tests:** 9 new Python tests in `TestSQLiteConnectionPath` + `TestProjectFolders`. 4 new PHP tests in `DatabaseUrlTest` covering relative/absolute/Windows/bruce-regression. 6 new Ruby specs in `database_drivers_spec.rb :: SqliteDriver.resolve_path`. Node URL tests expanded in `database.test.ts` with the full relative/absolute/Windows/:memory: matrix.

• **parity:** All 4 frameworks bumped to 3.11.12.

Migration note: If your `.env` has `TINA4_DATABASE_URL=sqlite:///data/app.db`, it will now create `./data/app.db` in the project root (which is what most users actually want). If you

genuinely want an absolute path, change to `sqlite:////data/app.db` (four slashes).

v3.11.11 (2026-04-16)

• **fix (python ORM):** `Field.validate` no longer re-coerces values that are already the correct type. Previously, any PostgreSQL/MSSQL read of a row containing a `DateTimeField` crashed because `datetime(datetime_instance)` raises `TypeError`. The fix accepts native driver types (`datetime`, `bytes`, `int`, `bool`, `float`, `str`) without re-wrapping, and parses ISO-8601 strings into `datetime` for SQLite. See `tina4-python/plan/orm-field-validate-native-types.md`.

• **fix (python ORM):** `BooleanField` vs `IntegerField` ordering handled explicitly. `BooleanField(1)` still coerces to `True`, `IntegerField(True)` still coerces to `1`; no regression for either direction (bool is a subclass of int in Python).

• **tests (python):** 10 new `TestFieldsNativeTypes` cases covering datetime/int/bool/float/bytes/string/ForeignKey round-trips.

• **tests (parity):** Regression-guard "datetime round-trip on read path" tests added to PHP (`ORMV3Test`), Ruby (`orm_spec`) and Node.js (`orm.test.ts`) so an equivalent bug can't creep in there later.

• **parity:** All 4 frameworks bumped to 3.11.11.

v3.11.10 (2026-04-15)

• **fix (php):** Hot-reload loop — DevAdmin's polling fallback used `mt=0` as the baseline, so the first poll after every page load triggered `location.reload()`, which reset `mt=0` again. Loop now initialises the baseline on the first poll.

• **fix (php):** Reload sentinel removed — PHP was the only framework recursively walking `src/` and touching `src/.reload_sentinel` on every reload POST. The sentinel lived inside the Rust CLI's watched tree and fed back into the watcher, triggering a second loop. Replaced with the same in-memory counter used by Python/Ruby/Node.

• **fix (php):** Polling no longer starts more than once when the WebSocket reconnect retry budget is exhausted (added a `pollStarted` guard).

• **feat (parity):** `GET /__dev/api/queue/topics` and `GET /__dev/api/queue/dead-letters` added to PHP, Ruby and Node (previously only in Python). PHP queue endpoints now read from the real `Tina4\Queue` backend instead of returning stubs.

• **feat (devadmin):** Refreshed `tina4-dev-admin.js` bundle (87.8 KB) across all 4 frameworks — adds the topic selector dropdown, inline payload expand/copy, and corrected version display.

• **tests:** 4-way parity tests for hot-reload: mtime starts at 0, POST /__dev/api/reload bumps the counter, no sentinel file is written to disk, mtime is monotonic across successive reloads. Mirrored in `tina4-php/tests/DevAdminTest.php`, `tina4-python/tests/test_dev_admin.py`, `tina4-ruby/spec/dev_admin_spec.rb`, `tina4-nodejs/test/devAdmin.test.ts`.

• **parity:** All 4 frameworks bumped to 3.11.10.

v3.11.9 (2026-04-15)

Catch-up release covering v3.11.0 → v3.11.9 across all 4 frameworks.

• **feat (websocket):** Full WebSocket parity across Python/PHP/Node/Ruby — `get_client_rooms()` / `getClientRooms()`, `route()` usable as decorator or direct handler registration, matching room/broadcast semantics, plus new parity tests on all 4.

• **feat (graphql):** Input validation and field-level `@auth` directives with context threading.

• **feat (graphql):** Auto-discovery of schemas; removed legacy DevAdmin HTML/JS in favour of the new UI.

• **feat (devadmin — Python):** Queue tab with topic selector, dead-letter listing and replay endpoints, inline payload expand/copy, version display.

• **feat (cli):** Rust CLI now owns file watching — frameworks receive `POST /__dev/api/reload` and internal watchers are disabled when launched by the Rust CLI (`--managed`).

• **fix (cli):** `parseFlags` / `parse_flags` / `parseCliArgs` no longer swallow `host:port` or positional args after boolean flags.

• **fix (scss):** SCSS recompilation loop fixed; output path corrected to `src/public/css/` to match CLI and static serving.

• **fix (frond — Python):** Numeric dotted index for lists (`items.0.name`) now resolves correctly.

• **fix (router — Ruby):** Bare `/*` wildcard capture exposed under `"*"` key for parity.

• **fix (orm — PHP):** Three data-sync bugs fixed: `load()` double-fill, `getPrimaryKeyValue`, `save()` ID sync.

• **fix (graphql):** `from_orm` / `fromOrm` list resolver used `select(skip=)` instead of `all(offset=)`.

• **fix (metrics):** Windows backslash paths normalised to forward slashes.

• **fix (app — PHP):** No longer crashes on notices/deprecations in loaded files; `run()` now prints the banner when starting the server directly.

• **chore:** Example demo store ships with the repo; Windows-friendly setup; `.env.example` and setup scripts added.

• **parity:** All 4 frameworks bumped to 3.11.9. PHP aligned to the 3.x tag scheme on `v3`.

v3.10.99 (2026-04-12)

• **breaking:** `auto_map` now defaults to `True` — ORM models automatically map between camelCase properties and snake_case DB columns. Set `auto_map = False` on your model to restore the old behaviour.

• **feat:** `to_dict(case=)` parameter — pass `case='camel'` to get camelCase keys (for JSON APIs) or `case='snake'` (default) for snake_case keys matching DB columns.

• **feat:** Frond `replace` filter now accepts dict args — `{{ v|replace({"T": " ", "-": "/"}) }}` for multiple substitutions in one call.

• **feat:** `background(callback, interval)` — register periodic tasks that run cooperatively in the asyncio event loop. Replaces `threading.Thread` for background work.

• **feat:** Background task protection — sync callbacks run in a `ThreadPoolExecutor` via `run_in_executor()` with `asyncio.wait_for()` timeout, preventing blocking functions from freezing the server.

• **feat:** Docker image now bundles the example store demo — `docker run tina4stack/tina4-python:v3` starts a working app out of the box.

• **fix:** Cart nav badge now updates reactively on quantity change and item removal (tina4-js `signal`/`computed`/`effect`).

• **fix:** Non-blocking queue consumer — `process_orders()` uses `queue.pop()` (single job per tick) instead of blocking `queue.consume()`.

• **tests:** 6 new parity tests covering `to_dict(case=)`, `auto_map` default, `replace` filter (dict + positional), and `background()` registration.

• **parity:** All features shipped identically across Python, PHP, Ruby, Node.js. 2,304 tests passing.

v3.10.97 (2026-04-11)

• **fix:** frond.form.submit redirect handling — XHR transparently follows 3xx redirects, so the callback received redirected HTML instead of navigating. Fixed by comparing `xhr.responseURL` with the original URL and calling `window.location.href` when a redirect is detected.

• **fix:** Currency placeholder — locale files now default to `$` for the currency symbol.

• **fix:** Admin sidebar alignment — widened sidebar to 220px with `min-width` to prevent label truncation.

• **fix:** Admin table overflow — added `min-width: 0` and `overflow-x: auto` on `.admin-main` to prevent content clipping.

• **fix:** Order detail template — corrected variable names (`items` instead of `order.items`, `item.name` instead of `item.product_name`) and used `.records` from `DatabaseResult`.

• **fix:** Status badges — dashboard recent orders and order list now show colored badge pills with translated status labels (pending, processing, shipped, delivered, cancelled).

• **fix:** Date formatting — admin order/dashboard dates trimmed to `YYYY-MM-DD HH:MM:SS` instead of raw ISO with microseconds.

• **feat:** Cart quantity spinner — reactive qty controls using tina4-js signals, computed values, and effects.

• **feat:** Multi-currency pricing — forex conversion via Api client (frankfurter.app), `|currency` template filter, currency selector in navbar.

• **feat:** MCP server tools — `check_stock`, `low_stock_report`, `search_products` tools and `store://categories`, `store://inventory-summary` resources for AI assistant integration.

• **feat:** Contact form — built with `HtmlElement` and `add_html_helpers()` to demonstrate programmatic HTML generation.

• **feat:** ORM named scopes — `Product.scope("active")`, `Product.scope("low_stock")`, `Product.scope("expensive")`.

• **feat:** Database connection pooling — `Database("sqlite:data/store.db", pool=4)`.

• **feat:** Inline tests — `@tests` decorators on `cart_service.py` and `forex_service.py`.

• **feat:** Language toggle — flag button (■■/■■) in navbar to switch locale.

• **feat:** Helpdesk chat persistence — chat messages stored in DB, history API (`GET /api/chat/history`).

• **dep:** Updated frond.min.js to v2.1.2 across all 4 frameworks (Python, PHP, Ruby, Node.js).

• **parity:** All 4 frameworks bumped to 3.10.97.

v3.10.93 (2026-04-11)

• **fix:** Frond array/dict literal support — `{% set items = ["a", "b"] %}` and `{% set obj = {"k": "v"} %}` now parse correctly.

• **fix:** Frond bracket depth tracking in `_find_outside_quotes()` and `_split_outside_quotes()` — expressions like `arr[i % 2]` no longer treated as top-level arithmetic.

• **fix:** Frond subscript expression evaluation — bracket content uses `_eval_expr()` instead of `_resolve()`, enabling `arr[loop.index0 % 2]`.

• **fix:** Frond slice with variable bounds — `items[start:end]` evaluates bounds through `_eval_expr()`.

• **fix:** Frond multiline `{% set %}` — `_SET_RE` regex now uses `re.DOTALL` flag.

• **docs:** Developer skills updated — Metrics Dashboard guidance, Frond Template Parity rules, `@noauth` security warnings.

• **demo:** Complete e-commerce store example (`example/store/`) with GraphQL search, SSE, WebSocket, Queue, Events, 13 test files.

• **parity:** All Frond fixes applied identically across Python, PHP, Ruby, Node.js. 2,304 tests passing.

v3.10.92 (2026-04-10)

• **refactor:** Extract `RateLimiter` from `core/middleware.py` into its own file `core/rate_limiter.py`. The old import path still works via re-export.

• **feat:** Add `RateLimiterMiddleware` wrapper class with `before_rate_limit()` and `check()` static methods.

• **breaking:** Rename `ErrorOverlay` methods — `render()` → `render_error_overlay()`, `render_production()` → `render_production_error()`, `debug_mode()` → `is_debug_mode()`.

• **feat:** Add `Server.start()` and `Server.stop()` for cross-framework parity.

• **feat:** Add `DatabaseResult.size()`, `to_array()`, `to_json()`, `to_csv()` methods.

• **feat:** Add `ScssCompiler` class with `compile()`, `compile_file()`, `add_import_path()`, `set_variable()`.

• **feat:** Add `DevAdmin.unresolved_count()`, `clear_all()`, `reset()`, `capture()` (5-param), `register()`.

• **fix:** GraphQL test API — update `add_query()` calls to use positional args (args, return_type, resolver).

• **parity:** 44/44 cross-framework features green. 2,263 tests passing.

v3.10.91 (2026-04-10)

• **feat:** Add parity methods — `GraphQLType.parse()`, `Response.send()` params, `QueryBuilder.from_()`, `Debug.configure()`.

• **breaking:** Remove alias methods `from_`, `configure`, `template` — use canonical names only (`from_table`, etc.).

v3.10.90 (2026-04-09)

• **docs:** Chapter 4 (Templates) — new "Dumping Values for Debugging" section covering both `{{ x|dump }}` and `{{ dump(x) }}` forms, their shared `<pre>`-wrapped output, and the `TINA4_DEBUG=true` production gate. Filter table entry updated to reference the new section.

• **docs:** `plan/parity/parity-template.md` updated with a cross-framework dump helper comparison table and marks dump parity as confirmed across all 4 frameworks at v3.10.89.

• **chore:** Version sync release — brings all 4 frameworks to the same patch version (3.10.90) so downstream users can upgrade PHP/Python/Ruby/Node.js in lockstep without hunting version mismatches.

v3.10.89 (2026-04-09)

• **feat:** `{{ dump(value) }}` global function form added to Frond alongside the existing `{{ value|dump }}` filter. Both call a single `_render_dump()` helper and produce identical output.

• **security:** Dump is now **gated on** `TINA4_DEBUG=true`. In production (env var unset or `false`) both the filter and function silently return an empty string. This prevents accidental leaks of internal state, object shapes, and sensitive values into rendered HTML when a developer leaves a `{{ dump(x) }}` call in a template.

• **refactor:** Dump output is wrapped in `<pre>` and HTML-escaped via a single shared code path.

• **test:** 6 new tests in `test_frond.py` (`TestDump`) covering debug-mode output, production silencing, unset-env default-to-production, function/filter parity, and circular references.

v3.10.86 (2026-04-09)

• **feat:** `ForeignKeyField` is now a proper `Field` subclass that auto-wires both sides of the relationship. Declaring `author_id = ForeignKeyField(to=Author)` injects `belongs_to` on the declaring model and `has_many` on the referenced model via `ORMMeta` — no manual descriptor calls required. Override the has-many name with `related_name=`.

• **feat:** Cross-framework parity — same FK auto-wiring semantics now available in PHP (`$foreignKeys`), Ruby (`foreign_key_field`), and Node.js (`type: "foreignKey"`)

• **fix:** `@orm_bind(db)` no longer nulls the decorated class — returns a pass-through decorator

• **fix:** `Auth.get_token`/`valid_token`/`get_payload`/`refresh_token`/`authenticate_request` can now be called on the class (e.g. `Auth.get_token(payload)`) or on an instance via the `_DualMethod` descriptor

• **fix:** `SQLiteAdapter` uses a class-level `threading.Lock` + `PRAGMA busy_timeout = 30000` + `timeout=30` on connect to eliminate `SQLITE_BUSY` deadlocks in the dev server under concurrent writes

• **docs:** Chapter 6 (ORM) updated with a new "ForeignKeyField — Auto-Wired Relationships" section

v3.10.85 (2026-04-09)

• **refactor:** Split queue adapters into separate files — `queue/rabbitmq_backend.py`, `queue/kafka_backend.py`, `queue/mongo_backend.py` (one class per file, aligning with PHP/Node/Ruby architecture)

• **fix:** Updated remaining tests to use bool `valid_token()` + `get_payload()` pattern

v3.10.84 (2026-04-09)

• **fix:** Router/middleware was setting `request.user` / `request.auth` / auth payload to `true` (boolean) instead of the actual JWT payload dict after `validToken()` was changed to return bool — any code reading `request.user["sub"]` etc. would have failed silently or crashed

• **fix:** CSRF middleware was not correctly rejecting invalid tokens (nil check on bool result always passed)

• **fix:** `AuthMiddleware.before_request` called `get_payload` incorrectly — would TypeError at runtime on valid token

• **add:** Headless routing auth payload integration tests to prevent regression

v3.10.83 (2026-04-08)

• **fix:** prevent orphaned session files on WebSocket and anonymous requests (#36)

• **feat:** WebSocket rooms — `join_room`, `leave_room`, `broadcast_to_room`, `room_count`, `get_room_connections`

• **feat:** queue signature parity — instance-scoped `push`/`pop`/`retry`, no topic params on public methods

v3.10.70 (2026-04-06)

• **New:** SSE (Server-Sent Events) support via `response.stream()` — pass a generator, framework handles chunked transfer encoding, keep-alive, and `text/event-stream` content type

• **New:** Chapter 24 added to documentation: Server-Sent Events

• Feature count: 45 (was 44)

• Full parity across Python, PHP, Ruby, Node.js

Version History

Tina4 Python follows semantic versioning. The major version (3) marks the ground-up rewrite from v2. Minor versions (3.1, 3.2, ...) introduce features. Patch versions fix bugs and polish edges.

Every release ships through PyPI. Upgrade with:

```
uv add tina4-python@latest

# or with pip

## v3.10.70 (2026-04-06)

- **New:** SSE (Server-Sent Events) support via `response.stream()` — pass a generator, framewor
- **New:** Chapter 24 added to documentation: Server-Sent Events
- Feature count: 45 (was 44)
- Full parity across Python, PHP, Ruby, Node.js

pip install --upgrade tina4-python
```

Check your current version:

```
import tina4_python
print(tina4_python.__version__)
```

---

v3.10.68 (2026-04-03) — Full Parity Release

• **100% API parity** across Python, PHP, Ruby, Node.js — 30+ issues fixed

• **ORM:** save() returns self/false, all/select/where return arrays, toDict/toAssoc standardized, scope() registers reusable method, where()/all() on Node, count() on PHP

• **Auth:** expires*in in minutes, PBKDF2 260k iterations, env TINA4*SECRET fallback, API key fallback in authenticateRequest

• **Session:** dual-mode flash(), get_flash/getFlash, cookieHeader on all, getSessionId on Node, save() public on Node

• **Database:** execute() returns bool/DatabaseResult for RETURNING, get/*ast*id/get_error on all, getColumns on PHP, cacheStats on Node

• **Request/Response:** Node files as dict, Python query property, cookies on PHP/Node, contentType on Node, xml() on PHP/Node, Ruby callable response

• **Queue:** consume() poll_interval (long-running generator with built-in sleep)

• **WebSocket:** event naming standardized (open/message/close/error), connection ip/headers/params, Python on() string API

• **GraphQL:** schema_sdl()/schemaSdl() and introspect() on all 4

• **WSDL:** Node.js zero-dep DOM parser (replaced regex)

• **Events:** emitAsync()/emit_async() on all 4

• **i18n:** zero-dep YAML locale file support on Python/PHP/Node (Ruby already had it)

v3.10.67 (2026-04-03)

• **BREAKING: request.files content is now raw bytes** — previously base64-encoded; remove any `base64.b64decode()` calls when saving uploaded files. Write `file["content"]` directly to disk

• **load() is now an instance method** — `model.load(sql, params)` calls selectOne internally, populates the instance, returns `True`/`False`. Use `Model.find(id)` for PK lookups

• **api.upload()** added to tina4-js — sends FormData with Bearer token auth for multipart file uploads

• **ORM CLAUDE.md rewrite** — all method stubs now match actual API signatures

• **tina4-js skill** — critical input binding warning, routing docs (`{param}` not `:param`), file upload pattern

v3.10.66 (2026-04-03)

• **Metrics file detail fix** — clicking bubbles in framework scanning mode now resolves paths correctly via scan root tracking

v3.10.65 (2026-04-03)

• **Metrics 3-stage test detection** — filename, path, and content matching

• **Metrics framework mode** — scans framework source with correct relative paths

• **tina4 console** — interactive REPL with framework loaded

• **tina4 env** — interactive environment configuration

• **Brand** — "TINA4 — The Intelligent Native Application 4ramework"

• **Quick references** — 36 sections, DotEnv API documented

• **37 chapters** — 7 new (Events, Localization, Logging, API Client, WSDL/SOAP, DI Container, Service Runner)

• **MongoDB + ODBC adapters** across all 4 frameworks

• **Pagination standardized** — limit/offset primary, merged dual-key response

• **Port kill-and-take-over** on startup

---

v3.10.60 (2026-04-03)

• **tina4 console** — interactive Python REPL with framework loaded (db, Router, ORM, Auth, Api, Log)

• **tina4 env** — interactive environment configuration (database, cache, session, queue, mail)

• **Brand update** — "TINA4 — The Intelligent Native Application 4ramework"

• **Quick reference** — 36 sections covering every framework feature

• **Chapter reshuffle** — 37 chapters, 7 new (Events, Localization, Logging, API Client, WSDL/SOAP, DI Container, Service Runner)

• **RouteGroup fix** — double prefix bug resolved

• **Port kill-and-take-over** — default port always reclaimed on startup

• **Metrics test detection** — expanded to check spec/, tests/, test/ directories

• **MongoDB adapter** (pymongo), **ODBC adapter** (pyodbc)

• **Pagination standardized** — limit/offset primary, merged dual-key response

• **9,138 tests** across all 4 frameworks

---

v3.10.57 (2026-04-02)

• **MongoDB adapter** — `Database("mongodb://host:port/db")`, requires `pip install pymongo`

• **ODBC adapter** — parity with PHP/Ruby/Node

• **RouteGroup class** — `group.get()`/`group.post()` syntax matching PHP/Ruby/Node

• **Pagination standardized** — limit/offset primary, merged dual-key toPaginate() response

• **Test port at +1000** — user testing port (e.g. 8145) stable, no hot-reload

• **Dynamic version** — `__version__` read at runtime, no hardcoded constants

• **ORM TINA4*DATABASE*URL discovery** — auto-connect from env

• **Firebird path parsing** — preserves absolute paths

• **108 features at 100% parity**, 2,112 tests

---

v3.10.54 (2026-04-02)

• **Auto AI dev port** — second listener on port+1 with no-reload when TINA4_DEBUG=true

• **TINA4*NO*RELOAD** env var + --no-reload CLI flag

• **Firebird path parsing** — preserve absolute paths

• **ORM TINA4*DATABASE*URL discovery** — auto-connect from env

• **SQLite transaction safety** — commit() no-op without transaction

- **QueryBuilder docs** — added to ORM chapter

---

v3.10.48 — April 2, 2026

Bug Fixes

Production server explicit opt-in — All frameworks now require an explicit `--production` flag to use production servers (Puma for Ruby, FrankenPHP for PHP, cluster mode for Node.js). Previously, production servers activated automatically when `TINA4_DEBUG=false`, which was surprising behaviour. Now `tina4 serve` always uses the dev server unless `--production` is passed.

Python `--no-browser` — The `run()` function now accepts `no_browser=True` and respects the `TINA4_NO_BROWSER` env var to prevent browser auto-opening on server start.

Test Coverage

Python: 2,132. PHP: 1,992. Ruby: 2,387. Node.js: 2,546. Total: 9,057 tests, 0 failures.

---

v3.10.46 — April 1, 2026

Test Coverage

Massive test parity push across all 4 frameworks. CSRF middleware tests expanded to 29+ per framework. Dedicated test suites added for FakeData, Cache, DevMailbox, Static files, Metrics, CLI scaffolding, and all remaining gap areas. Python: 2,132 tests. PHP: 1,937. Ruby: 2,274. Node.js: 2,546. Total: 8,889 tests, 0 failures, 49 core areas with full parity.

---

v3.10.45 — April 1, 2026

Bug Fixes

PHP CLI serve hijack — When `index.php` calls `App::run()`, the CLI `serve` command now sets a `TINA4_CLI_SERVE` constant so `run()` returns early, letting the CLI manage the server lifecycle (port, debug mode, browser open). Previously, `index.php`'s `run()` would start its own server and block the CLI's serve logic.

---

v3.10.44 — April 1, 2026

New Features

Database tab redesign — The dev admin Database panel now uses a split-screen layout. Tables are listed on the left as a navigation sidebar with click-to-select highlighting. The query editor, toolbar, and results occupy the right panel. Results render immediately below the query box with no gap.

Copy CSV / Copy JSON — Two new buttons in the database toolbar copy query results to the clipboard. CSV uses proper comma-separated format with quoting; JSON copies a formatted array of objects.

Paste data — A new Paste button opens a modal where you can paste JSON arrays or CSV/tab-separated data. The tool auto-detects the format and generates INSERT statements. If a table is selected on the left, it targets that table. If no table is selected, it prompts for a name and generates a CREATE TABLE statement for new tables. If you paste SQL directly, it passes through to the query box unchanged.

Multi-statement execution — The query runner now handles multiple SQL statements separated by semicolons. CREATE TABLE + INSERT batches run in a single transaction with automatic rollback on error.

Database badge on load — The Database tab count badge now shows the table count immediately when the dev admin opens, without needing to click the tab first.

Star wiggle animation — The GitHub star button on the landing page uses an empty star (■) with a playful wiggle animation: 3-second delay on page load, then wiggles at random 3–18 second intervals.

Bug Fixes

Default port — Python default port changed from 7145 to 7146 to avoid clashes when running multiple Tina4 frameworks (PHP=7145, Python=7146, Ruby=7147, Node=7148).

SQLite LIMIT fix — The SQLite adapter now checks if a query already contains a LIMIT clause before appending one, preventing double-LIMIT errors in the database browser.

browseTable quote escaping — Fixed broken onclick handlers for table names in the database panel. Now uses `addEventListener` instead of inline onclick with escaped quotes.

Migration UP/DOWN separation — Migration generator no longer puts DOWN SQL in the .sql file. UP SQL stays in the .sql file; DOWN SQL goes in the separate .down.sql file.

Test Coverage

Major test expansion across all 4 frameworks — 8,107 total tests (up from ~5,200), with full parity across 49 core feature areas. New dedicated test suites added for FakeData, Cache, DevMailbox, Static files, Metrics, and CLI scaffolding.

Python: 2,132 tests passing (12 skipped).

---

v3.10.40 — April 1, 2026

Bug Fixes

Dev overlay version check — Fixed misleading "You are up to date" message when running a version ahead of what's published on PyPI (e.g. running v3.10.39 locally while PyPI still has v3.10.24). The overlay now shows a purple "ahead of PyPI" message. Also added a breaking changes warning (red banner with changelog link) when a major or minor version update is available, so developers know to check for breaking changes before upgrading.

---

v3.10.39 — April 1, 2026

Breaking Changes

This release aligns the Python framework with the other three Tina4 implementations. Two breaking changes affect existing code:

`Auth.check_password()` **parameter order reversed**

```
# BEFORE (v3.10.38 and earlier)

## v3.10.70 (2026-04-06)

- **New:** SSE (Server-Sent Events) support via `response.stream()` — pass a generator, framewor
- **New:** Chapter 24 added to documentation: Server-Sent Events
- Feature count: 45 (was 44)
- Full parity across Python, PHP, Ruby, Node.js

Auth.check_password(hashed, password)

# AFTER (v3.10.39+)

## v3.10.70 (2026-04-06)

- **New:** SSE (Server-Sent Events) support via `response.stream()` — pass a generator, framewor
- **New:** Chapter 24 added to documentation: Server-Sent Events
- Feature count: 45 (was 44)
- Full parity across Python, PHP, Ruby, Node.js

Auth.check_password(password, hashed)  # password first — matches PHP, Ruby, Node.js
```

`Router.all()` **removed — use** `get_routes()` **or** `list_routes()`

```
# BEFORE

## v3.10.70 (2026-04-06)

- **New:** SSE (Server-Sent Events) support via `response.stream()` — pass a generator, framewor
- **New:** Chapter 24 added to documentation: Server-Sent Events
- Feature count: 45 (was 44)
- Full parity across Python, PHP, Ruby, Node.js

routes = Router.all()

# AFTER
```

```
## v3.10.70 (2026-04-06)

- **New:** SSE (Server-Sent Events) support via `response.stream()` — pass a generator, framewor
- **New:** Chapter 24 added to documentation: Server-Sent Events
- Feature count: 45 (was 44)
- Full parity across Python, PHP, Ruby, Node.js

routes = Router.get_routes()   # or Router.list_routes()
```

New Features

`Auth.validate_api_key(provided, expected=None)`

Compare API keys with constant-time comparison. Optionally pass `expected`; if omitted, reads `TINA4_API_KEY` (or `TINA4_API_KEY`) from environment.

```
Auth.validate_api_key("sk-abc123")              # check against env
Auth.validate_api_key("sk-abc123", "sk-abc123") # check against explicit value
```

`Auth.authenticate_request(headers)`

One-call header authentication: checks Bearer JWT, Bearer API key, and Basic auth in order.

```
payload = Auth.authenticate_request(request.headers)
# Returns: dict on success, None on failure
```


```
## v3.10.70 (2026-04-06)

- **New:** SSE (Server-Sent Events) support via `response.stream()` — pass a generator, framewor
- **New:** Chapter 24 added to documentation: Server-Sent Events
- Feature count: 45 (was 44)
- Full parity across Python, PHP, Ruby, Node.js
```


`ORM.find_by_id(pk)` **with** `find()` **and** `load()` **as aliases**

`find_by_id()` is now the explicit primary method. Both `find()` and `load()` continue to work as aliases, ensuring backward compatibility.

Test Coverage

2,054 tests passing (up from 2,051 in v3.10.38).

---

v3.10.38 — April 1, 2026

Code Metrics & Bubble Chart

The dev dashboard (`/__dev`) now includes a **Code Metrics** tab with a PHPMetrics-style bubble chart visualization. Files are represented as animated bubbles — sized by lines of code, colored by maintainability index (green = healthy, red = needs attention). Click any bubble to drill down into per-function cyclomatic complexity.

The metrics engine uses Python's `ast` module for zero-dependency static analysis:

- Cyclomatic complexity per function
- Halstead volume and maintainability index

- Afferent/efferent coupling and instability
- Violation detection (CC > 20, LOC > 500, MI < 20)

File analysis is sorted worst-first so problem areas surface immediately. Results are cached for 60 seconds with mtime-based invalidation.

AI Context Installer

The `tina4python ai` command now presents a simple numbered menu instead of unreliable auto-detection:

```
1. Claude Code        CLAUDE.md              [installed]
2. Cursor             .cursorules
3. GitHub Copilot     copilot-instructions.md
...
8. Install tina4-ai tools (requires Python)
```

Select by number (comma-separated or `all`). Already-installed tools show green. The generated context now includes the full skills table across all frameworks.

Dashboard Improvements

- Full-width layout (removed 1400px max-width constraint)
- Sticky header and tab bar when scrolling
- Dashboard overlay fills the screen (was constrained to 1200px)

Cleanup

- Removed `demo/` directories from all framework repos (demos live in documentation)
- Removed old `plan/` spec documents, replaced with `PARITY.md` and `TESTS.md`
- Removed junk sample files (broken migrations, test templates)
- Central parity matrix added to tina4-book

---

v3.10.x — Previous Releases (March 28-31, 2026)

The v3.10 line carries the most patches of any minor release. It refined the Frond template engine, hardened the ORM, and completed cross-framework parity.

Highlights

- **Singleton Frond engine** (v3.10.0). The template engine creates one instance and reuses it. Previous versions spawned a new engine per render call. This cut template rendering overhead across the board.
- **ORM `auto_map` flag** (v3.10.1). Models translate between `snake_case` and `camelCase` column names without manual mapping.

```
from tina4_python import ORM

class UserProfile(ORM):
    auto_map = True  # created_at ↔ createdAt handled for you
```

```
    table_name = "user_profiles"
```

- **Frond method calls and slice syntax** (v3.10.2). Templates gained the ability to call methods on objects and use Python slice syntax.

```
<!-- Method calls on template variables -->
{{ user.get_display_name() }}

<!-- Slice syntax -->
{{ long_text[:100] }}...
```

- **Frond quote-aware operator matching** (v3.10.5). Operators inside quoted strings no longer break the parser. Before this fix, a string containing `>=` or `==` could confuse the template engine.

- **ORM auto-commit on write operations** (v3.10.13). Save, delete, and update operations commit their transactions without an explicit call.

- `to_json` **and** `js_escape` **filters** (v3.10.16). Templates gained filters for safe JSON embedding and JavaScript string escaping.

```
<script>
  const config = {{ settings|to_json }};
  const message = "{{ user_input|js_escape }}";
</script>
```

- `formTokenValue()` **helper** (v3.10.23). CSRF tokens gained a dedicated template function for cleaner form markup.

```
<form method="post">
  <input type="hidden" name="form_token" value="{{ formTokenValue() }}">
  <!-- form fields -->
</form>
```

- **MCP server and TestClient parity** (v3.10.32). The built-in MCP server and integration test client reached feature parity with the PHP and Ruby implementations.

- **Arithmetic in** `{% set %}` **and expressions** (v3.10.31). The template engine handles math operations inside assignment blocks.

```
{% set total = price * quantity + shipping %}
```

Bug Fixes

Middleware not applied to routes (fixed in v3.10.1). Middleware functions registered with `@middleware` were silently skipped during route dispatch. Routes ran without their middleware.

```
# Before (broken) — middleware was silently skipped

## v3.10.70 (2026-04-06)

- **New:** SSE (Server-Sent Events) support via `response.stream()` — pass a generator, framewor
- **New:** Chapter 24 added to documentation: Server-Sent Events
- Feature count: 45 (was 44)
- Full parity across Python, PHP, Ruby, Node.js

@app.middleware
def log_request(request, response):
    print(f"Request: {request.method} {request.url}")
    return request, response
```

```
@app.get("/users")
def get_users(request, response):
    # middleware never ran
    return response("OK")

# After (fixed in v3.10.1) — middleware runs on every matching route

## v3.10.70 (2026-04-06)

- **New:** SSE (Server-Sent Events) support via `response.stream()` — pass a generator, framewor
- **New:** Chapter 24 added to documentation: Server-Sent Events
- Feature count: 45 (was 44)
- Full parity across Python, PHP, Ruby, Node.js

@app.middleware
def log_request(request, response):
    print(f"Request: {request.method} {request.url}")
    return request, response

@app.get("/users")
def get_users(request, response):
    # log_request fires before this handler
    return response("OK")
```

Wildcard route matching (fixed in v3.10.1). Routes with wildcard segments failed to match incoming requests. The router now handles wildcards as expected.

`Auth.valid_token` **reference error (fixed in v3.10.9).** The internal server module referenced the wrong attribute name when calling token validation, which caused a `TypeError` on every secured request. The fix resolved the reference so `Auth.valid_token` works as expected.

Frond `dict[variable_key]` **access (fixed in v3.10.11).** Accessing a dictionary with a variable key inside templates raised a `KeyError`. The engine now resolves the variable before the lookup.

```
<!-- Before (broken) — raised KeyError -->
{% set key = "name" %}
{{ user[key] }}

<!-- After (fixed in v3.10.11) — resolves variable, then looks up the key -->
{% set key = "name" %}
{{ user[key] }}  <!-- now outputs user["name"] -->
```

ORM transaction errors on SQLite (fixed in v3.10.25). Calling `save()` or `delete()` on an ORM model raised `"cannot commit -- no transaction is active"` on SQLite. The ORM now wraps every write operation in a proper `start_transaction` / `commit` / `rollback` cycle.

```
# Before (broken on SQLite)

## v3.10.70 (2026-04-06)

- **New:** SSE (Server-Sent Events) support via `response.stream()` — pass a generator, framewor
- **New:** Chapter 24 added to documentation: Server-Sent Events
- Feature count: 45 (was 44)
- Full parity across Python, PHP, Ruby, Node.js

user = User()
user.name = "Alice"
```

```
user.save()  # raised "cannot commit -- no transaction is active"

# After (fixed in v3.10.25) — save() wraps in a transaction automatically

## v3.10.70 (2026-04-06)

- **New:** SSE (Server-Sent Events) support via `response.stream()` — pass a generator, framewor
- **New:** Chapter 24 added to documentation: Server-Sent Events
- Feature count: 45 (was 44)
- Full parity across Python, PHP, Ruby, Node.js

user = User()
user.name = "Alice"
user.save()  # works on all database engines including SQLite
```

Stale templates in dev mode (fixed in v3.10.24). The dev server cached rendered templates and did not pick up file changes until restart. Templates now reload on every request in debug mode.

Macro output HTML escaping (fixed in v3.10.27). Frond macros returned raw strings that the auto-escaper then double-escaped. Macro output now wraps in `SafeString` to preserve the intended HTML.

DevReload performance (fixed in v3.10.28). The live-reload watcher polled too fast and triggered duplicate reloads. It now uses a 3-second default interval with debouncing.

Breaking Changes

`@noauth` **and** `@secured` **decorator behavior (v3.10.1).** Before this fix, these decorators did not update the route's auth flags. If you relied on the broken behavior (routes ignoring auth decorators), your routes will now enforce authentication as intended.

```
# Before (broken) — decorator had no effect

## v3.10.70 (2026-04-06)

- **New:** SSE (Server-Sent Events) support via `response.stream()` — pass a generator, framewor
- **New:** Chapter 24 added to documentation: Server-Sent Events
- Feature count: 45 (was 44)
- Full parity across Python, PHP, Ruby, Node.js

@app.get("/public")
@noauth
def public_page(response):
    return response("Open to all")  # still required auth

# After (fixed in v3.10.1) — decorator works as expected

## v3.10.70 (2026-04-06)

- **New:** SSE (Server-Sent Events) support via `response.stream()` — pass a generator, framewor
- **New:** Chapter 24 added to documentation: Server-Sent Events
- Feature count: 45 (was 44)
- Full parity across Python, PHP, Ruby, Node.js

@app.get("/public")
@noauth
def public_page(response):
```

```
    return response("Open to all")  # accessible without token
```

---

v3.9.x — QueryBuilder and Sessions (March 26-27, 2026)

Features

QueryBuilder with fluent API (v3.9.0). SQL construction through method chaining, integrated with the ORM.

```
from tina4_python import ORM

class User(ORM):
    table_name = "users"

# Fluent query through the ORM

## v3.10.70 (2026-04-06)

- **New:** SSE (Server-Sent Events) support via `response.stream()` — pass a generator, framewor
- **New:** Chapter 24 added to documentation: Server-Sent Events
- Feature count: 45 (was 44)
- Full parity across Python, PHP, Ruby, Node.js

admins = User.query() \
    .where("role = ?", ["admin"]) \
    .order_by("name") \
    .limit(10) \
    .get()

# Standalone query

## v3.10.70 (2026-04-06)

- **New:** SSE (Server-Sent Events) support via `response.stream()` — pass a generator, framewor
- **New:** Chapter 24 added to documentation: Server-Sent Events
- Feature count: 45 (was 44)
- Full parity across Python, PHP, Ruby, Node.js

from tina4_python import QueryBuilder

results = QueryBuilder.table("orders") \
    .select("customer_id", "SUM(total) as revenue") \
    .where("status = ?", ["completed"]) \
    .group_by("customer_id") \
    .having("revenue > ?", [1000]) \
    .get()
```

Path parameter injection (v3.9.0). Route handlers receive path parameters as named function arguments. No more digging through `request.params`.

```
# Before v3.9.0

## v3.10.70 (2026-04-06)

- **New:** SSE (Server-Sent Events) support via `response.stream()` — pass a generator, framewor
- **New:** Chapter 24 added to documentation: Server-Sent Events
```

```
- Feature count: 45 (was 44)
- Full parity across Python, PHP, Ruby, Node.js

@app.get("/users/{id}")
def get_user(request, response):
    user_id = request.params["id"]
    return response(f"User {user_id}")

# v3.9.0 and later

## v3.10.70 (2026-04-06)

- **New:** SSE (Server-Sent Events) support via `response.stream()` — pass a generator, framewor
- **New:** Chapter 24 added to documentation: Server-Sent Events
- Feature count: 45 (was 44)
- Full parity across Python, PHP, Ruby, Node.js

@app.get("/users/{id:int}")
def get_user(id, request, response):
    return response(f"User {id}")  # id is already an int
```

Auto-start sessions (v3.9.0). Every route handler receives `request.session` with zero configuration. The session API covers `get`, `set`, `delete`, `has`, `clear`, `destroy`, `regenerate`, `flash`, and `get_flash`.

```
@app.get("/dashboard")
def dashboard(request, response):
    visits = request.session.get("visits", 0)
    request.session.set("visits", visits + 1)
    return response(f"Visit #{visits + 1}")
```

CSRF middleware and form tokens (v3.9.1). Session-bound CSRF tokens protect forms by default. Toggle with the `TINA4_CSRF` environment variable.

```
# CSRF is on by default in v3.9.1+

## v3.10.70 (2026-04-06)

- **New:** SSE (Server-Sent Events) support via `response.stream()` — pass a generator, framewor
- **New:** Chapter 24 added to documentation: Server-Sent Events
- Feature count: 45 (was 44)
- Full parity across Python, PHP, Ruby, Node.js

# Disable for API-only apps:

## v3.10.70 (2026-04-06)

- **New:** SSE (Server-Sent Events) support via `response.stream()` — pass a generator, framewor
- **New:** Chapter 24 added to documentation: Server-Sent Events
- Feature count: 45 (was 44)
- Full parity across Python, PHP, Ruby, Node.js

# TINA4_CSRF=false

## v3.10.70 (2026-04-06)

- **New:** SSE (Server-Sent Events) support via `response.stream()` — pass a generator, framewor
- **New:** Chapter 24 added to documentation: Server-Sent Events
- Feature count: 45 (was 44)
```

```
- Full parity across Python, PHP, Ruby, Node.js
```

File-based queue backend (v3.9.1). Replaced the SQLite queue with a JSON file-based backend. Zero dependencies. Full cross-platform parity with the PHP and Ruby implementations.

NoSQL QueryBuilder (v3.9.2). The fluent API gained `to_mongo()` to generate MongoDB queries from the same builder syntax.

```
query = QueryBuilder.table("users") \
    .where("age > ?", [21]) \
    .order_by("name") \
    .limit(10) \
    .to_mongo()
# Returns a MongoDB-compatible query dict

## v3.10.70 (2026-04-06)

- **New:** SSE (Server-Sent Events) support via `response.stream()` — pass a generator, framewor
- **New:** Chapter 24 added to documentation: Server-Sent Events
- Feature count: 45 (was 44)
- Full parity across Python, PHP, Ruby, Node.js
```

WebSocket backplane (v3.9.2). Redis pub/sub for scaling WebSocket broadcast across multiple server instances.

SameSite cookie default (v3.9.2). Session cookies default to `SameSite=Lax`. Override with the `TINA4_SESSION_SAMESITE` environment variable.

Breaking Changes

Session API rename (v3.9.0). The `unset()` method became `delete()` for cross-framework parity. `unset()` still works as an alias but will show a deprecation warning in future releases.

```
# Before

## v3.10.70 (2026-04-06)

- **New:** SSE (Server-Sent Events) support via `response.stream()` — pass a generator, framewor
- **New:** Chapter 24 added to documentation: Server-Sent Events
- Feature count: 45 (was 44)
- Full parity across Python, PHP, Ruby, Node.js

request.session.unset("cart")

# After

## v3.10.70 (2026-04-06)

- **New:** SSE (Server-Sent Events) support via `response.stream()` — pass a generator, framewor
- **New:** Chapter 24 added to documentation: Server-Sent Events
- Feature count: 45 (was 44)
- Full parity across Python, PHP, Ruby, Node.js

request.session.delete("cart")
```

Environment variable standardization (v3.9.1). All framework environment variables now follow the `TINA4_*` naming convention. `TOKEN_LIMIT` became `TINA4_TOKEN_LIMIT`. Check your `.env` file and rename any bare variables.

Queue backend change (v3.9.1). The SQLite queue backend no longer exists. If you used it, the file-based backend is a drop-in replacement. Queue data from the old SQLite store must be migrated manually.

---

v3.8.x — Pooling, Validation, and Security (March 25-26, 2026)

Features

Connection pooling (v3.8.1). Pass `pool=N` to the `Database` constructor for round-robin, thread-safe connection pooling.

```
from tina4_python import Database

db = Database("sqlite:///app.db", pool=4)
# Four connections rotate across requests

## v3.10.70 (2026-04-06)

- **New:** SSE (Server-Sent Events) support via `response.stream()` — pass a generator, framewor
- **New:** Chapter 24 added to documentation: Server-Sent Events
- Feature count: 45 (was 44)
- Full parity across Python, PHP, Ruby, Node.js
```

Validator class (v3.8.1). Input validation with an error response envelope.

```
from tina4_python import Validator

errors = Validator.validate(request.body, {
    "email": "required|email",
    "age": "required|integer|min:18"
})

if errors:
    return response({"errors": errors}, 422)
```

Upload size limit (v3.8.1). The `TINA4_MAX_UPLOAD_SIZE` environment variable caps file uploads. Set it in bytes.

TestClient for integration tests (v3.8.1). Test your routes without starting the server.

```
from tina4_python import TestClient

client = TestClient(app)
result = client.get("/api/users")
assert result.status_code == 200
```

SecurityHeadersMiddleware (v3.8.1). One import adds `X-Frame-Options`, `Strict-Transport-Security`, `Content-Security-Policy`, and `X-Content-Type-Options` to every response.

```
from tina4_python import SecurityHeadersMiddleware

app.use(SecurityHeadersMiddleware())
```

Zero core dependencies (v3.8.x). Database drivers, queue backends, and session handlers became optional installs. The core framework runs on Python's standard library alone.

---

v3.7.x — Template Auto-Serve and Firebird Fixes (March 25, 2026)

Features

Template auto-serve at `/` (v3.7.0). Place an `index.html` or `index.twig` in `src/templates/` and the framework serves it at the root path. User-registered `GET /` routes take priority.

```
src/
  templates/
    index.html   ← served at / with no route needed
```

Firebird idempotent migrations (v3.7.0). `ALTER TABLE ADD` statements on Firebird check `RDB$RELATION_FIELDS` before executing. Columns that already exist are skipped. Other databases and statement types are not affected.

---

v3.6.x — API Parity (March 24, 2026)

Breaking Changes

Auth method renames (v3.6.0). The authentication API aligned with the PHP, Ruby, and Node.js implementations.

```
# Before v3.6.0

## v3.10.70 (2026-04-06)

- **New:** SSE (Server-Sent Events) support via `response.stream()` — pass a generator, framewor
- **New:** Chapter 24 added to documentation: Server-Sent Events
- Feature count: 45 (was 44)
- Full parity across Python, PHP, Ruby, Node.js

from tina4_python import Auth

token = Auth.create_token(payload)       # old name
valid = Auth.validate_token(token)       # old name
```

```
# v3.6.0 and later

## v3.10.70 (2026-04-06)

- **New:** SSE (Server-Sent Events) support via `response.stream()` — pass a generator, framewor
- **New:** Chapter 24 added to documentation: Server-Sent Events
- Feature count: 45 (was 44)
- Full parity across Python, PHP, Ruby, Node.js

from tina4_python import Auth

token = Auth.get_token(payload)          # new primary name
valid = Auth.valid_token(token)          # new primary name
# create_token and validate_token still work as aliases
```

```
## v3.10.70 (2026-04-06)

- **New:** SSE (Server-Sent Events) support via `response.stream()` — pass a generator, framewor
- **New:** Chapter 24 added to documentation: Server-Sent Events
- Feature count: 45 (was 44)
- Full parity across Python, PHP, Ruby, Node.js
```

Pagination parameter rename (v3.6.0). `skip` became `offset` across all query methods.

```
# Before

## v3.10.70 (2026-04-06)

- **New:** SSE (Server-Sent Events) support via `response.stream()` — pass a generator, framewor
- **New:** Chapter 24 added to documentation: Server-Sent Events
- Feature count: 45 (was 44)
- Full parity across Python, PHP, Ruby, Node.js

users = User.find(skip=10, limit=5)

# After

## v3.10.70 (2026-04-06)

- **New:** SSE (Server-Sent Events) support via `response.stream()` — pass a generator, framewor
- **New:** Chapter 24 added to documentation: Server-Sent Events
- Feature count: 45 (was 44)
- Full parity across Python, PHP, Ruby, Node.js

users = User.find(offset=10, limit=5)
```

Token expiry parameter rename (v3.6.0). `token_expiry` became `expires_in` and now accepts minutes instead of seconds.

```
# Before

## v3.10.70 (2026-04-06)

- **New:** SSE (Server-Sent Events) support via `response.stream()` — pass a generator, framewor
- **New:** Chapter 24 added to documentation: Server-Sent Events
- Feature count: 45 (was 44)
- Full parity across Python, PHP, Ruby, Node.js

token = Auth.create_token(payload, token_expiry=3600)  # seconds

# After

## v3.10.70 (2026-04-06)

- **New:** SSE (Server-Sent Events) support via `response.stream()` — pass a generator, framewor
- **New:** Chapter 24 added to documentation: Server-Sent Events
- Feature count: 45 (was 44)
- Full parity across Python, PHP, Ruby, Node.js

token = Auth.get_token(payload, expires_in=60)          # minutes
```

Locale environment variable (v3.6.0). `LOCALE` became `TINA4_LOCALE`. Update your `.env` file.

---

v3.5.x — Bundled Frontend (March 24, 2026)

Features

tina4js bundled (v3.5.0). The reactive frontend library (13.6 KB minified) ships with the framework. No CDN link needed. Import it from your templates and build reactive UIs with signals, components, and declarative routing.

AutoCrud Swagger metadata (v3.5.0). Routes generated by `AutoCrud` now include Swagger annotations. They appear in the auto-generated API docs without extra configuration.

---

v3.3.x — WebSockets, File Uploads, and Frond Improvements (March 24, 2026)

Features

Route-based WebSocket handlers (v3.3.0). Define WebSocket endpoints with the same decorator pattern as HTTP routes.

```
@app.websocket("/ws/chat")
def chat_handler(message, client):
    # message is the incoming data
    # client.send() to reply
    client.send(f"Echo: {message}")
```

File upload improvements (v3.3.0). Uploaded files include raw bytes, a `data_uri` template filter, and consistent property names across all frameworks.

```
@app.post("/upload")
def handle_upload(request, response):
    file = request.files[0]
    print(file.filename)    # standardized from file_name
    print(file.type)        # new in v3.3.0
    raw = file.content      # raw bytes
    return response("Uploaded")
```

Lazy `column_info()` on DatabaseResult (v3.3.0). Query results expose schema metadata on demand. Call `result.column_info()` to inspect column names, types, and sizes without a separate query.

`@any` **decorator alias (v3.3.0).** `@any` works as shorthand for `@any_method`, matching the PHP, Ruby, and Node.js API.

Ternary-with-filter support in Frond (v3.3.0). Templates handle inline conditionals that pipe their result through a filter.

```
{{ user.name if user else "Anonymous"|upper }}
```

Breaking Changes

`job.data` **renamed to** `job.payload` **(v3.3.0).** Queue jobs use `payload` as the primary attribute. `job.data` remains as a read-only property alias but will be removed in a future release.

```
# Before
```

```
## v3.10.70 (2026-04-06)

- **New:** SSE (Server-Sent Events) support via `response.stream()` — pass a generator, framewor
- **New:** Chapter 24 added to documentation: Server-Sent Events
- Feature count: 45 (was 44)
- Full parity across Python, PHP, Ruby, Node.js

@app.consume("emails")
def send_email(job):
    to = job.data["to"]

# After
```

```
## v3.10.70 (2026-04-06)

- **New:** SSE (Server-Sent Events) support via `response.stream()` — pass a generator, framewor
- **New:** Chapter 24 added to documentation: Server-Sent Events
- Feature count: 45 (was 44)
- Full parity across Python, PHP, Ruby, Node.js

@app.consume("emails")
def send_email(job):
    to = job.payload["to"]
```

File upload property rename (v3.3.0). `file_name` became `filename` (no underscore). The old name is no longer available.

```
# Before

## v3.10.70 (2026-04-06)

- **New:** SSE (Server-Sent Events) support via `response.stream()` — pass a generator, framewor
- **New:** Chapter 24 added to documentation: Server-Sent Events
- Feature count: 45 (was 44)
- Full parity across Python, PHP, Ruby, Node.js

name = request.files[0].file_name

# After
```

```
## v3.10.70 (2026-04-06)

- **New:** SSE (Server-Sent Events) support via `response.stream()` — pass a generator, framewor
- **New:** Chapter 24 added to documentation: Server-Sent Events
- Feature count: 45 (was 44)
- Full parity across Python, PHP, Ruby, Node.js

name = request.files[0].filename
```

Route params merged into `request.params` (v3.3.0). Path parameters now merge into `request.params` alongside query parameters. If a query parameter shares a name with a path parameter, the path parameter wins.

---

v3.2.x — Flexible Route Handlers and DevReload (March 24, 2026)

Features

Flexible handler signatures (v3.2.0). Route handlers accept any combination of parameters. The framework inspects the signature and injects what you ask for.

```
# No parameters — fire and forget

## v3.10.70 (2026-04-06)

- **New:** SSE (Server-Sent Events) support via `response.stream()` — pass a generator, framewor
- **New:** Chapter 24 added to documentation: Server-Sent Events
- Feature count: 45 (was 44)
- Full parity across Python, PHP, Ruby, Node.js

@app.get("/ping")
def ping():
    return "pong"

# Response only

## v3.10.70 (2026-04-06)

- **New:** SSE (Server-Sent Events) support via `response.stream()` — pass a generator, framewor
- **New:** Chapter 24 added to documentation: Server-Sent Events
- Feature count: 45 (was 44)
- Full parity across Python, PHP, Ruby, Node.js

@app.get("/hello")
def hello(response):
    return response("Hello")

# Request only (type-hinted)

## v3.10.70 (2026-04-06)

- **New:** SSE (Server-Sent Events) support via `response.stream()` — pass a generator, framewor
- **New:** Chapter 24 added to documentation: Server-Sent Events
- Feature count: 45 (was 44)
- Full parity across Python, PHP, Ruby, Node.js

@app.post("/echo")
def echo(request: Request):
    return request.body

# Both

## v3.10.70 (2026-04-06)

- **New:** SSE (Server-Sent Events) support via `response.stream()` — pass a generator, framewor
- **New:** Chapter 24 added to documentation: Server-Sent Events
- Feature count: 45 (was 44)
- Full parity across Python, PHP, Ruby, Node.js

@app.get("/users")
def users(request, response):
```

```
    return response({"users": []})
```

DevReload with SCSS compilation (v3.2.0). The development server watches for file changes and reloads the browser. SCSS files compile on change.

Route groups (v3.2.0). Group routes under a shared prefix with shared middleware.

MongoDB queue backend (v3.2.0). Use MongoDB as a queue backend alongside the existing file-based, RabbitMQ, and Kafka options.

Migration naming convention (v3.2.0). Migration files follow the `YYYYMMDDHHMMSS` timestamp format. The `tina4 migrate --status` command shows which migrations have run.

Auto-increment port (v3.2.0). If the default port is in use, the framework picks the next available port and opens your browser.

Breaking Changes

Queue constructor simplified (v3.2.0). The `db` parameter was removed from the Queue constructor. The queue reads its backend from the environment.

```
# Before

## v3.10.70 (2026-04-06)

- **New:** SSE (Server-Sent Events) support via `response.stream()` — pass a generator, framewor
- **New:** Chapter 24 added to documentation: Server-Sent Events
- Feature count: 45 (was 44)
- Full parity across Python, PHP, Ruby, Node.js

from tina4_python import Queue, Database
db = Database("sqlite:///queue.db")
queue = Queue("emails", db=db)

# After

## v3.10.70 (2026-04-06)

- **New:** SSE (Server-Sent Events) support via `response.stream()` — pass a generator, framewor
- **New:** Chapter 24 added to documentation: Server-Sent Events
- Feature count: 45 (was 44)
- Full parity across Python, PHP, Ruby, Node.js

from tina4_python import Queue
queue = Queue("emails")
# Backend set via TINA4_QUEUE_BACKEND env var

## v3.10.70 (2026-04-06)

- **New:** SSE (Server-Sent Events) support via `response.stream()` — pass a generator, framewor
- **New:** Chapter 24 added to documentation: Server-Sent Events
- Feature count: 45 (was 44)
- Full parity across Python, PHP, Ruby, Node.js
```

Producer/Consumer classes removed (v3.2.0). Use `queue.produce()` and `queue.consume()` directly.

```
# Before
```

```
## v3.10.70 (2026-04-06)

- **New:** SSE (Server-Sent Events) support via `response.stream()` — pass a generator, framewor
- **New:** Chapter 24 added to documentation: Server-Sent Events
- Feature count: 45 (was 44)
- Full parity across Python, PHP, Ruby, Node.js

from tina4_python import Producer, Consumer
producer = Producer(queue)
producer.send({"to": "user@example.com"})

# After

## v3.10.70 (2026-04-06)

- **New:** SSE (Server-Sent Events) support via `response.stream()` — pass a generator, framewor
- **New:** Chapter 24 added to documentation: Server-Sent Events
- Feature count: 45 (was 44)
- Full parity across Python, PHP, Ruby, Node.js

queue.produce({"to": "user@example.com"})
```

---

v3.1.x — Benchmarks and Internal Improvements (March 22, 2026)

Features

Automated benchmark suite (v3.1.0). A reproducible benchmark compares Tina4 Python against 17 frameworks across four languages. Run it yourself with `python benchmarks/run.py`.

No user-facing API changes in this release. Internal improvements to test infrastructure and benchmark tooling.

---

v3.0.0 — The Rewrite (March 22, 2026)

The v3.0.0 release replaced the entire v2 codebase. Zero external dependencies. Pure Python standard library. 38 features. Over 6,000 tests.

What Changed

- **New module structure.** `tina4_python.core` replaces the old flat namespace.
- **Frond template engine.** Built-in Twig-compatible templates replace the Jinja2 dependency. Pre-compilation caches tokens for a 2.8x speedup on file renders.
- **Decorator-based routing.** `@app.get`, `@app.post`, `@app.put`, `@app.delete`, `@app.patch` replace the old `Route` class.
- **Built-in Dev Admin.** A browser-based dashboard shows routes, database tables, and queue status.

• **Error overlay in debug mode.** Stack traces render in the browser with source context, request details, and suggested fixes.

• **Swagger auto-registration.** Decorated routes appear in the Swagger UI without manual annotation.

• **Unified queue system.** Switch between file-based, RabbitMQ, and Kafka backends through environment variables. No code changes.

• **Database query caching.** Set `TINA4_DB_CACHE=true` for transparent caching with a 4x speedup on repeated queries.

• `tina4 generate` **scaffolding.** Generate models, routes, migrations, and middleware from the command line.

• **Custom error pages.** Self-contained 404, 403, and 500 pages with clean, framework-neutral design.

For the full migration guide, see Chapter 36: Upgrading from v2 to v3.

Release Candidate History

Five release candidates preceded the v3.0.0 stable release between March 21-22, 2026. They resolved test failures, polished the Dev Admin UI, added the benchmark suite, and stabilized the Frond template engine. If you tested a release candidate, upgrade to v3.0.0 or later. The RC builds are not supported.

Upgrading from v2 to v3

1. Overview

Tina4 v3 is a ground-up rewrite. Zero external dependencies. Pure Python stdlib. The framework does more with less.

The concepts are the same -- routing, ORM, templates, migrations, authentication. The APIs are cleaner. The internals are simpler. If you built something with v2, you will recognise everything in v3. But the import paths, decorators, and project layout have all changed.

This chapter covers every breaking change and the migration path for each one. Read it top to bottom before you start, then use the checklist at the end.

Step 0: Run the Automated Upgrade Command

Before doing anything manually, run the automated upgrade tool:

```
cd your-v2-project
tina4 i-want-to-stop-using-v2-and-switch-to-v3
```

This command automatically:

- Moves `routes/`, `orm/`, `templates/`, `scss/`, `public/`, `app/`, `locales/`, `seeds/` into `src/`
- Updates your `pyproject.toml` dependency from v2 to v3

After running the command, continue with the steps below for the changes that require manual attention.

---

2. Package and Installation

v2 installed with pip and pulled in external dependencies:

```
# v2
pip install tina4_python
```

v3 uses uv (recommended) or pip. Zero external dependencies:

```
# v3
uv add tina4-python

# or with pip
pip install tina4-python
```

v3 requires Python 3.12 or later. Check your version:

```
python --version
```

If you are on 3.11 or earlier, upgrade Python first. v3 uses features from 3.12 that cannot be backported.

---

3. Project Structure Changes

v2 used a flat structure. Routes, models, and templates could live anywhere. v3 expects a standard layout:

```
project/
  .env
  app.py
  src/
    routes/
      products.py
      users.py
    orm/
      product.py
      user.py
    templates/
      index.twig
      layout.twig
    app/
      helpers.py
      services.py
```

v3 auto-discovers every `.py` file in `src/` and its subdirectories. No manual imports. No registration. Drop a file in `src/routes/` and it loads at startup.

The framework adds your project root (CWD) to `sys.path`, so imports like this work everywhere:

```
from src.app.helpers import format_currency
from src.orm.product import Product
```

If you have v2 code scattered across the project root, move it into the appropriate `src/` subdirectory. The `tina4 init python .` command creates this structure for you.

---

4. Routing Changes

Decorators

v2 used `@route.get()`. v3 uses standalone decorators:

```
# v2
from tina4_python import route

@route.get("/products")
def list_products(request, response):
    return response.json({"products": []})

# v3
from tina4_python.core.router import get

@get("/products")
async def list_products(request, response):
    return response.json({"products": []})
```

Import all HTTP methods from one place:

```
from tina4_python.core.router import get, post, put, patch, delete
```

Auth Defaults

v3 changes the default auth behaviour. GET routes are public. POST, PUT, PATCH, and DELETE routes require authentication.

To make a write route public, use `@noauth()`:

```
from tina4_python.core.router import post, noauth

@noauth()
@post("/api/feedback")
async def submit_feedback(request, response):
    return response.json({"status": "received"}, 201)
```

To protect a GET route, use `@secured()`:

```
from tina4_python.core.router import get, secured

@secured()
@get("/admin/dashboard")
async def admin_dashboard(request, response):
    return response.render("admin/dashboard.twig")
```

Decorator Order

Order matters. Stack them outermost to innermost:

- `@noauth()` or `@secured()` (auth control)
- `@description("...")` (Swagger docs)
- `@get("/path")` or `@post("/path")` (route binding)

```
@noauth()
@description("Submit anonymous feedback")
@post("/api/feedback")
async def submit_feedback(request, response):
    return response.json({"status": "received"}, 201)
```

Middleware

Middleware uses a decorator. It works in any order relative to the route decorator:

```
from tina4_python.core.router import get, middleware
from src.app.rate_limiter import RateLimiter

@middleware(RateLimiter)
@get("/api/data")
async def get_data(request, response):
    return response.json({"data": []})
```

Wildcard Routes

Wildcard routes now work correctly in v3:

```
@get("/api/*")
async def catch_all(request, response):
    return response.json({"path": request.path})
```

---

5. Database Changes

Connection URL

v2 imported `Database` from the top-level package. v3 imports from `tina4_python.database`:

```
# v2
from tina4_python import Database
db = Database("sqlite3", "data/app.db")

# v3
from tina4_python.database import Database
db = Database("sqlite:///data/app.db")
```

URL format examples:

```
# SQLite
db = Database("sqlite:///data/app.db")

# PostgreSQL
db = Database("postgresql://localhost:5432/myapp", "user", "password")

# Firebird
db = Database("firebird://localhost:3050//var/data/app.fdb", "SYSDBA", "masterkey")
```

Keyword Arguments

v3 passes `**kwargs` through to the underlying driver. Useful for Firebird charset, connection timeouts, and other driver-specific options:

```
db = Database("firebird://localhost:3050//var/data/app.fdb", "SYSDBA", "masterkey", charset="ISO
```

Firebird Column Names

v3 lowercases all Firebird column names. This makes Firebird consistent with every other adapter. Update your code:

```
# v2
email = row["EMAIL"]
first_name = row["FIRST_NAME"]

# v3
email = row["email"]
first_name = row["first_name"]
```

Search your codebase for uppercase column access. Every instance needs updating.

Firebird Drivers

v3 supports both `firebird-driver` (modern, recommended) and `fdb` (legacy). It tries `firebird-driver` first, falls back to `fdb`. Install the one you prefer:

```
uv add firebird-driver    # recommended
# or
uv add fdb                # legacy fallback
```

Transactions

Use the database object's transaction methods. Never use raw SQL for transaction control:

```
# Correct
db.start_transaction()
db.execute("INSERT INTO products (name) VALUES (?)", ["Widget"])
db.commit()

# Wrong -- do not do this
db.execute("BEGIN")
db.execute("INSERT INTO products (name) VALUES (?)", ["Widget"])
db.execute("COMMIT")
```

Connection Pooling

v3 adds connection pooling. Pass the `pool` parameter:

```
db = Database("postgresql://localhost:5432/myapp", "user", "password", pool=4)
```

This creates a pool of 4 connections. The framework manages checkout and return.

---

6. ORM Changes

Field Definitions

v2 field definitions varied. v3 uses typed field classes:

```
# v2
class Product:
    table_name = "products"
    id = None
    name = None
    price = None

# v3
from tina4_python.orm import ORM, orm_bind, IntegerField, StringField, FloatField

class Product(ORM):
    table_name = "products"
    id = IntegerField(primary_key=True)
    name = StringField()
    price = FloatField()
```

Binding the Database

Call `orm_bind(db)` before using any ORM class. Do this once in `app.py`:

```
from tina4_python.database import Database
from tina4_python.orm import orm_bind

db = Database("sqlite:///data/app.db")
orm_bind(db)
```

Auto Mapping

Set `auto_map = True` on your ORM class for automatic snake_case to camelCase field mapping. This exists for cross-language parity (Python, PHP, Ruby, Node.js all share the same ORM concepts):

```
class UserProfile(ORM):
    table_name = "user_profiles"
    auto_map = True
    id = IntegerField(primary_key=True)
    first_name = StringField()
    last_name = StringField()
```

Field Mapping

For columns that do not follow conventions, use `field_mapping`:

```
class LegacyUser(ORM):
    table_name = "tbl_users"
    field_mapping = {
        "user_id": "usr_id",
        "email": "usr_email",
        "name": "usr_full_name"
    }
    user_id = IntegerField(primary_key=True)
    email = StringField()
    name = StringField()
```

Relationships

v3 adds `has_many`, `has_one`, and `belongs_to` with eager loading:

```
from tina4_python.orm import ORM, IntegerField, StringField, has_many, belongs_to

class Customer(ORM):
    table_name = "customers"
    id = IntegerField(primary_key=True)
    name = StringField()
    orders = has_many("Order", "customer_id")

class Order(ORM):
    table_name = "orders"
    id = IntegerField(primary_key=True)
    customer_id = IntegerField()
    total = FloatField()
    customer = belongs_to("Customer", "customer_id")
```

---

7. Template Engine Changes

Frond Replaces Template

v2 used a `Template` class. v3 uses the Frond engine, which is Jinja2/Twig-compatible:

```
# v2
return response.template("page.html", {"title": "Home"})

# v3
```

```
return response.render("page.twig", {"title": "Home"})
```

Frond is a singleton. It is created once at startup and reused for every request.

Custom Filters

Register filters in `app.py` before calling `run()`:

```
from tina4_python.frond import Frond

def money(value):
    return f"${value:,.2f}"

Frond.add_filter("money", money)
```

Use in templates:

```
{{ product.price | money }}
```

Custom Globals

Add global variables available in every template:

```
Frond.add_global("APP_NAME", "My Store")
Frond.add_global("YEAR", 2026)
```

```
<footer>&copy; {{ YEAR }} {{ APP_NAME }}</footer>
```

New Template Features

v3 Frond supports method calls on dict values:

```
{{ user.t("greeting") }}
```

Python slice syntax works:

```
{{ text[:10] }}
{{ items[1:3] }}
```

---

8. Migration Tracking Table

v2 used a `tina4_migration` table with a `description` column.

v3 uses an expanded schema:

Column	Type	Description
`migration_id`	text	Unique identifier
`description`	text	Migration description
`batch`	integer	Batch number
`executed_at`	timestamp	When it ran
`passed`	boolean	Whether it succeeded

You do not need to alter the table yourself. Run `tina4 migrate` and v3 auto-detects the v2 schema. It adds the missing columns and backfills `migration_id` from `description`. Your existing migration history is preserved.

```
tina4 migrate
```

That is it. No manual SQL. No data loss.

---

9. Authentication Changes

v2 had various auth approaches. v3 consolidates into an `Auth` class:

```
from tina4_python.auth import Auth

# Generate a token
token = Auth.get_token({"user_id": 42, "role": "admin"})

# Validate a token
is_valid = Auth.valid_token(token)

# Extract the payload
payload = Auth.get_payload(token)
```

Password hashing:

```
hashed = Auth.hash_password("my-secret-password")
matches = Auth.check_password("my-secret-password", hashed)  # True
# Signature: check_password(password, hashed) -- plain text first, hash second
```

JWT uses HMAC-SHA256. Set the signing key in `.env`:

```
TINA4_SECRET=your-long-random-secret-key
```

Token lifetime defaults to 60 minutes. Override with:

```
TINA4_TOKEN_LIMIT=120
```

---

10. Session Changes

v2 had basic file-based sessions. v3 supports pluggable backends.

Set the backend in `.env`:

```
TINA4_SESSION_BACKEND=file
```

Available backends:

Value	Backend	Package Required
`file`	Local filesystem (default)	None
`redis`	Redis	`redis`
`valkey`	Valkey	`valkey`

`mongodb`	MongoDB	`pymongo`
`database`	Database table	None

Session cookies default to `SameSite=Lax`. Override with:

```
TINA4_SESSION_SAMESITE=Strict
```

---

11. New Features in v3

v3 adds capabilities that did not exist in v2. Each has its own chapter:

- **Events system** -- publish and subscribe to application events (Chapter 13)
- **GraphQL engine** -- schema-first GraphQL with resolvers (Chapter 22)
- **WSDL/SOAP services** -- consume and expose SOAP endpoints
- **WebSocket with Redis backplane** -- real-time with horizontal scaling (Chapter 23)
- **Response caching middleware** -- cache responses with TTL and invalidation (Chapter 11)
- **DI container** -- dependency injection for services and repositories
- **Queue system** -- RabbitMQ, Kafka, and MongoDB backends (Chapter 12)
- **Swagger/OpenAPI auto-generation** -- live API docs from route decorators (Chapter 20)
- **Auto-CRUD endpoint generator** -- CRUD routes from ORM models with one line
- **Seeder/fake data** -- populate databases with realistic test data
- **i18n translations** -- multi-language support with translation files
- **AI coding assistant context** -- `tina4 ai` generates context for LLM coding tools
- **Error overlay in dev mode** -- stack traces rendered in the browser
- **SCSS auto-compilation** -- `.scss` files compiled to CSS on change

You do not need to adopt all of these at once. They are opt-in. Migrate your existing code first, then add new features as you need them.

Common Pitfalls

1. POST/PUT/DELETE routes now require authentication

This is the most common upgrade issue. In v2, all routes were public by default. In v3, only GET routes are public -- POST, PUT, PATCH, and DELETE require a Bearer JWT token.

Symptom: Working v2 endpoints return `401 Unauthorized` after upgrading.

Fix: Add `@noauth` to any write route that should remain public.

Find affected routes:

```
grep -rn "@post\|@put\|@patch\|@delete" src/routes/
```

Review each match -- if the endpoint should be public (webhooks, public forms, etc.), add the `@noauth()` decorator.

2. Database connection strings changed

v2 used driver-specific classes. v3 uses URL format:

```
# v2
db = DatabaseSqlite3("data/app.db")
# v3
db = Database("sqlite:///data/app.db")
```

3. Template engine renamed

v2: `Template.render()` -- v3: `Frond.render()` (or `response.render()`)

The Twig syntax is the same -- your `.twig` files work unchanged. Only the Python API call changes.

---

12. Step-by-Step Migration Checklist

Follow this order. Each step builds on the previous one.

- **Install v3**

``bash uv add tina4-python ``

- **Create the v3 project structure**

``bash tina4 init python . ` This creates` src/routes/`,` src/orm/`,` src/templates/`, and` src/app/` if they do not exist. It will not overwrite existing files.

- **Move route files to** `src/routes/`

Update imports from `from tina4_python import route` to `from tina4_python.core.router import get, post, put, patch, delete`. Replace `@route.get()` with `@get()`. Make handler functions `async`.

- **Move ORM models to** `src/orm/`

Replace raw field definitions with typed fields: `IntegerField`, `StringField`, `FloatField`, `TextField`. Add `orm_bind(db)` in `app.py`.

- **Move templates to** `src/templates/`

Replace `response.template()` calls with `response.render()`. Templates are Twig-compatible -- most existing templates work without changes.

- **Update** `app.py`

``python from tina4_python.core import run run() ``

- **Update database connections**

Switch to URL format in `.env`: ``~~bash TINA4_DATABASE_URL=~~sqlite:///data/app.db ``

- **Run migrations**

``bash tina4 migrate `` The tracking table upgrades automatically.

- **Fix Firebird column names**

Search for uppercase column access (`row["UPPER"]`) and change to lowercase (`row["upper"]`). Only applies if you use Firebird.

- **Start the server and test**

``bash tina4 serve `` Hit every route. Check the logs for errors.

- **Run the doctor**

``bash tina4 doctor `` This verifies your project structure, database connection, and configuration.

That covers the migration. Most projects take under an hour. The biggest time sink is updating import paths and decorators -- a find-and-replace handles the bulk of it. Once you are running on v3, you get zero dependencies, faster startup, and access to every new feature listed in section 11.

Complete Feature List

1. What Tina4 Covers

Tina4 is four implementations of the same framework: Python, PHP, Ruby, and Node.js. Every feature listed here ships in all four. The same behaviour. The same API shape. The same test expectations.

9,311 tests across 280 files verify parity. A feature that works in Python works identically in PHP, Ruby, and Node.js.

---

2. Foundations

#	Feature	Description
01	**Router**	Decorator-based routing with `@get`, `@post`, `@put`, `@patch`, `@delete`
02	**Request / Response**	Typed request object (params, body, headers, session) and fluent response builder
03	**Middleware**	Per-route and global middleware pipeline with named middleware registry
04	**Static Files**	Automatic serving of `src/public/` with configurable root
05	**Environment Config**	`.env` loading via `TINA4_*` prefix with type coercion and validation
06	**Dev Dashboard**	Built-in `/__dev` panel showing routes, logs, cache, queue, and request trace
07	**Hot Reload**	File watcher restarts the server on source change during development

---

3. Templates

#	Feature	Description
08	**Frond Templates**	Tina4's own template engine: `{{ var }}`, `{% if %}`, `{% for %}`, partials, layouts
09	**Template Caching**	Compiled templates cached in memory; TTL configurable via `TINA4_TEMPLATE_CACHE_TTL`
10	**Asset Helpers**	`css()`, `js()`, `image()` helpers with cache-busting query strings

---

4. Database

#	Feature	Description
11	**Database**	Raw SQL execution: `execute`, `fetch_one`, `fetch_all`, parameterised queries
12	**Multi-Database**	Multiple simultaneous connections; switch via `Database.use("secondary")`
13	**ORM**	Active record pattern: `save()`, `find()`, `find_all()`, `delete()`, soft deletes
14	**Query Builder**	Fluent query API: `select`, `where`, `join`, `order_by`, `limit`, `paginate`
15	**Migrations**	Version-controlled schema changes via `migrate up`, `migrate down`, auto-runner
16	**Transactions**	`db.begin()`, `db.commit()`, `db.rollback()` with context manager support
17	**Sequence / Next ID**	`Database.get_next_id(table)` — race-safe sequence via a dedicated sequence table

---

5. Authentication and Security

#	Feature	Description
18	**Authentication**	JWT-based auth middleware; `@require_auth` decorator; token decode helpers
19	**Sessions**	Server-side sessions with cookie binding; memory, file, and Redis backends
20	**CSRF Protection**	Token generation, form injection, and validation middleware
21	**Rate Limiting**	Per-IP and per-route request limits via `RateLimit:n` middleware
22	**CORS**	Configurable origin, method, and header policy via `TINA4_CORS_*` env vars
23	**Security Headers**	Helmet-style middleware: CSP, X-Frame-Options, HSTS, Referrer-Policy

---

6. Performance

#	Feature	Description
24	**Response Cache**	HTTP-level response caching via `ResponseCache:ttl` middleware
25	**Cache API**	`cache_get`, `cache_set`, `cache_delete`; memory, Redis, and file backends
26	**DB Query Cache**	Automatic caching of identical database queries; `TINA4_DB_CACHE=true`
27	**Compression**	Gzip/Brotli response compression via `Compress` middleware

---

7. Background Processing

#	Feature	Description
28	**Queues**	Push/pop/consume pattern; file, RabbitMQ, Kafka, and MongoDB backends
29	**Job Lifecycle**	Pending → Reserved → Completed/Failed → Dead Letter; retry with `retry_failed()`
30	**Service Runner**	Long-running background services with start/stop lifecycle and thread management

---

8. Events and Messaging

#	Feature	Description
31	**Events**	`emit` / `on` / `once` / `off`; priority ordering; async via `emit_async`
32	**Email (Messenger)**	SMTP email with HTML bodies, attachments, cc/bcc; dev interception mode
33	**WebSocket**	`@ws` route decorator; broadcast, rooms, ping/pong; `tina4-js` client library
34	**SSE / Streaming**	`response.stream()` for Server-Sent Events; async generator, chunked transfer encoding, keep-alive

---

9. APIs

#	Feature	Description
35	**Swagger / OpenAPI**	Auto-generated OpenAPI 3.0 spec and Swagger UI at `/swagger`
36	**GraphQL**	Schema-first GraphQL with resolvers via `@graphql_query` and `@graphql_mutation`
37	**API Client**	Outbound HTTP: `Api.get/post/put/patch/delete`; auth headers, SSL, timeout
38	**WSDL / SOAP**	SOAP 1.1 service with `@wsdl_operation`; auto WSDL at `?wsdl`

---

10. Tooling and Developer Experience

#	Feature	Description
39	**Scaffolding**	`tina4 init`, `tina4 generate route`, `tina4 generate model`, `tina4 generate migration`
40	**Testing**	Built-in test client with `client.get/post/put/delete`; fixture helpers
41	**Localization**	JSON locale files; `t()` function; `{placeholder}` interpolation; fallback chain
42	**Structured Logging**	`Log.info/debug/warning/error`; log levels; file output with rotation
43	**DI Container**	`Container.register` (transient) and `Container.singleton` (cached)
44	**Frontend (tina4-js)**	1.5KB core gzipped reactive JS: signals, templating, routing, WebSocket client
45	**MCP Dev Tools**	Model Context Protocol integration: route inspection, ORM access, log streaming

---

11. Cross-Language Parity

Every feature in the table above ships in all four Tina4 implementations:

Implementation	Language	Package
tina4-python	Python 3.12+	`pip install tina4-python`
tina4-php	PHP 8.2+	`composer require tina4/tina4-php`
tina4-ruby	Ruby 3.3+	`gem install tina4-ruby`

tina4-nodejs	Node.js 20+	`npm install tina4-nodejs`

What parity means in practice:

- A route defined in Python with `@get("/api/users")` works identically in PHP as `#[Get("/api/users")]`, in Ruby as `get "/api/users"`, and in Node as `router.get("/api/users")`.
- A Frond template written once renders correctly when served by any of the four implementations.
- A test written against the Python test client can be ported line-for-line to the PHP, Ruby, or Node equivalent with only syntax changes.
- `TINA4_*` environment variables are honoured by all four implementations with the same semantics.

---

12. Test Coverage Summary

Metric	Value
Total tests	9,311
Test files	280
Cross-language parity tests	45 feature × 4 implementations
Minimum coverage per feature	100% (all four languages)

Parity is enforced by the CI pipeline. A feature is not considered shipped until tests pass in all four languages. A bug fix must be applied to all four before it is closed.

---

13. What Is Not in Tina4

Tina4 is deliberately minimal:

- **No ORM magic / active record inflation**: Queries are explicit; the ORM maps rows to objects, it does not generate SQL automatically from relationships.
- **No dependency scanning**: The DI container is explicit. You register what you want, not what a scanner finds.
- **No bundled frontend framework**: `tina4-js` is a separate library (1.5KB core, under 6KB full) you opt into. Tina4 itself is backend-only.
- **No cloud vendor lock-in**: Queues, cache, sessions, and email are backend-agnostic. The backend is a config variable.

The goal is a framework you can understand completely in a weekend and build on confidently for years.