Tina4 for Ruby 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 Ruby 56

1. What Is Tina4 Ruby . . . 56

2. Prerequisites and Installation . . . 56

What You Need . . . 56

Installing the Tina4 CLI . . . 57

Creating a New Project . . . 57

Starting the Dev Server . . . 58

3. Project Structure Walkthrough . . . 58

4. Your First Route . . . 59

Test It . . . 60

Understanding What Happened . . . 60

Adding More HTTP Methods . . . 60

5. Your First Template . . . 61

Create a Base Layout . . . 61

Create a Product Listing Page . . . 62

Create the Route That Renders the Template . . . 62

See It in the Browser . . . 63

How Template Rendering Works . . . 63

About tina4css . . . 64

6. Understanding .env . . . 64

7. The Dev Dashboard . . . 65

8. Manual Setup (No CLI) . . . 66

Step 1: Create Gemfile . . . 66

Step 2: Create app.rb . . . 66

Step 3: Create the Folder Structure . . . 66
Step 4: Create .env . . . 67
Step 5: Run It . . . 67
9. Request & Response Fundamentals . . . 67
Reading Query Parameters . . . 67
Reading URL Path Parameters . . . 67
Reading the Request Body . . . 68
Reading Headers . . . 68
Sending JSON Responses . . . 68
Sending HTML / Template Responses . . . 68
Status Codes . . . 68
Worked Example: A Complete Route File . . . 69
10. Exercise: Greeting API + Product List Template . . . 70
Exercise Part A: Greeting API . . . 70
Exercise Part B: Product List Page . . . 71
11. Solutions . . . 71
Solution A: Greeting API . . . 71
Solution B: Product List Page . . . 72
12. Gotchas . . . 74
1. File not auto-discovered . . . 74
2. "Uninitialized constant" errors . . . 74
3. JSON response shows HTML . . . 74
4. Template not found . . . 74
5. Port already in use . . . 74
6. Changes not reflected . . . 75
7. .env not loaded . . . 75
8. Debug mode in production . . . 75

Routing 76

1. How Routing Works in Tina4 . . . 76
2. HTTP Methods . . . 76
3. Path Parameters . . . 77
    Typed Parameters . . . 77
    Typed Parameters in Action . . . 78
4. Query Parameters . . . 79
5. Route Groups . . . 79
6. Middleware . . . 80
    Middleware on a Single Route . . . 80
    Blocking Middleware . . . 81
    Middleware on a Group . . . 81
    Multiple Middleware . . . 82
7. Route Decorators: @noauth and @secured . . . 82
    @noauth -- Public Routes . . . 82
    @secured -- Protected GET Routes . . . 82
8. Route Chaining: .secure and .cache . . . 82
    .secure . . . 83
    .cache . . . 83
    Chaining Both . . . 83
9. Wildcard and Catch-All Routes . . . 83
    Wildcard Routes . . . 83
    Catch-All Route (Custom 404) . . . 84
10. Route Listing via CLI . . . 84
11. Organizing Route Files . . . 85
    Pattern 1: One File Per Resource . . . 85
    Pattern 2: Subdirectories by Feature . . . 85

12. Exercise: Build a Full CRUD API for Products . . . 85
Requirements . . . 85
13. Solution . . . 87
14. Gotchas . . . 89
1. Trailing Slashes Matter . . . 89
2. Parameter Names Must Be Unique in a Path . . . 89
3. Method Conflicts . . . 89
4. Route Handler Must Return a Response . . . 89
5. Block Syntax Matters . . . 89
6. Middleware Must Be a Named Function or String . . . 90
7. Group Prefix Must Start with a Slash . . . 90
Request & Response **91**
1. The Two Objects You Use Everywhere . . . 91
2. The Request Object . . . 91
method . . . 91
path . . . 91
params . . . 91
query . . . 92
body . . . 92
headers . . . 92
ip . . . 92
cookies . . . 92
files . . . 92
Inspecting the Full Request . . . 92
3. The Response Object . . . 93
json -- JSON Response . . . 93
html -- Raw HTML Response . . . 93

text -- Plain Text Response . . . 94
render -- Template Response . . . 94
redirect -- Redirect Response . . . 94
file -- File Download Response . . . 94
xml -- XML Response . . . 94
error -- Error Envelope . . . 94
4. Status Codes . . . 95
5. Custom Headers . . . 96
CORS Headers . . . 96
6. Cookies . . . 96
7. File Uploads . . . 97
Handling a Single File Upload . . . 98
Saving the Uploaded File . . . 98
Handling Multiple Files . . . 99
8. File Downloads . . . 99
9. Content Negotiation . . . 100
10. Input Validation . . . 101
The Validator Class . . . 101
The Error Response Envelope . . . 101
Upload Size Limits . . . 101
11. Exercise: Build an Image Upload API . . . 102
Requirements . . . 102
Test with: . . . 102
12. Solution . . . 102
13. Gotchas . . . 104
1. Forgetting the response method . . . 104
2. request.body is nil for GET ~~requests~~ . . . 105

3. Body is nil for JSON requests . . . 105
4. Content-Type Mismatch . . . 105
5. File Uploads Return Empty . . . 105
6. Redirect Loops . . . 105
7. Cookie Not Set . . . 105
8. Large Request Body Rejected . . . 106
9. Chaining Response Methods in Wrong Order . . . 106

Templates **107**

1. Beyond JSON -- Rendering HTML . . . 107
2. Variables and Expressions . . . 107
Accessing Nested Data . . . 107
Method Calls on Values . . . 108
Expressions . . . 108
3. Template Inheritance . . . 108
Base Layout . . . 108
Child Template . . . 109
Using {{ parent() }} . . . 109
4. Includes . . . 110
Passing Variables to Includes . . . 111
5. For Loops . . . 111
The loop Variable . . . 111
Empty Lists . . . 112
Looping Over Key-Value Pairs . . . 112
6. Conditionals . . . 112
if / elseif / else . . . 112
Ternary Operator . . . 112
Testing for Existence . . . 112

Truthiness . . . 112
{% set %} -- Local Variables . . . 113
7. Filters . . . 113
Complete Filter Reference . . . 114
String Filters . . . 114
Array Filters . . . 115
Encoding Filters . . . 116
Numeric Filters . . . 116
JSON Filters . . . 116
Dict Filters . . . 117
Other Filters . . . 117
Chaining Filters . . . 117
Dumping Values for Debugging . . . 117
8. Macros . . . 118
Defining a Macro . . . 118
Using Macros . . . 118
9. Special Tags . . . 119
{% raw %} -- Literal Output . . . 119
{% spaceless %} -- Remove Whitespace . . . 119
{% autoescape %} -- Control Escaping . . . 120
Comments . . . 120
10. tina4css Integration . . . 120
Layout Classes . . . 120
Common Components . . . 121
Utility Classes . . . 121
11. Exercise: Build a Product Catalog Page . . . 121
Requirements . . . 121

Data . . . 122
Test with: . . . 122
12. Solution . . . 122
13. Gotchas . . . 125
1. {% extends %} Must Be the First Tag . . . 125
2. Undefined Variables Show Nothing . . . 125
3. Auto-Escaping Prevents HTML Output . . . 125
4. Variable Scope in Includes . . . 125
5. Macro Arguments Are Positional . . . 125
6. Template File Extension Does Not Matter . . . 126
7. Filters Are Not Ruby Methods . . . 126
Database **127**
1. From Arrays to Real Data . . . 127
2. Connecting to a Database . . . 127
The Default: SQLite . . . 127
Connection Strings for Other Databases . . . 127
Firebird URL Forms . . . 127
Separate Credentials . . . 128
Programmatic Connection . . . 128
Connection Pooling . . . 128
Verifying the Connection . . . 128
3. Getting the Database Object . . . 129
4. Raw Queries . . . 129
fetch -- Get Multiple Rows . . . 129
DatabaseResult . . . 129
Properties . . . 130
Iteration . . . 130

Index Access . . . 130
Countable . . . 130
Conversion Methods . . . 130
Schema Metadata with column_info . . . 130
fetch_one -- Get a Single Row . . . 131
execute -- Run a Statement . . . 131
Full Example: A Simple Query Route . . . 131
5. Parameterised Queries . . . 131
A Safe Search Endpoint . . . 132
6. Transactions . . . 132
7. Schema Inspection . . . 133
tables . . . 133
columns . . . 133
table_exists? . . . 134
driver_name . . . 134
A Schema Info Endpoint . . . 134
8. Batch Operations with execute_many . . . 134
9. Helper Methods: insert, update, delete . . . 135
insert . . . 135
update . . . 135
delete . . . 135
10. Migrations . . . 135
File Naming . . . 135
Generating a Migration . . . 136
SQL Migrations . . . 136
Ruby Class Migrations . . . 136
Running Migrations . . . 137

Checking Migration Status . . . 137
Rolling Back . . . 137
A Real Migration Sequence . . . 137
Migration Tracking . . . 138
11. Query Caching . . . 138
12. Exercise: Build a Notes App . . . 138
Requirements . . . 138
Test with: . . . 139
13. Solution . . . 140
Migration . . . 140
Routes . . . 140
14. Seeder -- Generating Test Data . . . 143
FakeData . . . 143
Deterministic Output . . . 143
Seeding with ORM . . . 144
Overrides . . . 144
Seeding a Raw Table . . . 144
Batch Seeding . . . 144
Seed Files . . . 144
When to Use It . . . 145
15. Gotchas . . . 145
1. Forgetting the transaction block . . . 145
2. Connection String Formats . . . 145
3. SQLite File Paths . . . 146
4. Parameterised Queries with LIKE . . . 146
5. Boolean Values in SQLite . . . 146
6. Migration Already Applied . . . 146

7. fetch Returns Empty Array, Not Nil . . . 146

8. SQL Injection Through String Interpolation . . . 146

ORM **148**

1. From SQL to Objects . . . 148

ORM at a Glance: Four Languages, One Shape . . . 148

Defining a Model . . . 148

Common Query Operations . . . 149

2. Defining a Model . . . 150

Field Types . . . 151

Field Options . . . 152

Field Mapping . . . 152

find() vs where() -- naming convention . . . 153

auto_map and Case Conversion Utilities . . . 153

3. create_table -- Schema from Models . . . 153

4. CRUD Operations . . . 154

save -- Create or Update . . . 154

create -- Build and Save in One Step . . . 154

findbyid -- Fetch One Record by Primary Key . . . 154

find -- Query by Filter Hash . . . 155

where -- Query with SQL Conditions . . . 155

delete -- Remove a Record . . . 155

Listing Records . . . 155

select_one -- Fetch a Single Record by SQL . . . 156

load -- Populate an Existing Instance . . . 156

count -- Count Records . . . 156

5. toh, todict, to_json, and Other Serialisation . . . 156

toh and todict . . . 156

to_json . . . 157
Other Serialisation Methods . . . 157
6. Relationships . . . 157
foreignkeyfield — Auto-Wired Relationships . . . 157
Declarative Relationships . . . 158
has_many . . . 158
has_one . . . 159
belongs_to . . . 159
Imperative Relationships . . . 159
7. Eager Loading . . . 160
Nested Eager Loading . . . 160
to_h with Nested Includes . . . 160
8. Soft Delete . . . 161
Deleting and Restoring . . . 161
Including Soft-Deleted Records . . . 161
Counting with Soft Delete . . . 162
Custom Soft Delete Field . . . 162
When to Use Soft Delete . . . 162
9. Auto-CRUD . . . 162
The auto_crud Flag . . . 162
Manual Registration . . . 162
What the Generated Routes Do . . . 163
Custom Routes Alongside Auto-CRUD . . . 164
10. Scopes . . . 164
Class Methods as Scopes . . . 164
Dynamic Scopes with scope() . . . 164
11. Input Validation . . . 165

12. Exercise: Build a Blog with Relationships . . . 165
Requirements . . . 165
13. Solution . . . 166
14. Gotchas . . . 169
1. Forgetting to call save . . . 169
2. findbyid returns nil . . . 169
3. find takes a hash, not keyword arguments . . . 169
4. to_h includes everything . . . 169
5. Validation runs on save, but check errors . . . 169
6. Foreign key not enforced . . . 170
7. N+1 query problem . . . 170
8. Auto-CRUD endpoint conflicts . . . 170
9. Soft-deleted records appearing in queries . . . 170
10. Table name pluralisation . . . 170
15. Raw SQL . . . 171
16. QueryBuilder Integration . . . 171
QueryBuilder **173**
1. Why a Query Builder? . . . 173
2. Creating a QueryBuilder . . . 173
3. Selecting Columns . . . 173
4. Filtering with where . . . 173
Multiple where Calls . . . 174
5. OR Conditions with or_where . . . 174
6. Joins . . . 175
Inner Join . . . 175
Left Join . . . 175
Multiple Joins . . . 175

7. Ordering . . . 175
8. Limiting and Offsetting . . . 176
9. Grouping and Having . . . 176
Group By . . . 176
Having . . . 176
10. Executing Queries . . . 177
get -- Multiple Rows . . . 177
first -- Single Row . . . 177
count -- Row Count . . . 177
exists? -- Boolean Check . . . 177
11. Inspecting the SQL with to_sql . . . 177
12. Using QueryBuilder in Routes . . . 178
A Filtered API Endpoint . . . 178
A Dashboard Summary Endpoint . . . 179
13. Exercise: Build a Product Search API . . . 179
Requirements . . . 179
Test with: . . . 180
14. Solution . . . 180
Search Endpoint . . . 180
Stats Endpoint . . . 181
15. NoSQL: MongoDB Queries . . . 182
Operator Mapping . . . 182
Example . . . 182
16. Gotchas . . . 183
1. Default Limit of 100 . . . 183
2. Parameter Order Matters . . . 183
3. No Database Connection . . . 183

4. count Replaces Your Columns . . . 183
5. to_sql Does Not Show Parameter Values . . . 183
6. or_where as First Condition . . . 184

Authentication **185**

1. Locking the Door . . . 185
2. JWT Tokens . . . 185
Generating a Token . . . 185
Token Expiry . . . 185
Validating a Token . . . 186
Reading the Payload . . . 186
The Secret Key and Algorithm . . . 186
3. Password Hashing . . . 187
Hashing a Password . . . 187
Checking a Password . . . 187
Registration Example . . . 187
4. The Login Flow . . . 188
5. Using Tokens in Requests . . . 189
6. Protecting Routes . . . 189
Auth Middleware . . . 189
Applying Middleware to Routes . . . 190
Applying Middleware to a Group . . . 190
7. @noauth and @secured Decorators . . . 190
@noauth -- Skip Authentication . . . 190
@secured -- Require Authentication for GET Routes . . . 191
Role-Based Authorization . . . 191
8. CSRF Protection . . . 191
Generating a Token . . . 191

Validating the Token . . . 192
9. Sessions . . . 192
Using Sessions . . . 192
10. Exercise: Build Login, Register, and Profile . . . 193
Requirements . . . 193
Test with: . . . 194
11. Solution . . . 194
Migration . . . 194
Middleware . . . 194
Routes . . . 195
12. Gotchas . . . 198
1. Token Expiry Confusion . . . 198
2. Secret Key Management . . . 198
3. CORS with Authentication . . . 198
4. Storing Tokens in localStorage . . . 199
5. Forgetting @noauth on Login . . . 199
6. Password Hash Column Too Short . . . 199
7. Token in URL Query Parameters . . . 199
Sessions & Cookies **200**
1. State in a Stateless World . . . 200
2. How Sessions Work . . . 200
3. The Session API . . . 200
Full API Reference . . . 201
4. File Sessions (Default) . . . 201
5. Redis Sessions . . . 202
6. MongoDB Sessions . . . 202

7. Valkey Sessions . . . 202
8. Database Sessions . . . 202
9. Reading and Writing Session Data . . . 202
Storing Complex Data . . . 203
10. Flash Messages . . . 204
Setting a Flash Message . . . 204
Reading and Clearing Flash Messages . . . 204
Using Flash Messages in Templates . . . 204
11. Setting and Reading Cookies . . . 205
Setting a Cookie . . . 205
Reading a Cookie . . . 205
Deleting a Cookie . . . 205
When to Use Cookies vs Sessions . . . 205
12. Remember Me Functionality . . . 206
13. Session Security . . . 206
Configuration Options . . . 206
Session Regeneration . . . 207
Destroy a Session . . . 207
14. Exercise: Build a Shopping Cart with Session Storage . . . 207
Requirements . . . 208
Business Rules . . . 208
Test with: . . . 208
15. Solution . . . 209
16. Gotchas . . . 211
1. Sessions Do Not Work with curl Without Cookie Flags . . . 211
2. Session Data Disappears After Server Restart . . . 211
3. Session Cookie Not Sent Over HTTP in Production . . . 211

4. Flash Messages Show Twice . . . 211
5. Large Session Data Causes Slow Requests . . . 211
6. Remember Me Token Not Invalidated on Password Change . . . 211
7. Session Fixation . . . 212

Middleware **213**

1. The Gatekeepers . . . 213
2. What Middleware Is . . . 213
3. Built-in CorsMiddleware . . . 214
4. Built-in RateLimiter . . . 214
Custom Limits Per Group . . . 215
Built-in RequestLoggerMiddleware . . . 215
Built-in SecurityHeadersMiddleware . . . 215
Combining All Four Built-In Middleware . . . 216
5. Writing Custom Middleware . . . 216
Request Logging Middleware . . . 216
Request Timing Middleware . . . 217
IP Whitelist Middleware . . . 217
Request Validation Middleware . . . 217
Writing Class-Based Middleware . . . 217
JWT Authentication Middleware (Class-Based) . . . 218
6. Applying Middleware to Individual Routes . . . 219
7. Route Groups with Shared Middleware . . . 219
8. Middleware Execution Order . . . 220
9. Short-Circuiting . . . 220
Maintenance Mode . . . 221
10. Modifying Requests in Middleware . . . 221
11. Real-World Middleware Stack . . . 221

12. Exercise: Build an API Key Middleware . . . 222
Setup . . . 222
Test with: . . . 223
13. Solution . . . 223
14. Gotchas . . . 224
1. Middleware Must Be a Named Method . . . 224
2. Forgetting to Return next_handler Result . . . 224
3. Middleware Order Matters . . . 224
4. CORS Preflight Returns 404 . . . 224
5. Rate Limiter Counts Preflight Requests . . . 225
6. Middleware File Not Auto-Loaded . . . 225
7. Short-Circuiting Skips Cleanup Middleware . . . 225
Security 225
1. Secure-by-Default Routing . . . 226
Making a Write Route Public . . . 226
Protecting a GET Route . . . 226
The Rule . . . 226
2. CSRF Protection . . . 227
How It Works . . . 227
The Template . . . 227
The Middleware . . . 227
AJAX Requests . . . 227
Tokens in Query Strings — Blocked . . . 228
Skipping CSRF Validation . . . 228
Disabling CSRF Globally . . . 228
3. Session-Bound Tokens . . . 228
How Stolen Tokens Fail . . . 228

4. Security Headers . . . 229
HSTS — Enforcing HTTPS . . . 229
Customising Headers . . . 229
5. SameSite Cookies . . . 230
6. Login Flow — Complete Example . . . 230
The Login Route . . . 231
The Login Form . . . 232
Protected Pages — Checking the Session . . . 232
Logout — Destroying the Session . . . 232
7. Handling Expired Sessions . . . 233
The Pattern: Redirect to Login, Then Back . . . 233
Token Refresh . . . 233
8. Rate Limiting . . . 234
9. CORS and Credentials . . . 234
10. Security Checklist . . . 235
Gotchas . . . 235
1. "My POST route returns 401 but I didn't add auth" . . . 235
2. "CSRF validation fails on AJAX requests" . . . 235
3. "I disabled CSRF but forms still fail" . . . 236
4. "My Content-Security-Policy blocks inline scripts" . . . 236
5. "User stays logged in after session expires" . . . 236
Exercise: Secure Contact Form . . . 236
Solution . . . 237
Caching **239**
1. From 800ms to 3ms . . . 239
2. Response Caching with ResponseCache Middleware . . . 239

Cache Headers . . . 240
Caching with Query Parameters . . . 240
What Not to Cache . . . 240
3. Memory Cache (Default) . . . 240
4. Redis Cache . . . 240
Why Redis? . . . 241
5. File Cache . . . 241
When to Use File Cache . . . 241
6. Direct Cache API . . . 241
cache_set . . . 241
cache_get . . . 242
cache_delete . . . 242
Real-World Pattern: Cache-Aside . . . 242
7. Database Query Caching . . . 243
When to Use DB Cache . . . 243
When Not to Use DB Cache . . . 243
8. Cache Invalidation Strategies . . . 244
Strategy 1: Time-Based Expiry (TTL) . . . 244
Strategy 2: Event-Based Invalidation . . . 244
Strategy 3: Write-Through Cache . . . 244
9. TTL Management . . . 245
Dynamic TTL . . . 245
10. Cache Statistics . . . 246
11. Combining Cache Layers . . . 246
12. Exercise: Cache an Expensive Product Listing Endpoint . . . 247
Requirements . . . 247
Test with: . . . 248

13. Solution . . . 248

14. Gotchas . . . 250

1. Caching Authenticated Responses . . . 250

2. Cache Stampede . . . 250

3. Memory Cache Lost on Restart . . . 250

4. Stale Data After Database Update . . . 250

5. Cache Key Collisions . . . 251

6. Serialization Overhead . . . 251

7. Forgetting to Set TTL . . . 251

Queue System **252**

1. Do Not Make the User Wait . . . 252

2. Why Queues Matter . . . 252

3. File Queue (Default) . . . 252

Creating a Queue and Pushing a Job . . . 253

Convenience Method: produce . . . 253

Push with Priority . . . 253

Queue Size . . . 253

4. Pushing from Route Handlers . . . 254

5. Consuming Jobs . . . 254

Retry with Delay . . . 254

Manual Pop . . . 255

6. Job Lifecycle . . . 255

Job Methods . . . 255

Job Properties . . . 255

7. Retry and Dead Letters . . . 256

Max Retries . . . 256

Retrying Failed Jobs . . . 256

Dead Letters . . . 256
Purging Jobs . . . 256
8. Producing Multiple Jobs . . . 256
9. Switching Backends . . . 257
Default: File . . . 257
RabbitMQ . . . 257
Kafka . . . 257
MongoDB . . . 257
10. Separate Producer and Consumer Patterns . . . 257
11. Exercise: Build an Email Queue . . . 258
Requirements . . . 258
Test with: . . . 258
12. Solution . . . 258
13. Gotchas . . . 260
1. Always call complete or fail . . . 260
2. Worker not picking up messages . . . 260
3. Payload must be serializable . . . 260
4. Dead letters pile up . . . 260
5. File backend for production . . . 260
6. Consumer block returns nothing useful . . . 261
Events **262**
1. Decouple With Events . . . 262
2. Basic Usage . . . 262
Registering a Listener . . . 262
Emitting an Event . . . 262
3. Multiple Listeners . . . 262

4. Priority . . . 263

5. Once: Fire a Listener One Time . . . 263

6. off: Remove a Listener . . . 264

7. clear: Remove All Listeners . . . 264

8. Events in Route Handlers . . . 264

9. Listing Registered Events . . . 265

10. Gotchas . . . 266

1. Listeners run synchronously . . . 266

2. Exceptions in listeners . . . 266

3. clear in tests . . . 266

4. once is not thread-safe by design . . . 266

Localization **267**

1. Your App Speaks One Language . . . 267

2. Locale Files . . . 267

3. Basic Translation . . . 268

4. Interpolation . . . 268

5. Switching Locales . . . 268

6. Fallback Locale . . . 268

7. Per-Request Locale in Routes . . . 269

8. Available Locales . . . 269

9. Locale in Templates . . . 270

10. Date and Number Formatting . . . 270

11. Gotchas . . . 270

1. Missing locale file returns fallback silently . . . 270

2. Nested key typo . . . 271

3. Reload locale files in development . . . 271

4. Interpolation keys are case-sensitive . . . 271

Structured Logging **272**

1. puts Is Not a Logging Strategy . . . 272
2. Log Levels . . . 272
3. Basic Usage . . . 272
4. Controlling the Log Level . . . 273
5. Setting the Level in Code . . . 273
6. Logging in Route Handlers . . . 273
7. Logging Errors with Exceptions . . . 274
8. Request Logging Middleware . . . 274
9. JSON Output Format . . . 274
10. Writing to a File . . . 275
11. Checking the Current Level . . . 275
12. Gotchas . . . 275
    1. Never log passwords or tokens . . . 275
    2. Avoid string interpolation in the message . . . 275
    3. DEBUG in production floods logs . . . 275
    4. fatal does not exit . . . 275

Email with Messenger **276**

1. Every App Sends Email . . . 276
2. Messenger Configuration via .env . . . 276
    Common Provider Configurations . . . 277
3. Constructor Override Pattern . . . 277
4. Sending Plain Text Email . . . 277
    The send Method Signature . . . 278
5. Sending HTML Email with Text Fallback . . . 278

6. Adding Attachments . . . 279
Multiple Attachments . . . 280
7. CC, BCC, and Reply-To . . . 280
8. Custom Headers . . . 280
9. Reading Inbox via IMAP . . . 281
Listing Inbox Messages . . . 281
Reading a Specific Message . . . 282
Searching Messages . . . 282
Other IMAP Operations . . . 283
10. Dev Mode: Email Interception . . . 283
Disabling Interception . . . 283
11. Using Templates for Email Content . . . 283
Rendering and Sending . . . 284
12. Sending Email via Queues . . . 285
13. Exercise: Build an Email Notification System . . . 286
Requirements . . . 286
Test with: . . . 286
14. Solution . . . 286
15. Gotchas . . . 287
1. Gmail Requires App Passwords . . . 287
2. Emails Go to Spam . . . 287
3. HTML Email Rendering Differences . . . 287
4. Attachment File Not Found . . . 287
5. SMTP Connection Timeout . . . 288
6. Dev Mode Emails Disappear on Restart . . . 288
7. Unicode Characters Display as Question Marks . . . 288
8. IMAP Connection Fails . . . 288

Frontend with tina4css 289

1. The Problem with Frontend Toolchains . . . 289
2. What Ships with Tina4 . . . 289
3. The Grid System . . . 289
    Responsive Breakpoints . . . 290
4. Cards . . . 290
    Card Grid . . . 290
5. Tables . . . 291
6. Forms . . . 291
7. Buttons and Alerts . . . 292
    Buttons . . . 292
    Alerts . . . 292
8. frond.js -- AJAX Without jQuery . . . 292
    API Calls . . . 292
    Token Management . . . 293
    Form Submission via AJAX . . . 293
9. Dark Mode . . . 293
10. Building a Dashboard . . . 293
11. SCSS Customization . . . 295
    Live SCSS Compilation . . . 296
12. frond.js API Reference . . . 296
    Including the Scripts . . . 296
    tina4.min.js -- Core Utilities . . . 296
        loadPage(url, targetId) . . . 296
        saveForm(formId, url, method) . . . 296
        sendRequest(url, method, data, callback) . . . 297
    frond.min.js -- Template Engine Client-Side Helpers . . . 297

AJAX Requests . . . 297
Token Management . . . 297
Loading Indicators . . . 298
WebSocket Auto-Reconnect . . . 298
tina4js.min.js -- Reactive Frontend Framework . . . 298
Reactive State . . . 298
Web Components . . . 299
13. Responsive Design . . . 299
Responsive Columns . . . 299
Responsive Visibility . . . 299
14. Building a Users Page with AJAX . . . 300
15. Exercise: Build a Product Management Page . . . 300
Requirements . . . 301
16. Solution . . . 301
17. Gotchas . . . 301
1. tina4css Classes Do Not Work . . . 301
2. frond.js Not Loading . . . 301
3. AJAX Calls Return HTML Instead of JSON . . . 301
4. Forms Submit Twice . . . 302
5. Dark Mode Resets on Page Load . . . 302
6. Bootstrap Classes Not Working . . . 302
7. SCSS Changes Not Reflected . . . 302
18. HtmlElement — Programmatic HTML Builder . . . 302
Testing **303**
1. Why Tests Matter More Than You Think . . . 303
2. Your First Test . . . 303

3. RSpec Matchers Reference . . . 304
Equality . . . 304
Truthiness . . . 304
Comparison . . . 304
Strings . . . 304
Collections . . . 304
Exceptions . . . 305
Floating Point . . . 305
Type Checking . . . 305
4. Testing Routes . . . 305
TestClient Methods . . . 306
TestResult Properties . . . 306
5. Testing ORM Models . . . 307
6. Testing Authentication . . . 309
7. Test Database Isolation . . . 310
8. Setup and Teardown . . . 310
9. Running Tests . . . 311
10. Testing Best Practices . . . 311
Test One Thing Per Test . . . 311
Use Descriptive Test Names . . . 312
Isolate Tests . . . 312
Code Coverage . . . 312
11. Exercise: Write Tests for a Notes API . . . 312
Requirements . . . 312
12. Solution . . . 313
13. Gotchas . . . 315
1. Tests Share Database State . . . 315

2. TestClient Does Not Start a Real Server . . . 315
3. Test Order Dependency . . . 315
4. Database Migrations Not Applied . . . 315
5. Token Expired During Test . . . 315
6. Floating Point Comparison . . . 315
7. Test Output Too Verbose . . . 315

Scaffolding **316**

The CRUD Generator . . . 316
What Each File Contains . . . 316
Run It . . . 318
Individual Generators . . . 318
Model . . . 319
Route . . . 319
Migration . . . 319
Middleware . . . 319
Test . . . 319
Form . . . 319
View . . . 319
CRUD . . . 319
Auth . . . 320
AutoCRUD . . . 320
Usage . . . 320
The Auth Generator . . . 320
Field Types . . . 321
Table Naming Convention . . . 321
Combining Generators . . . 322
Exercise: Scaffold a Blog . . . 322

Gotchas . . . 323
Generators Do Not Overwrite . . . 323
Run Migrate After Generate . . . 323
File Naming Matters . . . 323
Field Changes Need New Migrations . . . 323
Singular Table Names . . . 323
API Documentation with Swagger 324
1. The 47-Endpoint Problem . . . 324
2. What Swagger/OpenAPI Is . . . 324
3. Enabling Swagger . . . 324
The Swagger JSON Endpoint . . . 325
4. Adding Descriptions to Routes . . . 325
Documenting Path Parameters . . . 325
Documenting Query Parameters . . . 326
5. Documenting Request and Response Schemas . . . 326
Request Body . . . 326
6. Tags for Grouping Endpoints . . . 327
7. Example Values . . . 327
8. Try-It-Out from the Swagger UI . . . 327
Authentication in Try-It-Out . . . 328
9. Customizing the Swagger Info Block . . . 328
10. Generating Client SDKs from the Spec . . . 328
11. A Complete Documented API . . . 328
12. Exercise: Document a Complete User API . . . 329
Requirements . . . 329
Test by visiting: . . . 330
13. Solution . . . 330

14. Gotchas . . . 332
1. Annotations Must Be Directly Above the Route . . . 332
2. Missing @tags Makes Endpoints Hard to Find . . . 332
3. @body Must Be Valid JSON . . . 332
4. Swagger Shows Routes You Did Not Annotate . . . 332
5. Response Examples Do Not Match Actual Responses . . . 332
6. Swagger UI Not Available in Production . . . 332
7. SDK Generation Produces Incorrect Types . . . 332

API Client **333**

1. Talking to Other Services . . . 333
2. Creating a Client . . . 333
With Default Headers . . . 333
3. GET Requests . . . 333
With Query Parameters . . . 333
Response Object . . . 333
4. POST Requests . . . 334
5. PUT Requests . . . 334
6. DELETE Requests . . . 334
7. Authentication Headers . . . 334
Bearer Token . . . 334
API Key Header . . . 335
Basic Auth . . . 335
Per-Request Header Override . . . 335
8. Timeouts . . . 335
9. Using the Client in Route Handlers . . . 336
10. Shared Client Instances . . . 336

11. Error Handling . . . 337
12. Gotchas . . . 337
1. Never hard-code credentials . . . 337
2. Check success? before reading body . . . 337
3. Timeout is per-request . . . 337
4. Base URL trailing slash . . . 337
GraphQL and SOAP **338**
1. The Problem GraphQL Solves . . . 338
2. GraphQL vs REST -- A Quick Comparison . . . 338
3. Enabling GraphQL . . . 339
4. Defining Queries . . . 339
Testing the Queries . . . 340
5. Mutations . . . 340
Testing Mutations . . . 342
6. Nested Types and Relationships . . . 343
7. Auto-Generating Schema from ORM Models . . . 344
Example: ORM + Custom Queries . . . 345
8. Authentication in GraphQL . . . 345
9. The GraphiQL Playground . . . 346
10. Query Variables . . . 346
11. Exercise: Build a GraphQL API for a Blog . . . 347
Requirements . . . 347
Test with these queries: . . . 348
12. Solution . . . 348
13. Input Validation . . . 350
Example: Missing Required Argument . . . 351

14. Field-Level Auth Directives . . . 351
Passing Auth Context . . . 352
Schema Example . . . 352
Behavior on Failed Auth . . . 352
15. Gotchas . . . 353
1. Resolver Not Called . . . 353
2. Nested Data Returns Wrong Shape . . . 353
3. Mutation Returns Null . . . 353
4. N+1 Query Problem . . . 353
5. GraphQL Playground Returns 404 . . . 354
6. Type Mismatch Between Schema and Data . . . 354
7. Authentication Not Applied . . . 354
8. Error Messages Expose Internal Details . . . 354
14. SOAP / WSDL Services . . . 355
What is SOAP? . . . 355
Defining a SOAP Service . . . 355
Type Annotations Map to XSD . . . 356
A More Complete Example . . . 357
Lifecycle Hooks . . . 357
Testing with curl . . . 358
When to Use SOAP vs REST vs GraphQL . . . 358
Real-time with WebSocket 359
1. The Refresh Button Problem . . . 359
2. What WebSocket Is . . . 359
3. Router.websocket -- WebSocket as a Route . . . 359
Shorthand Syntax . . . 360
Starting the Server . . . 360

4. Connection Events . . . 360
A Complete Handler . . . 361
5. Connection Methods . . . 361
Sending to a Single Client . . . 361
Closing a Connection . . . 362
6. Broadcasting to All Clients . . . 362
Broadcast Excluding Sender . . . 363
7. Path Parameters and Scoped Isolation . . . 363
8. Building a Live Chat . . . 364
WebSocket Handler . . . 364
9. Live Notifications . . . 365
10. Connecting from JavaScript . . . 366
Using Plain WebSocket . . . 366
Using Frond.js . . . 366
Auto-Reconnect . . . 367
11. A Complete Chat Page . . . 367
12. Server Compatibility . . . 368
13. Exercise: Build a Real-Time Chat Room . . . 369
Requirements . . . 369
Test by: . . . 369
14. Solution . . . 369
15. Scaling with a Backplane . . . 370
Configuration . . . 370
Requirements . . . 371
16. Gotchas . . . 371
1. WebSocket Needs a Persistent Server . . . 371
2. Events Are Symbols, Not Strings . . . 371

3. Messages Are Strings, Not Objects . . . 371
4. Connection Count Is Per-Path . . . 371
5. Broadcasting Does Not Scale Across Servers . . . 371
6. Large Messages Cause Disconnects . . . 371
7. Memory Leak from Tracking Connected Users . . . 372
8. No Authentication on WebSocket Connect . . . 372

Server-Sent Events (SSE) 373

1. The Polling Problem . . . 373
2. What SSE Is . . . 373
3. Your First Stream . . . 374
4. The SSE Protocol . . . 375
The data: Field . . . 375
The event: Field . . . 375
The id: Field . . . 375
The retry: Field . . . 375
The Delimiter . . . 376
5. Frontend: EventSource . . . 376
Basic Usage . . . 376
Named Events . . . 376
Closing the Connection . . . 377
With Authentication . . . 377
6. Real Examples . . . 377
Live Dashboard . . . 377
Build Progress . . . 378
LLM Chat Streaming . . . 378
File Processing Progress . . . 379
7. Custom Content Types . . . 380

NDJSON Streaming . . . 380
Consuming NDJSON on the Client . . . 380
8. SSE vs WebSocket . . . 381
9. Production Considerations . . . 382
Nginx Proxy Buffering . . . 382
Connection Limits . . . 382
Heartbeat Messages . . . 382
10. Exercise: Build a Live Metrics Dashboard . . . 383
Requirements . . . 383
Test by: . . . 383
11. Solution . . . 383
12. What You Learned . . . 385
WSDL / SOAP **386**
1. When the World Still Uses SOAP . . . 386
2. What SOAP and WSDL Are . . . 386
3. Defining a WSDL Service . . . 386
4. Mounting the Service . . . 387
5. Auto-Generated WSDL . . . 387
6. Calling the SOAP Service . . . 388
7. Calling a Remote WSDL Service . . . 388
With Authentication . . . 389
8. Complex Types . . . 389
9. Error Handling in Operations . . . 389
10. Gotchas . . . 390
1. WSDL caching in production . . . 390
2. Character encoding . . . 390

3. Namespace conflicts . . . 390

4. SOAP is not REST . . . 390

Dependency Injection Container **391**

1. The Problem With Hard Dependencies . . . 391

2. Registering a Service . . . 391

Registering Multiple Services . . . 391

3. Resolving a Service . . . 392

4. Checking Registration . . . 392

5. Using Services in Route Handlers . . . 392

6. clear!: Reset the Container . . . 393

7. Testing With the Container . . . 393

8. Service Dependencies . . . 394

9. Gotchas . . . 395

1. clear! drops cached instances . . . 395

2. Singletons by default . . . 395

3. Circular dependencies cause infinite loops . . . 395

4. Registration order matters . . . 395

Service Runner **396**

1. Work That Runs Forever . . . 396

2. Defining a Service . . . 396

3. Starting All Services . . . 396

4. Starting a Single Service . . . 396

5. Stopping Services . . . 397

6. Checking Service Status . . . 397

7. Automatic Restart on Failure . . . 397

8. Queue Consumer Service . . . 397

9. Scheduled Task Service . . . 398
10. Metrics Collector Service . . . 399
11. Listing Registered Services . . . 399
12. Shutdown Hooks . . . 399
13. Gotchas . . . 400
1. Services share the main process . . . 400
2. sleep is required in loops . . . 400
3. Thread safety . . . 400
4. stop_all on shutdown . . . 400
MCP Dev Tools **401**
1. What is MCP? . . . 401
2. How It Works . . . 401
Localhost-Only Security . . . 401
3. Connecting AI Tools . . . 401
Claude Code . . . 401
Cursor . . . 402
Manual Connection . . . 402
4. Built-in Tools . . . 402
Database . . . 402
Routes . . . 402
Templates . . . 403
Files . . . 403
Migrations . . . 403
Data and ORM . . . 403
Infrastructure . . . 404
Debugging . . . 404
Project introspection . . . 404

Persistent code index . . . 405
Framework documentation (prose) . . . 405
Live API RAG (reflection) . . . 405
Plan management . . . 405
5. Protocol Details . . . 406
SSE Endpoint (GET) . . . 406
Message Endpoint (POST) . . . 406
Lifecycle . . . 407
6. Troubleshooting . . . 407
Building Custom MCP Servers **408**
1. Beyond Dev Tools . . . 408
2. Creating an MCP Server . . . 408
3. Registering Tools with mcp_tool . . . 408
4. Registering Resources with mcp_resource . . . 409
5. Class-Based MCP Services . . . 409
6. Securing MCP Endpoints . . . 410
7. Testing MCP Tools . . . 411
8. Complete Example: CRM MCP Server . . . 411
9. Best Practices . . . 412
Dev Tools **414**
1. Debugging at 2am . . . 414
2. Enabling the Dev Dashboard . . . 414
3. Dashboard Overview . . . 414
System Overview . . . 414
4. The Dev Toolbar . . . 415
Disabling the Toolbar . . . 415

5. Error Overlay . . . 415
Example Error . . . 416
Error Overlay in Production . . . 416
6. The Gallery . . . 416
7. Live Reload . . . 417
How It Works . . . 417
8. Request Inspector . . . 418
Request Details Panel . . . 418
Filtering Requests . . . 418
Request Replay . . . 418
9. SQL Query Runner . . . 419
Safety . . . 419
10. Queue Monitor . . . 419
11. Log Viewer . . . 420
Logging from Code . . . 420
12. System Info . . . 420
13. Health Check Endpoint . . . 421
14. Exercise: Debug a Broken Endpoint . . . 421
Setup . . . 421
Requirements . . . 422
15. Solution . . . 422
16. Gotchas . . . 423
1. Dev Dashboard Accessible on Network . . . 423
2. Live Reload Causes Connection Drops . . . 423
3. Error Overlay Shows in Production . . . 423
4. Request Inspector Slows Down the Server . . . 423
5. SQL Runner Executes Destructive Queries . . . 423

6. Template Changes Not Reflected . . . 423
7. Log Files Growing Without Bound . . . 424
8. Queue Monitor Shows No Jobs . . . 424
9. Memory Usage Grows During Development . . . 424

Tina4 CLI **425**

1. Getting a New Developer Up to Speed . . . 425
2. tina4 init -- Project Scaffolding . . . 425
   - Language Detection . . . 425
   - Explicit Language Selection . . . 426
   - Init into an Existing Directory . . . 426
3. tina4 serve -- Dev Server . . . 426
   - Options . . . 426
   - Bypassing the CLI . . . 427
4. tina4 generate model -- ORM Scaffolding . . . 427
   - Adding Fields . . . 427
   - Field Types . . . 428
   - Options . . . 428
5. tina4 generate route -- CRUD Route Scaffolding . . . 429
   - Options . . . 430
6. tina4 generate migration -- Migration Scaffolding . . . 430
7. tina4 generate middleware -- Middleware Scaffolding . . . 431
8. Generate All at Once . . . 431
9. tina4 doctor -- Health Check . . . 431
10. tina4 test -- Running Tests . . . 432
    - Test Options . . . 432
11. tina4 routes -- Route Listing . . . 433
    - Filtering . . . 433

12. tina4 migrate -- Database Migrations . . . 433
Rollback . . . 433
Migration Table Auto-Upgrade . . . 434
Status . . . 434
Other Migration Commands . . . 434
13. tina4 queue -- Queue Management . . . 434
14. tina4 build -- Build Commands . . . 434
15. tina4 deploy -- Deployment . . . 434
16. Environment-Specific Commands . . . 435
17. Exercise: Scaffold a Feature in 5 Commands . . . 435
Requirements . . . 435
Expected Commands . . . 435
18. Solution . . . 435
19. Gotchas . . . 437
1. tina4 Command Not Found . . . 437
2. Wrong Language Detected . . . 437
3. Generated Files Overwrite Existing Code . . . 437
4. Migration Order Issues . . . 437
5. tina4 serve Uses Wrong Port . . . 437
6. Doctor Shows False Warnings . . . 437
7. Model Name Must Be PascalCase . . . 438
8. build:css Fails . . . 438
9. deploy:docker Generates Wrong Dockerfile . . . 438
20. Documentation . . . 438
21. Test Port (Dual-Port Development) . . . 438

Why Tina4 is Built for AI-Assisted Development . . . 440
The Zero-Dependency Advantage . . . 440
CLAUDE.md -- The AI's Instruction Manual . . . 440
Skills -- Deep Framework Knowledge . . . 441
tina4-maintainer . . . 441
tina4-developer . . . 441
tina4-js . . . 441
Supported AI Tools . . . 441
The AI Chat in Dev Dashboard . . . 442
The Convention Advantage . . . 442
Prompt Engineering for Tina4 . . . 442
Creating a New Feature . . . 442
Adding Authentication . . . 443
Building a Dashboard . . . 443
Adding WebSocket . . . 443
The Ruby Advantage for AI . . . 443
Block Syntax . . . 443
Hash Syntax . . . 443
Enumerable Methods . . . 444
Vibe Coding Workflow . . . 444
1. Describe What You Want . . . 444
2. Review the Generated Code . . . 444
3. Run and Test . . . 444
4. Iterate . . . 444
5. Deploy . . . 444
Real-World Example: Building an API in 5 Minutes . . . 445
Why This Matters . . . 446

Exercise: Vibe-Code a Feature . . . 446
Gotchas . . . 446
1. AI Generates Rails Syntax Instead of Tina4 . . . 446
2. AI Invents Non-Existent Methods . . . 447
3. AI Generates Tests That Do Not Match the Implementation . . . 447
4. AI Forgets to Run Migrations . . . 447
5. AI Uses require Instead of Auto-Loading . . . 447
6. AI Generates Over-Engineered Solutions . . . 447
7. AI Does Not Know About to_h . . . 447
Environment Variables 448
Core Server . . . 449
Routing . . . 451
Secrets and Authentication . . . 451
Database . . . 452
CORS . . . 453
Security Headers . . . 453
Rate Limiting . . . 454
Sessions . . . 454
Valkey/Redis session backend . . . 455
Cache . . . 455
Templates . . . 456
GraphQL . . . 456
Queues . . . 456
Kafka queue backend . . . 457
RabbitMQ queue backend . . . 457
MongoDB queue backend . . . 457
WebSocket . . . 458

Email . . . 459
Logging . . . 460
Uploads . . . 461
Services (background tasks) . . . 461
Localisation . . . 461
AI and MCP Tooling . . . 462
MCP . . . 463
Swagger / OpenAPI . . . 463
Minimal .env for Development . . . 464
Minimal .env for Production . . . 464
Deployment **465**
1. From Development to Production . . . 465
2. Production .env Configuration . . . 465
Key Differences from Development . . . 466
Sensitive Values . . . 466
3. Puma Configuration . . . 466
How Many Workers? . . . 467
4. Docker Deployment . . . 467
Dockerfile . . . 468
.dockerignore . . . 468
Docker Compose . . . 468
Building and Running . . . 469
5. Health Checks . . . 470
6. Graceful Shutdown . . . 470
Configuring Shutdown Timeout . . . 471
Docker Stop Grace Period . . . 471

Graceful Restart Pattern . . . 471
7. Log Rotation . . . 471
Using logrotate (Linux) . . . 471
Docker Logging . . . 472
8. Reverse Proxy with Nginx . . . 472
9. SSL/TLS with Let's Encrypt . . . 473
With Nginx and Certbot . . . 473
With Docker (Using Traefik as Reverse Proxy) . . . 474
Certificate Monitoring . . . 474
10. Process Management with systemd . . . 474
11. Scaling . . . 475
Multiple Workers . . . 475
Load Balancing with Nginx . . . 475
Docker Scaling . . . 476
Scaling Considerations . . . 476
12. Monitoring . . . 476
Log Aggregation . . . 477
Prometheus Metrics . . . 477
Uptime Monitoring . . . 477
Application Performance Monitoring (APM) . . . 477
13. Exercise: Docker Deploy . . . 478
Requirements . . . 478
14. Solution . . . 479
15. Gotchas . . . 479
1. .env Not Loaded in Docker . . . 479
2. SQLite Database Lost on Container Restart . . . 479
3. WebSocket Connections Drop Behind Nginx . . . 479

4. Puma Workers Not Reconnecting to Database . . . 480
5. Built-in Server Used in Production . . . 480
6. Static Files Slow . . . 480
7. Memory Usage Grows Over Time . . . 480
8. SSL Certificate Not Renewing . . . 480
9. Scaled Instances Have Different State . . . 481
10. Queue Worker Stops Processing . . . 481

Building a Complete App **482**

1. Putting It All Together . . . 482
2. Planning the App . . . 482
Models . . . 482
Relationships . . . 482
Routes . . . 483
Templates . . . 483
3. Step 1: Init Project and Set Up Database . . . 483
Create Migrations . . . 483
4. Step 2: ORM Models . . . 484
5. Step 3: Authentication . . . 485
Test Registration and Login . . . 486
6. Step 4: Task CRUD . . . 487
Test the Task API . . . 490
7. Step 5: Dashboard Stats with Caching . . . 491
8. Step 6: WebSocket Real-Time Updates . . . 492
9. Step 7: Email Notifications . . . 492
10. Step 8: Dashboard Template . . . 493
11. Step 9: Tests . . . 495
12. Step 10: Docker Deployment . . . 497

13. The Complete Project Structure . . . 498
14. What You Built . . . 498
15. What to Build Next . . . 499
16. Gotchas . . . 500
1. Circular Dependencies Between Models . . . 500
2. Cache Not Invalidated on Related Model Changes . . . 500
3. Queue Worker Not Processing Emails . . . 500
4. Token Expired During Long Sessions . . . 500
5. WebSocket Notifications Not Received . . . 500
6. Database Locked Under Load . . . 500
7. Missing Migration Before Deployment . . . 500
17. Closing Thoughts -- The Tina4 Philosophy . . . 501
Release Notes **502**
v3.12.4 (2026-05-06) . . . 502
25 new env vars across all 4 frameworks . . . 502
Logging — env-driven file output + rotation . . . 502
Documentation-truth CI gate now strict on both axes . . . 503
Tests added . . . 503
Upgrade path . . . 503
v3.12.3 (2026-05-05) . . . 503
Breaking changes (Ruby + PHP only) . . . 503
Doc parity — CLAUDE.md and book chapter 33 . . . 504
Other fixes . . . 504
Genuine gaps surfaced by the parity audit (follow-up, not blocking 3.12.3) . . . 504
Upgrade path . . . 505
v3.12.2 (2026-05-05) . . . 505
Firebird URL auto-detect . . . 505

Bug fix specific to PHP — mysqli localhost+port quirk . . . 506
Other fixes . . . 506
v3.12.1 (2026-05-04) . . . 506
v3.12.0 (2026-05-04) . . . 506
Why this release . . . 507
What changed . . . 507
Bug fixes shipped alongside the rename . . . 507
Migration — every renamed var . . . 508
Names that stay un-prefixed (not framework config) . . . 509
How to upgrade . . . 509
Parity . . . 509
v3.11.32 (2026-04-25) . . . 509
v3.11.13 (2026-04-16) . . . 510
v3.11.12 (2026-04-16) . . . 511
v3.11.11 (2026-04-16) . . . 512
v3.11.10 (2026-04-15) . . . 512
v3.11.9 (2026-04-15) . . . 513
v3.10.99 (2026-04-12) . . . 513
v3.10.97 (2026-04-11) . . . 514
v3.10.93 (2026-04-11) . . . 514
v3.10.92 (2026-04-10) . . . 514
v3.10.91 (2026-04-10) . . . 514
v3.10.90 (2026-04-09) . . . 515
v3.10.89 (2026-04-09) . . . 515
v3.10.86 (2026-04-09) . . . 515
v3.10.85 (2026-04-09) . . . 515

v3.10.84 (2026-04-09) . . . 516
v3.10.83 (2026-04-08) . . . 516
v3.10.70 (2026-04-06) . . . 516
v3.10.68 (2026-04-03) — Full Parity Release . . . 516
v3.10.67 (2026-04-03) . . . 517
v3.10.66 (2026-04-03) . . . 517
v3.10.65 (2026-04-03) . . . 517
v3.10.60 (2026-04-03) . . . 518
v3.10.57 (2026-04-02) . . . 518
v3.10.54 (2026-04-02) . . . 518
v3.10.48 — April 2, 2026 . . . 519
Bug Fixes . . . 519
v3.10.46 — April 1, 2026 . . . 519
Test Coverage . . . 519
v3.10.45 — April 1, 2026 . . . 519
Notes . . . 519
v3.10.44 — April 1, 2026 . . . 519
New Features . . . 519
Bug Fixes . . . 519
Test Coverage . . . 520
v3.10.40 — April 1, 2026 . . . 520
Bug Fixes . . . 520
v3.10.39 — April 1, 2026 . . . 520
New Features . . . 520
v3.10.38 -- April 1, 2026 . . . 521
Code Metrics & Bubble Chart . . . 521
AI Context Installer . . . 521

Dashboard Improvements . . . 521
Cleanup . . . 521
v3.10.x -- Previous Releases . . . 521
v3.10.29 -- Version Parity (March 30) . . . 521
v3.10.27 -- Frond Macro HTML Escaping Fix (March 30) . . . 521
v3.10.25 -- ORM Transaction Fix (March 30) . . . 522
v3.10.22 -- Unique Form Tokens (March 30) . . . 523
v3.10.18 -- Frond Ternary Parser Fix (March 29) . . . 523
v3.10.16 -- Template Filters: tojson, jsescape (March 28) . . . 524
v3.10.15 -- Replace Filter Backslash Fix (March 28) . . . 524
v3.10.14 -- getnextid() (March 28) . . . 524
v3.10.13 -- ORM Auto-Commit on Write (March 28) . . . 525
v3.10.12 -- Session GC and NATS Backplane (March 28) . . . 525
v3.10.11 -- Frond Variable Key Access (March 28) . . . 525
v3.10.10 -- Firebird Migration Runner Fixes (March 28) . . . 526
v3.10.6 -- WSDL/SOAP Rewrite (March 28) . . . 526
v3.10.5 -- Frond Quote-Aware Operator Matching (March 28) . . . 526
v3.10.4 -- Auto-CRUD REST Endpoint Generator (March 28) . . . 526
v3.10.2 -- Frond Hash Method Calls (March 28) . . . 526
v3.10.1 -- autoMap and Case Conversion (March 28) . . . 526
v3.10.0 -- Optimized For-Loops (March 28) . . . 526
v3.9.x . . . 527
v3.9.0 -- QueryBuilder, Sessions, Path Injection (March 26) . . . 527
v3.9.1 -- Security Defaults (March 27) . . . 527
v3.9.2 -- NoSQL QueryBuilder, WebSocket Backplane (March 27) . . . 528
v3.8.x . . . 528
v3.8.0 -- Base64 Filters, Template Cache (March 25) . . . 528

v3.8.1 -- Security Headers Middleware (March 25) . . . 529
v3.8.2 -- Connection Pooling (March 26) . . . 529
v3.8.3 -- Claude Code Commands (March 26) . . . 529
v3.8.7 -- Benchmark and Stability (March 26) . . . 529
v3.7.x . . . 529
v3.7.0 -- Template Auto-Serve, Firebird Migrations (March 25) . . . 529
v3.7.1 -- Full Template Auto-Serve (March 25) . . . 529
v3.6.x . . . 530
v3.6.0 -- Architectural Parity (March 25) . . . 530
v3.5.x . . . 530
v3.5.0 -- Bundled Frontend, Swagger CRUD, Middleware (March 24) . . . 530
v3.4.x . . . 530
v3.4.0 -- Auth, WebSocket, DatabaseResult (March 24) . . . 531
v3.3.x . . . 532
v3.3.0 -- Queue API, Route Chaining (March 24) . . . 532
v3.2.x . . . 533
v3.2.0 -- Flexible Route Handlers (March 23) . . . 533
v3.1.x . . . 534
v3.1.0 -- ORM Relationships, Caching, Queues (March 22) . . . 534
v3.1.1 -- DevMailbox Fix (March 22) . . . 535
v3.1.2 -- Documentation Fixes (March 22) . . . 535
v3.0.x . . . 535
v3.0.0 -- Initial Release (March 21) . . . 536
Pre-Release (v0.x) . . . 536

Complete Feature List **537**

1. Core Server . . . 537
2. Database . . . 538

3. Authentication and Sessions 539

4. Templates and Frontend 539

5. Caching 540

6. Background Processing 540

7. Events 541

8. Localization 541

9. Logging 542

10. API and Integrations 543

11. Developer Tools 544

Feature Summary by Version 544

Quick Reference: Environment Variables 545

--- outline: deep ---

<div v-pre>

Getting Started with Tina4 Ruby

1. What Is Tina4 Ruby

Tina4 Ruby is a zero-dependency web framework for Ruby 3.1+. One gem. Routing, ORM, template engine, authentication, queues, WebSocket, and 70 other features -- all built in.

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

Tina4 Ruby follows Ruby conventions. Method names use `snake_case` -- `fetch_one`, `soft_delete`, `has_many`. Class names use `PascalCase`. Constants use `UPPER_SNAKE_CASE`.

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

---

2. Prerequisites and Installation

What You Need

Four things. Nothing more.

- **Ruby 3.1 or later** -- check with:

```
ruby -v
```

You should see output like:

```
ruby 3.3.0 (2023-12-25 revision 5124f9ac75) [arm64-darwin23]
```

Below 3.1? Upgrade first.

- **Bundler** -- Ruby's dependency manager:

```
bundle --version
```

```
Bundler version 2.5.6
```

Not installed? `gem install bundler`.

- **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:

```
tina4 --version
```

```
tina4 0.1.0
```

- **SQLite3 development libraries** -- most systems ship these:

```
# macOS (already included)
# Ubuntu/Debian
sudo apt-get install libsqlite3-dev

# Fedora
sudo dnf install sqlite-devel
```

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:

```
tina4 init ruby 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.

```
Creating Tina4 project in ./my-store ...
  Detected language: Ruby (Gemfile)
  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 .keys/
  Created tests/
```

```
Project created! Next steps:
  cd my-store
  bundle install
  tina4 serve
```

Install the Ruby dependencies:

```
cd my-store
bundle install
```

```
Fetching gem metadata from https://rubygems.org/...
Resolving dependencies...
Installing tina4 (3.2.1)
Bundle complete! 1 Gemfile dependency, 1 gem installed.
```

One gem. No dependency tree. No version conflicts. Just `tina4`.

Starting the Dev Server

```
tina4 serve
```

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

  Tina4 Ruby v3.2.1
  Server running at http://0.0.0.0:7147
  Debug mode: ON
  Database: sqlite:///data/app.db
  Press Ctrl+C to stop
```

Open `http://localhost:7147`. The Tina4 welcome page appears.

Hit the health endpoint:

```
curl http://localhost:7147/health
```

```
{
  "status": "ok",
  "database": "connected",
  "uptime_seconds": 12,
  "version": "3.2.1",
  "framework": "tina4-ruby"
}
```

Your project is alive.

---

3. Project Structure Walkthrough

Here is what `tina4 init` built:

```
my-store/
■■■ .env                    # Your configuration (gitignored)
■■■ .env.example            # Template for other developers
■■■ .gitignore              # Pre-configured
■■■ Gemfile                 # Gem dependencies
■■■ Gemfile.lock            # Locked dependency versions
```

```
■■■ app.rb                  # Application entry point
■■■ 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/Twig 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)
■■■ .keys/                  # JWT keys (gitignored)
■■■ tests/                  # Your test files
```

Five directories matter:

- `src/routes/` -- Every `.rb` file here is auto-loaded at startup. Drop your route definitions here. Subdirectories work too.
- `src/orm/` -- Every `.rb` file here is auto-loaded. ORM model classes live 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` maps to `/images/logo.png`.
- `data/` -- The default SQLite database (`app.db`) lives here. Gitignored. Databases do not belong in version control.

---

4. Your First Route

Create `src/routes/greeting.rb`:

```
Tina4::Router.get("/api/greeting/{name}") do |request, response|
  name = request.params["name"]
  response.json({
    message: "Hello, #{name}!",
    timestamp: Time.now.iso8601
  })
end
```

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

> ***Cross-language note.*** *Ruby, PHP, and Node read path parameters from* `request.params` / `$request->params` / `req.params`*. Python passes them as function arguments instead. Same concept, two shapes — pick the chapter that matches*

your runtime.

Test It

```
http://localhost:7147/api/greeting/Alice
```

```
{
  "message": "Hello, Alice!",
  "timestamp": "2026-03-22T14:30:00+00:00"
}
```

Or curl:

```
curl http://localhost:7147/api/greeting/Alice
```

```
{"message":"Hello, Alice!","timestamp":"2026-03-22T14:30:00+00:00"}
```

Force pretty output:

```
curl "http://localhost:7147/api/greeting/Alice?pretty=true"
```

```
{
  "message": "Hello, Alice!",
  "timestamp": "2026-03-22T14:30:00+00:00"
}
```

Understanding What Happened

- You created a file in `src/routes/`. Tina4 discovered it at startup.
- `Tina4::Router.get("/api/greeting/{name}")` registered a GET route with a path parameter `{name}`.
- A request to `/api/greeting/Alice` matched the pattern. The router called your handler block.
- `request.params["name"]` returned `"Alice"` from the URL.
- `response.json(...)` serialized the hash to JSON, set `Content-Type: application/json`, and sent a `200 OK`.

Adding More HTTP Methods

Add a POST endpoint. Update `src/routes/greeting.rb`:

```
Tina4::Router.get("/api/greeting/{name}") do |request, response|
  name = request.params["name"]
  response.json({
    message: "Hello, #{name}!",
    timestamp: Time.now.iso8601
  })
end

Tina4::Router.post("/api/greeting") do |request, response|
  name = request.body["name"] || "World"
  language = request.body["language"] || "en"

  greetings = {
    "en" => "Hello",
    "es" => "Hola",
    "fr" => "Bonjour",
    "de" => "Hallo",
```

```
    "ja" => "Konnichiwa"
  }

  greeting = greetings[language] || greetings["en"]

  next response.json({
    message: "#{greeting}, #{name}!",
    language: language
  }, 201)
end.no_auth
```

::: tip Two Ruby-isms to notice

- `.no_auth` is chained on the block's return value. Tina4 secures `POST`, `PUT`, `PATCH`, and `DELETE` routes by default — without `no_auth` the example returns `401 Unauthorized`. In production, remove `no_auth` and send an `Authorization: Bearer <token>` header instead.
- `next response.json(...)` exits the block with the response value. Inside a Tina4 route block, never use the `return` keyword — that is a `LocalJumpError` in Ruby (blocks are not methods). Use `next` when you need an early exit, otherwise rely on the block's final expression.

:::

Test:

```
curl -X POST http://localhost:7147/api/greeting \
  -H "Content-Type: application/json" \
  -d '{"name": "Carlos", "language": "es"}'
```

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

Status code: `201 Created`.

---

5. Your First Template

Tina4 uses **Frond** -- a zero-dependency, Twig-compatible template engine built from scratch. If you know Twig, Jinja2, or Nunjucks, you know Frond.

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>
```

Two blocks: `title` and `content`. Child templates override only what they need. `tina4.css` provides built-in styling. `frond.js` provides JS helpers. Both ship with every project.

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{{ products | length != 1 ? "s" : "" }}</p>

    {% if products | length > 0 %}
        {% for product in products %}
            <div class="product-card">
                <h3>{{ product.name }}</h3>
                <p>{{ product.description }}</p>
                <p class="price">${{ product.price | number_format(2) }}</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 not loop.last %}
                    {# Don't add separator after the last item #}
                {% 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.rb`:

```
Tina4::Router.get("/products") do |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
    }
  ]

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

See It in the Browser

Open `http://localhost:7147/products`. You see:

- A dark nav bar with "Home" and "Products"
- The heading "Our Products"
- "Showing 4 products"
- Four product cards with name, description, price, and stock badge
- "Monitor Stand" wears a red "Out of Stock" badge
- The other three wear green "In Stock" badges

How Template Rendering Works

- `response.render("products.html", { products: products })` tells Frond to render `src/templates/products.html`.
- Frond sees `{% extends "base.html" %}` and loads the base template.
- `{% block content %}` in `products.html` replaces the same block in `base.html`.
- `{{ product.name }}` outputs the value, auto-escaped for HTML safety.
- `{{ product.price | number_format(2) }}` formats the number with 2 decimal places.

- `{% for product in products %}` loops through the array.
- `{% if product.in_stock %}` renders the right badge.
- `{{ products | length }}` returns the item count.

About tina4css

`tina4.css` is Tina4's built-in CSS utility framework. Layout utilities, typography, common UI patterns. No Bootstrap. No Tailwind. No npm. It ships with every scaffolded project.

---

6. Understanding .env

Open `.env` at the project root:

```
TINA4_DEBUG=true
```

That is it. The scaffold creates a minimal `.env`. Everything else uses defaults.

The defaults that matter:

Variable	Default Value	What It Means
`TINA4_PORT`	`7147`	Server runs on port 7147
`TINA4_DATABASE_URL`	`sqlite:///data/app.db`	SQLite database in `data/`
`TINA4_LOG_LEVEL`	`ALL`	All log messages output
`CORS_ORIGINS`	`*`	All origins allowed (fine for dev)
`TINA4_RATE_LIMIT`	`100`	100 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).
`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 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 Ruby 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 and navigate to:

```
http://localhost:7147/__dev
```

The dashboard shows:

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

The dev dashboard is a debugging powerhouse. It shows you what your application does without print statements.

HTML pages also get a **debug overlay** -- 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

The overlay vanishes when `TINA4_DEBUG=false`. Production users never see it.

---

8. Manual Setup (No CLI)

The `tina4` CLI scaffolds everything for you. But if you start from an empty folder — just Ruby and Bundler — here is the minimum you need.

Step 1: Create `Gemfile`

```
source "https://rubygems.org"

gem "tina4-ruby", "~> 3.0"

# Required: Ruby 3.0+ removed webrick from the standard library
gem "webrick", "~> 1.8"

# Required: the default SQLite database driver
gem "sqlite3", "~> 2.0"
```

Then install:

```
bundle install
```

::: warning Two dependencies you must declare

- `webrick` — Tina4's dev server uses WEBrick. Ruby 3.0 dropped WEBrick from the standard library, so you must list it in your `Gemfile` or `tina4 serve` fails with `LoadError: cannot load such file -- webrick`.
- `sqlite3` — the default `TINA4_DATABASE_URL` points to SQLite. Without this gem the server restarts in a loop with `LoadError: cannot load such file -- sqlite3`.

`tina4 init ruby` adds both gems automatically. You only need to add them by hand when bootstrapping an empty project. :::

Step 2: Create `app.rb`

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

```
require "tina4"
Tina4.initialize!(__dir__)
app = Tina4::RackApp.new
Tina4::WebServer.new(app, port: 7147).start
```

Four lines. `initialize!` sets up the project directory. `RackApp` builds the Rack application. `WebServer` starts it on the given port.

Step 3: Create the Folder Structure

Tina4 expects this layout:

```
my-project/
■■■ app.rb
```

```
■■■ Gemfile
■■■ .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:7147`. 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 Ruby refuses to start without the Rust CLI. To bypass it (for example inside a Docker image that already wraps the framework) set* `TINA4_OVERRIDE_CLIENT=true` *in* `.env` *and run* `bundle exec ruby app.rb` *directly.*

---

9. Request & Response Fundamentals

Before jumping into the exercises, let's consolidate how route handlers work in Tina4 Ruby. Every handler receives two arguments: `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["q"]                  # "laptop"
request.params["page"]               # "2" (always a string)
request.params["sort"] || "name"     # "name" (default -- param was not sent)
```

Reading URL Path Parameters

Route patterns like `/users/{id}` capture segments of the URL. Access them through `request.params`:

```
Tina4::Router.get("/users/{id:int}/posts/{slug}") do |request, response|
  id = request.params["id"]      # 5 (int, because of :int)
  slug = request.params["slug"]  # "hello-world" (string)
  response.json({ user_id: id, slug: slug })
end
```

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 hash automatically (as long as the client sends `Content-Type: application/json`):

```
Tina4::Router.post("/api/items") do |request, response|
  name = request.body["name"] || ""
  price = request.body["price"] || 0
  response.json({ received_name: name, received_price: price })
end
```

Reading Headers

Headers are available as a hash with their original casing:

```
content_type = request.headers["Content-Type"] || "not set"
auth_token = request.headers["Authorization"] || ""
custom = request.headers["X-Custom-Header"] || ""
```

Sending JSON Responses

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

```
response.json({ id: 1, name: "Widget" })        # 200 OK (default)
response.json({ id: 1, name: "Widget" }, 201)    # 201 Created
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:

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

For raw HTML without a template file:

```
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.rb`:

```
# 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 }
]

Tina4::Router.get("/api/books") do |request, response|
  # List all books. Supports ?author= filter and ?sort=year.
  author = request.params["author"] || ""
  sort_by = request.params["sort"] || ""

  result = books

  # Filter by author if the query param is present
  unless author.empty?
    result = result.select { |b| b[:author].downcase.include?(author.downcase) }
  end

  # Sort by year if requested
  if sort_by == "year"
    result = result.sort_by { |b| b[:year] }
  end

  response.json({ books: result, count: result.length })
end

Tina4::Router.get("/api/books/{id:int}") do |request, response|
  # Get a single book by ID. Returns 404 if not found.
  id = request.params["id"]
  book = books.find { |b| b[:id] == id }

  if book.nil?
    next response.json({ error: "Book with id #{id} not found" }, 404)
  end

  response.json(book)
end

Tina4::Router.post("/api/books") do |request, response|
  # Create a new book from the JSON body. Returns 201 on success.
  title = request.body["title"] || ""
  author = request.body["author"] || ""
  year = request.body["year"] || 0

  if title.empty? || author.empty?
    next response.json({ error: "title and author are required" }, 400)
  end

  new_book = {
    id: books.map { |b| b[:id] }.max + 1,
```

```
    title: title,
    author: author,
    year: year
  }
  books << new_book

  response.json(new_book, 201)
end.no_auth
```

::: tip `next` vs `return` inside route blocks Ruby raises `LocalJumpError: unexpected return` if you use `return` inside a block. Route handlers in Tina4 are blocks (`do ... end`), so use `next value` for early exits — it yields `value` as the block's result, just like `return` would inside a method. The final line of the block is its implicit return, so the terminal `response.json(...)` at the bottom needs no `next`. :::

Test it:

```
# List all books
curl http://localhost:7147/api/books

# Filter by author
curl "http://localhost:7147/api/books?author=gibson"

# Sort by year
curl "http://localhost:7147/api/books?sort=year"

# Get a single book
curl http://localhost:7147/api/books/2

# Get a book that does not exist (returns 404)
curl http://localhost:7147/api/books/99

# Create a new book
curl -X POST http://localhost:7147/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 two features from scratch. No peeking 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`)
- Defaults to `"Stranger"` if `name` is missing
- Returns JSON:

```
{
```

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

- Calculates `time_of_day` 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:

```
curl "http://localhost:7147/api/greet?name=Sarah"
curl "http://localhost:7147/api/greet"
```

Exercise Part B: Product List Page

Create a page at `GET /store` that:

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

Product data for 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: true }
  { name: "Water Bottle", category: "Fitness", price: 24.99, featured: false }
]
```

Expected browser output:

- A page titled "Our Store"
- "5 products, 3 featured"
- Product cards with name, category, price, and "Featured" badge on highlighted items
- Featured products wear a distinct visual style

---

11. Solutions

Solution A: Greeting API

Create `src/routes/greet.rb`:

```
Tina4::Router.get("/api/greet") do |request, response|
  name = request.params["name"] || "Stranger"
  hour = Time.now.hour

  time_of_day = if hour >= 5 && hour < 12
                  "morning"
                elsif hour >= 12 && hour < 17
                  "afternoon"
                elsif hour >= 17 && hour < 21
                  "evening"
                else
                  "night"
                end

  response.json({
    greeting: "Welcome, #{name}!",
    time_of_day: time_of_day
  })
end
```

Test:

```
curl "http://localhost:7147/api/greet?name=Sarah"

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

curl "http://localhost:7147/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{{ product.featured ? ' featured' : '' }}">
                <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">${{ product.price | number_format(2) }}</p>
            </div>
        {% endfor %}
    </div>
{% endblock %}
```

Create `src/routes/store.rb`:

```
Tina4::Router.get("/store") do |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, featured: true
    { name: "Water Bottle", category: "Fitness", price: 24.99, featured: false }
  ]

  featured_count = products.count { |p| p[:featured] }

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

Open `http://localhost:7147/store`**.** You see:

- A dark header reading "Our Store"
- "5 products, 3 featured"
- Five product cards in a grid

- Three cards (Espresso Machine, Standing Desk, Noise-Canceling Headphones) wear a yellow border, light yellow background, and "Featured" badge
- Two cards (Yoga Mat, Water Bottle) wear a standard white background with gray border
- Each card shows name, category, and price formatted to 2 decimal places

---

12. Gotchas

1. File not auto-discovered

Problem: You created a route file but the URL returns nothing.

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

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

2. "Uninitialized constant" errors

Problem: `NameError: uninitialized constant Tina4::Router`.

Cause: The Tina4 gem is not loaded.

Fix: Confirm your `Gemfile` includes `gem "tina4"` and you ran `bundle install`. Route files in `src/routes/` auto-load -- no `require` statements needed.

3. JSON response shows HTML

Problem: Your JSON endpoint returns HTML.

Cause: You returned a string instead of calling `response.json`. Plain strings become HTML in Tina4.

Fix: Use `response.json(data)` for JSON endpoints. Not `puts data.to_json`.

4. Template not found

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

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

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 7147)`.

Cause: Another process owns port 7147.

Fix: Stop the other process, or change the port:

```
TINA4_PORT=8080
```

Or: `tina4 serve --port 8080`.

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 (`Ctrl+Shift+R` or `Cmd+Shift+R`). If that fails, restart with `Ctrl+C` and `tina4 serve`.

7. .env not loaded

Problem: Environment variables have no effect.

Cause: The `.env` file must sit at the project root (same directory as `Gemfile`).

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

8. Debug mode in production

Problem: Production shows stack traces and query details.

Cause: `TINA4_DEBUG=true` in production.

Fix: Set `TINA4_DEBUG=false`. This hides debug information, enables HTML minification, and activates `.broken` file health checks.

Routing

1. How Routing Works in Tina4

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

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

The simplest possible route:

```
Tina4::Router.get("/hello") do |request, response|
  response.json({ message: "Hello, World!" })
end
```

Save that as `src/routes/hello.rb`, start the server, visit `http://localhost:7147/hello`:

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

One line registers the route. One block handles the request. Done.

---

2. HTTP Methods

Tina4 supports all five standard HTTP methods. Each one lives on `Tina4::Router`:

```
Tina4::Router.get("/products") do |request, response|
  response.json({ action: "list all products" })
end

Tina4::Router.post("/products") do |request, response|
  response.json({ action: "create a product" }, 201)
end

Tina4::Router.put("/products/{id}") do |request, response|
  id = request.params[:id]
  response.json({ action: "replace product #{id}" })
end

Tina4::Router.patch("/products/{id}") do |request, response|
  id = request.params[:id]
  response.json({ action: "update product #{id}" })
end

Tina4::Router.delete("/products/{id}") do |request, response|
  id = request.params[:id]
  response.json({ action: "delete product #{id}" })
end
```

Test each one:

```
curl http://localhost:7147/products

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

```
curl -X POST http://localhost:7147/products \
  -H "Content-Type: application/json" \
  -d '{"name": "Widget"}'
```

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

```
curl -X PUT http://localhost:7147/products/42
```

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

```
curl -X PATCH http://localhost:7147/products/42
```

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

```
curl -X DELETE http://localhost:7147/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 name in curly braces:

```
Tina4::Router.get("/users/{id}/posts/{post_id}") do |request, response|
  user_id = request.params[:id]
  post_id = request.params[:post_id]

  response.json({
    user_id: user_id,
    post_id: post_id
  })
end
```

```
curl http://localhost:7147/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.

> ***Auto-casting:*** *Tina4 automatically casts path parameter values that are purely numeric to integers. For example, requesting* `/users/42/posts/99` *will give you* `request.params[:id]` *as the integer* `42` *and* `request.params[:post_id]` *as the integer* `99` *-- 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 with a colon after the parameter name:

```
Tina4::Router.get("/orders/{id:int}") do |request, response|
  id = request.params[:id]
  response.json({
    order_id: id,
    type: id.class.name
  })
end
```

```
curl http://localhost:7147/orders/42

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

Pass a non-integer? The route refuses to match. You get a 404:

```
curl http://localhost:7147/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:

```
# Integer parameter -- only digits match, auto-cast to Integer
Tina4::Router.get("/products/{id:int}") do |request, response|
  id = request.params[:id] # Integer, e.g. 42
  response.json({
    product_id: id,
    type: id.class.name
  })
end

# Float parameter -- decimal numbers, auto-cast to Float
Tina4::Router.get("/products/{id:int}/price/{price:float}") do |request, response|
  id = request.params[:id]
  price = request.params[:price]
  response.json({
    product_id: id,
    price: price,
    type: price.class.name
  })
end

# Path parameter -- catch-all, captures remaining segments as a string
Tina4::Router.get("/files/{filepath:path}") do |request, response|
  filepath = request.params[:filepath]
  # filepath could be "images/photos/cat.jpg"
  response.json({
    filepath: filepath,
    type: filepath.class.name
  })
```

```
end

# Integer route -- matches digits, returns an Integer
curl http://localhost:7147/products/42

{"product_id":42,"type":"Integer"}

# Integer route -- non-integer gives a 404
curl http://localhost:7147/products/abc

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

# Path catch-all -- captures everything after /files/
curl http://localhost:7147/files/images/photos/cat.jpg

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

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

```
Tina4::Router.get("/search") do |request, response|
  q = request.params["q"] || ""
  page = (request.params["page"] || 1).to_i
  limit = (request.params["limit"] || 10).to_i

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

curl "http://localhost:7147/search?q=keyboard&page=2&limit=20"

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

Missing query parameter? `request.params["key"]` returns `nil`. Use `||` to provide defaults.

---

5. Route Groups

A set of routes shares a common prefix. `Tina4::Router.group` eliminates repetition:

```
Tina4::Router.group("/api/v1") do

  Tina4::Router.get("/users") do |request, response|
    response.json({ users: [] })
  end
```

```
  Tina4::Router.get("/users/{id:int}") do |request, response|
    id = request.params[:id]
    response.json({ user: { id: id, name: "Alice" } })
  end

  Tina4::Router.post("/users") do |request, response|
    response.json({ created: true }, 201)
  end

  Tina4::Router.get("/products") do |request, response|
    response.json({ products: [] })
  end

end
```

These routes register as `/api/v1/users`, `/api/v1/users/{id}`, and `/api/v1/products`. Short paths inside the group. Tina4 prepends the prefix.

```
curl http://localhost:7147/api/v1/users

{"users":[]}

curl http://localhost:7147/api/v1/products

{"products":[]}
```

Groups nest:

```
Tina4::Router.group("/api") do
  Tina4::Router.group("/v1") do
    Tina4::Router.get("/status") do |request, response|
      response.json({ version: "1.0" })
    end
  end

  Tina4::Router.group("/v2") do
    Tina4::Router.get("/status") do |request, response|
      response.json({ version: "2.0" })
    end
  end
end

curl http://localhost:7147/api/v1/status

{"version":"1.0"}

curl http://localhost:7147/api/v2/status

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

---

6. Middleware

Middleware is code that runs before or after your route handler. Authentication. Logging. Rate limiting. Input validation. Anything that belongs on multiple routes but not in every handler.

Middleware on a Single Route

Pass middleware as the third argument:

```
log_request = lambda do |request, response, next_handler|
  start = Process.clock_gettime(Process::CLOCK_MONOTONIC)
  $stderr.puts "[#{Time.now.strftime('%Y-%m-%d %H:%M:%S')}] #{request.method} #{request.path}"

  result = next_handler.call(request, response)

  duration = ((Process.clock_gettime(Process::CLOCK_MONOTONIC) - start) * 1000).round(2)
  $stderr.puts "  Completed in #{duration}ms"

  result
end

Tina4::Router.get("/api/data", middleware: "log_request") do |request, response|
  response.json({ data: [1, 2, 3] })
end
```

The middleware receives `request`, `response`, and `next_handler`. Call `next_handler.call(request, response)` to proceed to the route handler. Skip the call and the handler never runs -- a gatekeeper that blocks unauthorized requests.

Blocking Middleware

Middleware that checks for an API key:

```
require_api_key = lambda do |request, response, next_handler|
  api_key = request.headers["X-API-Key"] || ""

  if api_key != "my-secret-key"
    return response.json({ error: "Invalid API key" }, 401)
  end

  next_handler.call(request, response)
end

Tina4::Router.get("/api/secret", middleware: "require_api_key") do |request, response|
  response.json({ secret: "The answer is 42" })
end
```

```
curl http://localhost:7147/api/secret
```

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

Status: `401 Unauthorized`.

```
curl http://localhost:7147/api/secret -H "X-API-Key: my-secret-key"
```

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

Middleware on a Group

Apply middleware to every route inside a group:

```
Tina4::Router.group("/api/admin", middleware: "require_auth") do

  Tina4::Router.get("/dashboard") do |request, response|
    response.json({ page: "admin dashboard" })
  end

  Tina4::Router.get("/users") do |request, response|
    response.json({ page: "user management" })
  end
```

```
end
```

Every route in the group now demands the `Authorization` header. No per-route repetition.

Multiple Middleware

Chain them with an array:

```
Tina4::Router.get("/api/important", middleware: ["log_request", "require_api_key", "require_auth
  response.json({ data: "important stuff" })
end
```

Middleware runs in order: `log_request` first, then `require_api_key`, then `require_auth`, then the route handler. If any middleware skips `next_handler`, the chain stops there.

---

7. Route Decorators: @noauth and @secured

Tina4 provides two decorators for controlling authentication at the route level.

@noauth -- Public Routes

When your application has global authentication middleware, `@noauth` marks specific routes as public:

```
# @noauth
Tina4::Router.get("/api/public/info") do |request, response|
  response.json({
    app: "My Store",
    version: "1.0.0"
  })
end
```

The `@noauth` comment tells Tina4 to skip authentication for this route, even if global auth middleware guards the parent group.

@secured -- Protected GET Routes

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

```
# @secured
Tina4::Router.get("/api/profile") do |request, response|
  # request.user is populated by the auth middleware
  response.json({ user: request.user })
end
```

By default, `POST`, `PUT`, `PATCH`, and `DELETE` routes are secured. `GET` routes are public unless you add `@secured`. Reading is open. Writing demands credentials.

---

8. Route Chaining: .secure and .cache

Routes 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:

```
Tina4::Router.get("/api/account") do |request, response|
  response.json({ account: request.user })
end.secure
```

```
curl http://localhost:7147/api/account
# 401 Unauthorized

curl http://localhost:7147/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:

```
Tina4::Router.get("/api/catalog") do |request, response|
  # Expensive database query
  response.json({ products: products })
end.cache
```

Chaining Both

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

```
Tina4::Router.get("/api/data") do |request, response|
  response.json({ data: data })
end.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 everything after it:

```
Tina4::Router.get("/docs/*") do |request, response|
  path = request.params["*"] || ""
  response.json({
    section: "docs",
    path: path
  })
end
```

```
curl http://localhost:7147/docs/getting-started
```

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

```
curl http://localhost:7147/docs/api/authentication/jwt
```

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

Catch-All Route (Custom 404)

Handle any unmatched URL:

```
Tina4::Router.get("/*") do |request, response|
  response.json({
    error: "Page not found",
    path: request.path
  }, 404)
end
```

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.

Or create a custom 404 page 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 unmatched routes when the file exists.

---

10. Route Listing via CLI

As your application grows, you need to 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/admin/dashboard          require_auth        public
GET      /api/admin/users              require_auth        public
GET      /api/public/info              -                   @noauth
GET      /api/profile                  -                   @secured
GET      /search                       -                   public
GET      /docs/*                       -                   public
```

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
GET      /api/admin/users             require_auth       public
```

---

11. Organizing Route Files

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

Pattern 1: One File Per Resource

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

Pattern 2: Subdirectories by Feature

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

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

---

12. Exercise: Build a Full CRUD API for Products

Build a complete REST API for managing products. Data lives in a Ruby array (no database yet -- Chapter 5 handles that).

Requirements

Create `src/routes/product_api.rb` with these 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).

Seed data:

```
products = [
  { id: 1, name: "Wireless Keyboard", category: "Electronics", price: 79.99, in_stock: true },
  { 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:7147/api/products

# Filter by category
curl "http://localhost:7147/api/products?category=Fitness"

# Get one
curl http://localhost:7147/api/products/3

# Create
curl -X POST http://localhost:7147/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:7147/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:7147/api/products/3

# Not found
curl http://localhost:7147/api/products/999
```

---

13. Solution

Create `src/routes/product_api.rb`:

```
# In-memory product store (resets on server restart)
$products = [
  { id: 1, name: "Wireless Keyboard", category: "Electronics", price: 79.99, in_stock: true },
  { 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
Tina4::Router.get("/api/products") do |request, response|
  category = request.params["category"]

  if category
    filtered = $products.select { |p| p[:category].downcase == category.downcase }
    response.json({ products: filtered, count: filtered.length })
  else
    response.json({ products: $products, count: $products.length })
  end
end

# Get a single product by ID
Tina4::Router.get("/api/products/{id:int}") do |request, response|
  id = request.params[:id]

  product = $products.find { |p| p[:id] == id }

  if product
    response.json(product)
  else
    response.json({ error: "Product not found", id: id }, 404)
  end
end

# Create a new product
Tina4::Router.post("/api/products") do |request, response|
  body = request.body

  if body["name"].nil? || body["name"].empty?
    return response.json({ error: "Name is required" }, 400)
  end

  product = {
    id: $next_id,
    name: body["name"],
    category: body["category"] || "Uncategorized",
    price: (body["price"] || 0).to_f,
    in_stock: body["in_stock"] != false
  }

  $next_id += 1
  $products << product
```

```
  response.json(product, 201)
end

# Replace a product
Tina4::Router.put("/api/products/{id:int}") do |request, response|
  id = request.params[:id]
  body = request.body

  index = $products.index { |p| p[:id] == id }

  if index.nil?
    response.json({ error: "Product not found", id: id }, 404)
  else
    $products[index] = {
      id: id,
      name: body["name"] || $products[index][:name],
      category: body["category"] || $products[index][:category],
      price: (body["price"] || $products[index][:price]).to_f,
      in_stock: body.key?("in_stock") ? body["in_stock"] : $products[index][:in_stock]
    }
    response.json($products[index])
  end
end

# Delete a product
Tina4::Router.delete("/api/products/{id:int}") do |request, response|
  id = request.params[:id]

  index = $products.index { |p| p[:id] == id }

  if index.nil?
    response.json({ error: "Product not found", id: id }, 404)
  else
    $products.delete_at(index)
    response.json(nil, 204)
  end
end
```

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 404.

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

Fix: Pick one convention. Stick with it. Set `TINA4_TRAILING_SLASH_REDIRECT=true` in `.env` and Tina4 redirects `/products/` to `/products`.

2. Parameter Names Must Be Unique in a Path

Problem: `/users/{id}/posts/{id}` behaves wrong -- both parameters share the same 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 `Tina4::Router.get("/items/{id}", ...)` and `Tina4::Router.get("/items/{action}", ...)` and the wrong handler runs.

Cause: Both patterns match `/items/42`. First registered wins.

Fix: Use typed parameters: `Tina4::Router.get("/items/{id:int}", ...)` matches integers only, leaving `/items/export` free. Or restructure: `/items/{id:int}` and `/items/actions/{action}`.

4. Route Handler Must Return a Response

Problem: Handler runs but the browser shows an empty page or 500 error.

Cause: No call to `response.json`, `response.render`, or another response method. Without a return value from the response object, Tina4 has nothing to send.

Fix: Every handler must end with `response.json(...)`, `response.html(...)`, or `response.render(...)`.

5. Block Syntax Matters

Problem: Route handler raises a syntax error about unexpected blocks.

Cause: Ruby blocks with `do...end` and `{...}` have different precedence.

Fix: Use `do |request, response| ... end` for route blocks. The curly brace form works for single-line handlers but causes parsing issues with method arguments.

6. Middleware Must Be a Named Function or String

Problem: Passing an inline lambda as middleware causes unexpected behavior.

Cause: Tina4 expects middleware referenced by name (a string), resolved at runtime.

Fix: Define middleware as a named method or lambda. Pass the name as a string: `"my_middleware"`.

7. Group Prefix Must Start with a Slash

Problem: `Tina4::Router.group("api/v1")` produces routes that do not match.

Cause: The group prefix needs a leading `/`.

Fix: Start group prefixes with `/`: `Tina4::Router.group("/api/v1")`.

Request & 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 shows 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`.

```
Tina4::Router.get("/echo") do |request, response|
  response.json({
    method: request.method,
    path: request.path,
    your_ip: request.ip
  })
end
```

```
curl http://localhost:7147/echo
```

```
{"method":"GET","path":"/echo","your_ip":"127.0.0.1"}
```

The pattern holds for every route: inspect the request, build the response, return it.

---

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.

method

The HTTP method arrives as an uppercase string:

```
Tina4::Router.get("/api/check") do |request, response|
  response.json({ method: request.method })
end
```

```
curl http://localhost:7147/api/check
```

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

path

The URL path strips query parameters:

```
# Request to /api/users?page=2
request.path # "/api/users"
```

params

Path parameters from the URL pattern (see Chapter 2). The parameter names match the `{name}` placeholders in the route:

```
# Route: /users/{id}/posts/{post_id}
# Request: /users/5/posts/42
request.params["id"]      # "5" (or 5 if typed as {id:int})
request.params["post_id"] # "42"
```

query

Query string parameters live in the same `params` hash:

```
# Request: /search?q=keyboard&page=2&sort=price
request.params["q"]    # "keyboard"
request.params["page"] # "2"
request.params["sort"] # "price"
```

body

The parsed request body. JSON requests become a hash. Form submissions contain the form fields. GET requests produce `nil`:

```
# POST with {"name": "Widget", "price": 9.99}
request.body["name"]  # "Widget"
request.body["price"] # 9.99
```

headers

Request headers as a hash. Header names keep their original casing:

```
request.headers["Content-Type"]  # "application/json"
request.headers["Authorization"] # "Bearer eyJhbGci..."
request.headers["X-Custom"]      # "my-value"
```

ip

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

```
request.ip # "127.0.0.1"
```

cookies

A hash of cookies the client sent:

```
request.cookies["session_id"]  # "abc123"
request.cookies["preferences"] # "dark-mode"
```

files

Uploaded files (section 7 covers the details):

```
request.files["avatar"] # File object with name, type, size, tmp_path
```

Inspecting the Full Request

A route that dumps everything:

```
Tina4::Router.post("/debug/request") do |request, response|
  response.json({
    method: request.method,
    path: request.path,
    params: request.params,
    query: request.params,
```

```
    body: request.body,
    headers: request.headers,
    ip: request.ip,
    cookies: request.cookies
  })
end
```

```
curl -X POST "http://localhost:7147/debug/request?page=1" \
  -H "Content-Type: application/json" \
  -H "X-Custom: hello" \
  -d '{"name": "test"}'
```

```
{
  "method": "POST",
  "path": "/debug/request",
  "params": {},
  "query": {"page": "1"},
  "body": {"name": "test"},
  "headers": {
    "Content-Type": "application/json",
    "X-Custom": "hello",
    "Host": "localhost:7147",
    "User-Agent": "curl/8.4.0",
    "Accept": "*/*",
    "Content-Length": "16"
  },
  "ip": "127.0.0.1",
  "cookies": {}
}
```

---

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.

json -- JSON Response

The workhorse for APIs. Pass any hash or value and it becomes JSON:

```
response.json({ name: "Alice", age: 30 })
```

```
{"name":"Alice","age":30}
```

Pass a status code as the second argument:

```
response.json({ id: 7, name: "Widget" }, 201)
```

Returns `201 Created` with the JSON body.

html -- Raw HTML Response

Return an HTML string:

```
response.html("<h1>Hello</h1><p>This is HTML.</p>")
```

Sets `Content-Type: text/html; charset=utf-8`.

text -- Plain Text Response

Return plain text:

```
response.text("Just a plain string.")
```

Sets `Content-Type: text/plain; charset=utf-8`.

render -- Template Response

Render a Frond template with data (Chapter 4: Templates goes deep):

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

Tina4 finds the template in `src/templates/`, renders it, returns the HTML.

redirect -- Redirect Response

Send the client elsewhere:

```
response.redirect("/login")
```

Sends a `302 Found` by default. Pass a different status for permanent redirects:

```
response.redirect("/new-location", 301)
```

file -- File Download Response

Send a file for download:

```
response.file("/path/to/report.pdf")
```

Tina4 sets the right `Content-Type` from the file extension and adds `Content-Disposition` so the browser downloads instead of displaying.

Custom filename:

```
response.file("/path/to/report.pdf", "monthly-report-march-2026.pdf")
```

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

xml -- XML Response

Return XML content:

```
Tina4::Router.get("/api/feed") do |request, response|
  response.xml("<feed><entry><title>Hello</title></entry></feed>")
end
```

Sets `Content-Type: application/xml; charset=utf-8`.

error -- Error Envelope

Return a structured error response:

```
Tina4::Router.post("/api/things") do |request, response|
  response.error("VALIDATION_FAILED", "Name is required", 400)
end
```

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. Use this pattern across your API for consistent error handling.

---

4. Status Codes

Every response method accepts a status code. The ones 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 permanently changed.
`302`	Found	Temporary redirect.
`400`	Bad Request	Invalid input from the client.
`401`	Unauthorized	Missing or invalid authentication.
`403`	Forbidden	Authenticated but not allowed.
`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.

Set the status with chaining:

```
response.status(201).json({ id: 7, created: true })
```

Equivalent to `response.json({ id: 7, created: true }, 201)`. Some developers prefer the chained form.

---

5. Custom Headers

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

```
Tina4::Router.get("/api/data") do |request, response|
  response
    .header("X-Request-Id", SecureRandom.hex(8))
    .header("X-Rate-Limit-Remaining", "57")
    .header("Cache-Control", "no-cache")
    .json({ data: [1, 2, 3] })
end
```

```
curl -v http://localhost:7147/api/data 2>&1 | grep "< X-"
```

```
< X-Request-Id: 65f3a7b8c1234567
< X-Rate-Limit-Remaining: 57
```

Convention: `Title-Case` for custom headers, prefixed with `X-`.

CORS Headers

Tina4 handles CORS based on the `CORS_ORIGINS` setting in `.env`. The default `*` allows all origins. For production, restrict:

```
CORS_ORIGINS=https://myapp.com,https://admin.myapp.com
```

Manual override when needed:

```
response
  .header("Access-Control-Allow-Origin", "https://myapp.com")
  .header("Access-Control-Allow-Methods", "GET, POST, PUT, DELETE")
  .header("Access-Control-Allow-Headers", "Content-Type, Authorization")
  .json({ data: "value" })
```

---

6. Cookies

Set cookies on the response:

```
Tina4::Router.post("/login") do |request, response|
  # After validating credentials...
  response
    .cookie("session_id", "abc123xyz", {
      http_only: true,
      secure: true,
      same_site: "Strict",
      max_age: 3600,       # 1 hour in seconds
      path: "/"
    })
```

```
    .json({ message: "Logged in" })
end
```

Cookie options:

Option	Type	Default	Description
`http_only`	bool	`false`	JavaScript cannot access it
`secure`	bool	`false`	Only sent over HTTPS
`same_site`	string	`"Lax"`	`"Strict"`, `"Lax"`, or `"None"`
`max_age`	int	session	Lifetime in seconds
`path`	string	`"/"`	URL path scope
`domain`	string	current	Domain scope

Read cookies from the request:

```
Tina4::Router.get("/profile") do |request, response|
  session_id = request.cookies["session_id"]

  if session_id.nil?
    return response.json({ error: "Not logged in" }, 401)
  end

  response.json({ session: session_id })
end
```

Delete a cookie by setting `max_age` to `0`:

```
response
  .cookie("session_id", "", { max_age: 0, path: "/" })
  .json({ message: "Logged out" })
```

---

7. File Uploads

Uploaded files live in `request.files`. Each file object holds metadata and a temporary path.

Handling a Single File Upload

```
Tina4::Router.post("/api/upload") do |request, response|
  if request.files["image"].nil?
    return response.json({ error: "No file uploaded" }, 400)
  end

  file = request.files["image"]

  response.json({
    name: file.name,        # "photo.jpg"
    type: file.type,        # "image/jpeg"
    size: file.size,        # 245760 (bytes)
    tmp_path: file.tmp_path # Temporary file location
  })
end
```

Test with curl:

```
curl -X POST http://localhost:7147/api/upload \
  -F "image=@/path/to/photo.jpg"
```

```
{
  "name": "photo.jpg",
  "type": "image/jpeg",
  "size": 245760,
  "tmp_path": "/tmp/tina4_upload_abc123"
}
```

Saving the Uploaded File

The uploaded file sits in a temporary location. Move it to a permanent path:

```
Tina4::Router.post("/api/upload") do |request, response|
  if request.files["image"].nil?
    return response.json({ error: "No file uploaded" }, 400)
  end

  file = request.files["image"]

  # Validate file type
  allowed_types = ["image/jpeg", "image/png", "image/gif", "image/webp"]
  unless allowed_types.include?(file.type)
    return response.json({ error: "Invalid file type. Allowed: JPEG, PNG, GIF, WebP" }, 400)
  end

  # Validate file size (max 5MB)
  max_size = 5 * 1024 * 1024
  if file.size > max_size
    return response.json({ error: "File too large. Maximum size: 5MB" }, 400)
  end

  # Generate a unique filename
  extension = File.extname(file.name)
  filename = "img_#{SecureRandom.hex(8)}#{extension}"
  upload_dir = File.join(__dir__, "../../public/uploads")
  destination = File.join(upload_dir, filename)

  # Ensure the uploads directory exists
  FileUtils.mkdir_p(upload_dir) unless Dir.exist?(upload_dir)
```

```
  # Move the file
  FileUtils.mv(file.tmp_path, destination)

  response.json({
    message: "File uploaded successfully",
    filename: filename,
    url: "/uploads/#{filename}",
    size: file.size
  }, 201)
end
```

```
curl -X POST http://localhost:7147/api/upload \
  -F "image=@/path/to/photo.jpg"
```

```
{
  "message": "File uploaded successfully",
  "filename": "img_a1b2c3d4e5f6a7b8.jpg",
  "url": "/uploads/img_a1b2c3d4e5f6a7b8.jpg",
  "size": 245760
}
```

The file now lives at `http://localhost:7147/uploads/img_a1b2c3d4e5f6a7b8.jpg`.

Handling Multiple Files

When the HTML form uses `multiple` or you have multiple file inputs:

```
Tina4::Router.post("/api/upload-many") do |request, response|
  results = []

  request.files.each do |key, file|
    extension = File.extname(file.name)
    filename = "file_#{SecureRandom.hex(8)}#{extension}"
    upload_dir = File.join(__dir__, "../../public/uploads")
    destination = File.join(upload_dir, filename)

    FileUtils.mkdir_p(upload_dir) unless Dir.exist?(upload_dir)
    FileUtils.mv(file.tmp_path, destination)

    results << {
      original_name: file.name,
      saved_as: filename,
      url: "/uploads/#{filename}"
    }
  end

  response.json({ uploaded: results, count: results.length }, 201)
end
```

---

8. File Downloads

Send files with `response.file`:

```
Tina4::Router.get("/api/reports/{filename}") do |request, response|
  filename = request.params["filename"]
  filepath = File.join(__dir__, "../../data/reports", filename)

  unless File.exist?(filepath)
```

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

  response.file(filepath)
end
```

The browser downloads the file. Tina4 detects the MIME type from the extension and sets the right headers.

Force a specific download filename:

```
response.file(filepath, "Q1-2026-Sales-Report.pdf")
```

---

9. Content Negotiation

The same endpoint can return different formats based on what the client asks for. Check the `Accept` header:

```
Tina4::Router.get("/api/products/{id:int}") do |request, response|
  id = request.params["id"]
  product = {
    id: id,
    name: "Wireless Keyboard",
    price: 79.99
  }

  accept = request.headers["Accept"] || "application/json"

  if accept.include?("text/html")
    response.render("product-detail.html", { product: product })
  elsif accept.include?("text/plain")
    text = "Product ##{id}: #{product[:name]} - $#{product[:price]}"
    response.text(text)
  else
    # Default to JSON
    response.json(product)
  end
end
```

```
# JSON (default)
curl http://localhost:7147/api/products/1
```

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

```
# Plain text
curl http://localhost:7147/api/products/1 -H "Accept: text/plain"
```

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

```
# HTML (renders the template)
curl http://localhost:7147/api/products/1 -H "Accept: text/html"
```

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

---

10. Input Validation

Tina4 ships a `Validator` class for declarative input validation. Chain rules together, then check the result. If validation fails, `response.error` returns a structured error envelope (see section 3).

The Validator Class

```
Tina4::Router.post "/api/users" do |request, response|
  v = Tina4::Validator.new(request.body)
  v.required("name").required("email").email("email").min_length("name", 2)

  unless v.valid?
    return response.error("VALIDATION_FAILED", v.errors.first[:message], 400)
  end

  # proceed with valid data
end
```

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

Method	Description
`required(field)`	Field 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
`numeric(field)`	Field must be a number
`in_list(field, values)`	Field must be one of the allowed values

Call `v.valid?` to check all rules. Call `v.errors` to get the list of failures, each with a `:field` and `:message` key.

The Error Response Envelope

`response.error` returns a consistent JSON error envelope:

```
response.error("VALIDATION_FAILED", "Name is required", 400)
```

This produces:

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

The three arguments are: an error code string, a human-readable message, and the HTTP status code. Use this pattern across your API for consistent error handling.

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 returns a `413 Payload Too Large` response before your handler runs. The check fires before body parsing begins. Your route code never executes.

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.

---

11. Exercise: Build an Image Upload API

Build an API that handles image uploads and serves them back.

Requirements

Method	Path	Description
`POST`	`/api/images`	Upload an image. Validate type and size. Return the image URL.
`GET`	`/api/images/{filename}`	Return the uploaded image file. Return 404 if not found.

Rules:

- Accept JPEG, PNG, and WebP only
- Maximum file size: 2MB
- Save files to `src/public/uploads/` with a unique filename
- Return original filename, saved filename, file size in KB, and the URL
- The GET endpoint serves the file (not JSON)

Test with:

```
# Upload
curl -X POST http://localhost:7147/api/images \
  -F "image=@/path/to/photo.jpg"

# Download
curl http://localhost:7147/api/images/img_a1b2c3d4e5f6a7b8.jpg --output downloaded.jpg
```

---

12. Solution

Create `src/routes/images.rb`:

```
require "securerandom"
require "fileutils"

Tina4::Router.post("/api/images") do |request, response|
  # Check if a file was uploaded
  if request.files["image"].nil?
    return response.json({ error: "No image file provided. Use field name 'image'." }, 400)
  end

  file = request.files["image"]

  # Validate file type
  allowed_types = ["image/jpeg", "image/png", "image/webp"]
  unless allowed_types.include?(file.type)
    return response.json({
      error: "Invalid file type",
      received: file.type,
      allowed: allowed_types
    }, 400)
  end

  # Validate file size (max 2MB)
  max_size = 2 * 1024 * 1024
  if file.size > max_size
    return response.json({
      error: "File too large",
      size_bytes: file.size,
      max_bytes: max_size
    }, 400)
  end

  # Generate unique filename preserving extension
  extension = File.extname(file.name).downcase
  saved_name = "img_#{SecureRandom.hex(8)}#{extension}"
  upload_dir = File.join(__dir__, "../../public/uploads")
  destination = File.join(upload_dir, saved_name)

  # Create uploads directory if it does not exist
  FileUtils.mkdir_p(upload_dir) unless Dir.exist?(upload_dir)

  # Move the uploaded file
  FileUtils.mv(file.tmp_path, destination)

  response.json({
    message: "Image uploaded successfully",
    original_name: file.name,
    saved_name: saved_name,
    size_kb: (file.size / 1024.0).round(1),
    type: file.type,
    url: "/uploads/#{saved_name}"
  }, 201)
end

Tina4::Router.get("/api/images/{filename}") do |request, response|
  filename = request.params["filename"]

  # Prevent directory traversal
  if filename.include?("..") || filename.include?("/")
    return response.json({ error: "Invalid filename" }, 400)
```

```
  end

  filepath = File.join(__dir__, "../../public/uploads", filename)

  unless File.exist?(filepath)
    return response.json({ error: "Image not found", filename: filename }, 404)
  end

  response.file(filepath)
end
```

Expected output for upload:

```
{
  "message": "Image uploaded successfully",
  "original_name": "photo.jpg",
  "saved_name": "img_a1b2c3d4e5f6a7b8.jpg",
  "size_kb": 240.0,
  "type": "image/jpeg",
  "url": "/uploads/img_a1b2c3d4e5f6a7b8.jpg"
}
```

(Status: `201 Created`)

Expected output for invalid type:

```
{
  "error": "Invalid file type",
  "received": "application/pdf",
  "allowed": ["image/jpeg", "image/png", "image/webp"]
}
```

(Status: `400 Bad Request`)

Expected output for file too large:

```
{
  "error": "File too large",
  "size_bytes": 5242880,
  "max_bytes": 2097152
}
```

(Status: `400 Bad Request`)

The GET endpoint returns the raw image file with the correct `Content-Type` header. The browser displays it. Curl with `--output` saves it to disk.

---

13. Gotchas

1. Forgetting the response method

Problem: Handler runs (log output confirms it) but the browser gets an empty response or 500 error.

Cause: No call to `response.json`, `response.html`, or another response method. Ruby returns the last expression in a block, but if it is not a response, Tina4 has nothing to send.

Fix: End every handler with a response method call.

2. request.body is nil for GET requests

Problem: Accessing `request.body` in a GET handler returns `nil`.

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.

3. Body is nil for JSON requests

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

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

Fix: Include `-H "Content-Type: application/json"` with curl. In frontend JavaScript, `fetch()` with `JSON.stringify()` needs `headers: {"Content-Type": "application/json"}`.

4. Content-Type Mismatch

Problem: `response.json` sends HTML, or `response.html` sends plain text.

Cause: A middleware or error handler is overwriting the response. Or you used `puts` instead of a response method.

Fix: Use `response.json(...)`, `response.html(...)`, or another response method. `puts` bypasses the response object entirely.

5. File Uploads Return Empty

Problem: `request.files` is empty despite uploading a file.

Cause: The form lacks `enctype="multipart/form-data"`, or curl uses `-d` instead of `-F`.

Fix: HTML forms need `<form enctype="multipart/form-data">`. Curl needs `-F "field=@file.jpg"` (with `@`), not `-d`.

6. Redirect Loops

Problem: The browser shows "too many redirects" or hangs.

Cause: A route redirects to another route, which redirects back. Example: `/login` redirects to `/dashboard`, and `/dashboard` redirects to `/login` because the user is not authenticated.

Fix: Trace the redirect chain with the browser's network inspector. Make sure auth checks do not redirect authenticated users away from pages they can access.

7. Cookie Not Set

Problem: `response.cookie(...)` was called but the browser shows no cookie.

Cause: `secure: true` means the cookie travels over HTTPS only. Local development uses `http://localhost`, so the cookie is dropped.

Fix: Set `secure: false` during development, or use `secure: ENV["TINA4_DEBUG"] != "true"` to auto-switch.

8. Large Request Body Rejected

Problem: POST requests with large bodies return 413.

Cause: The body exceeds the configured maximum size.

Fix: Increase `TINA4_MAX_UPLOAD_SIZE` in `.env`. Default is `10mb`. For file uploads, you may need `50mb` or more:

```
TINA4_MAX_UPLOAD_SIZE=50mb
```

9. 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
response
  .header("X-Custom", "value")
  .cookie("token", "abc", { http_only: true })
  .json({ ok: true })

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

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.

Templates live in `src/templates/`. When you call `response.render("page.html", data)`, Frond loads `src/templates/page.html`, processes the tags and expressions, and returns the final HTML.

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. Variables and Expressions

Output a variable with double curly braces:

```
<h1>Hello, {{ name }}!</h1>
```

Route handler:

```
Tina4::Router.get("/welcome") do |request, response|
  response.render("welcome.html", { name: "Alice" })
end
```

Create `src/templates/welcome.html`:

```
<!DOCTYPE html>
<html>
<head><title>Welcome</title></head>
<body>
    <h1>Hello, {{ name }}!</h1>
</body>
</html>
```

Browser output:

```
Hello, Alice!
```

Accessing Nested Data

Dot notation reaches into nested hashes:

```
data = {
  user: {
    name: "Alice",
    email: "alice@example.com",
    address: {
```

```
      city: "Cape Town",
      country: "South Africa"
    }
  }
}

response.render("profile.html", data)
```

```
<p>{{ user.name }} lives in {{ user.address.city }}, {{ user.address.country }}.</p>
```

Output:

```
Alice lives in Cape Town, South Africa.
```

Method Calls on Values

If a Hash value is a `Proc` or `lambda`, you can call it directly in a template expression:

```
data = {
  user: {
    name: "Alice",
    t: ->(key) { I18n.t(key) },
    greet: ->(greeting) { "#{greeting}, Alice!" }
  }
}

response.render("page.twig", data)
```

```
<p>{{ user.t("welcome_message") }}</p>
<p>{{ user.greet("Hello") }}</p>
```

This works for any callable value -- `Proc`, `lambda`, or `method` objects. Arguments are parsed from the parenthesized expression.

Expressions

Basic expressions work inside `{{ }}`:

```
<p>Total: ${{ price * quantity }}</p>
<p>Discounted: ${{ price * 0.9 }}</p>
<p>Full name: {{ first_name ~ " " ~ last_name }}</p>
```

The `~` operator concatenates strings.

---

3. Template Inheritance

Template inheritance is the headline feature. Define a base layout once. Extend it in every page.

Base Layout

Create `src/templates/base.twig`:

```
<!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 App{% endblock %}</title>
```

```
    <link rel="stylesheet" href="/css/tina4.css">
    {% block head %}{% endblock %}
</head>
<body>
    <nav>
        <a href="/">Home</a>
        <a href="/about">About</a>
        <a href="/contact">Contact</a>
    </nav>

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

    <footer>
        <p>&copy; 2026 My App. All rights reserved.</p>
    </footer>

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

Four blocks: `title`, `head`, `content`, `scripts`. Child templates override only the blocks they need.

Child Template

Create `src/templates/about.twig`:

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

{% block title %}About Us{% endblock %}

{% block content %}
    <h1>About Us</h1>
    <p>We have been building things since {{ founded_year }}.</p>
    <p>Our team has {{ team_size }} members across {{ office_count }} offices.</p>
{% endblock %}
```

Route handler:

```
Tina4::Router.get("/about") do |request, response|
  response.render("about.twig", {
    founded_year: 2020,
    team_size: 12,
    office_count: 3
  })
end
```

Browser output: A full HTML page with the nav, the "About Us" content, and the footer. The `<title>` tag reads "About Us". The `head` and `scripts` blocks stay empty -- the child did not override them.

Using `{{ parent() }}`

Add to a block instead of replacing it:

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

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

{% block content %}
    <h1>Contact Us</h1>
    <form>...</form>
{% endblock %}
```

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

---

4. Includes

Break templates into reusable pieces with `{% include %}`:

Create `src/templates/partials/header.twig`:

```
<header>
    <div class="logo">{{ site_name | default("My App") }}</div>
    <nav>
        <a href="/">Home</a>
        <a href="/products">Products</a>
        <a href="/contact">Contact</a>
    </nav>
</header>
```

Create `src/templates/partials/product-card.twig`:

```
<div class="product-card{{ product.featured ? ' featured' : '' }}">
    <h3>{{ product.name }}</h3>
    <p class="price">${{ product.price | number_format(2) }}</p>
    {% if product.in_stock %}
        <span class="badge-success">In Stock</span>
    {% else %}
        <span class="badge-danger">Out of Stock</span>
    {% endif %}
</div>
```

Use them in a page:

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

{% block content %}
    {% include "partials/header.twig" %}

    <h1>Products</h1>
    {% for product in products %}
        {% include "partials/product-card.twig" %}
    {% endfor %}
{% endblock %}
```

The `product` variable is available inside the included template because it exists in the current scope (the for loop).

Passing Variables to Includes

Pass specific variables with `with`:

```
{% include "partials/header.twig" with {"site_name": "Cool Store"} %}
```

Use `only` to isolate the included template from the parent scope:

```
{% include "partials/header.twig" with {"site_name": "Cool Store"} only %}
```

With `only`, the included template sees `site_name` and nothing else.

---

5. For Loops

Loop through arrays with `{% for %}`:

```
<ul>
{% for item in items %}
    <li>{{ item }}</li>
{% endfor %}
</ul>
```

The `loop` Variable

Inside a for loop, Frond provides a special `loop` variable:

Property	Type	Description
`loop.index`	int	Current iteration (1-based)
`loop.index0`	int	Current iteration (0-based)
`loop.first`	bool	True on the first iteration
`loop.last`	bool	True on the last iteration
`loop.length`	int	Total number of items
`loop.revindex`	int	Iterations remaining (1-based)

```
<table>
    <thead>
        <tr><th>#</th><th>Name</th><th>Price</th></tr>
    </thead>
    <tbody>
    {% for product in products %}
        <tr class="{{ loop.index is odd ? 'row-light' : 'row-dark' }}">
            <td>{{ loop.index }}</td>
            <td>{{ product.name }}</td>
            <td>${{ product.price | number_format(2) }}</td>
        </tr>
    {% endfor %}
    </tbody>
</table>
```

Empty Lists

Handle empty lists with `{% else %}`:

```
{% for product in products %}
    <div class="product-card">
        <h3>{{ product.name }}</h3>
    </div>
{% else %}
    <p>No products found.</p>
{% endfor %}
```

If `products` is empty or undefined, the `else` block renders instead.

Looping Over Key-Value Pairs

```
{% for key, value in metadata %}
    <dt>{{ key }}</dt>
    <dd>{{ value }}</dd>
{% endfor %}
```

---

6. Conditionals

if / elseif / else

```
{% if user.role == "admin" %}
    <a href="/admin">Admin Panel</a>
{% elseif user.role == "editor" %}
    <a href="/editor">Editor Dashboard</a>
{% else %}
    <a href="/profile">My Profile</a>
{% endif %}
```

Ternary Operator

Inline conditionals:

```
<span class="{{ is_active ? 'text-green' : 'text-gray' }}">
    {{ is_active ? 'Active' : 'Inactive' }}
</span>
```

Testing for Existence

Use `is defined` to check if a variable exists:

```
{% if error_message is defined %}
    <div class="alert alert-danger">{{ error_message }}</div>
{% endif %}
```

Truthiness

These values are false: `false`, `nil`, `0`, `""` (empty string), `[]` (empty array). Everything else is true.

```
{% if items %}
    <p>{{ items | length }} items found.</p>
{% else %}
    <p>No items.</p>
```

```
{% endif %}
```

{% 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>
```

---

7. Filters

Filters transform values. Apply them with the `|` (pipe) character:

```
{{ name | upper }}
```

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 just the raw JWT token string	
`to_json`	`{{ data | to_json }}`	JSON-encode a value (safe, 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" #}
```

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 `inspect` of the value. Handles hashes, arrays, class instances, and cyclic references — Ruby's `inspect` prints `{...}` for back-edges.

```
{{ dump(order) }}

{# Output: #}
{# <pre>#&lt;Order:0x00007f8 @id=42, @items=[...], @total=99.99&gt;</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 `SafeString`. 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`.

---

8. Macros

Macros are reusable template functions. Components you define once and call many times.

Defining a Macro

Create `src/templates/macros.twig`:

```
{% macro button(text, url, style) %}
    <a href="{{ url | default('#') }}" class="btn btn-{{ style | default('primary') }}">
        {{ text }}
    </a>
{% endmacro %}

{% macro alert(message, type) %}
    <div class="alert alert-{{ type | default('info') }}">
        {{ message }}
    </div>
{% endmacro %}

{% macro input(name, label, type, value) %}
    <div class="form-group">
        <label for="{{ name }}">{{ label | default(name | capitalize) }}</label>
        <input type="{{ type | default('text') }}" id="{{ name }}" name="{{ name }}" value="{{ v
    </div>
{% endmacro %}
```

Using Macros

Import macros in your template:

```
{% from "macros.twig" import button, alert, input %}

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

```
{% block content %}
    {{ alert("Your profile has been updated.", "success") }}

    <form method="POST" action="/profile">
        {{ input("name", "Full Name", "text", user.name) }}
        {{ input("email", "Email Address", "email", user.email) }}
        {{ input("phone", "Phone Number", "tel", user.phone) }}

        {{ button("Save Changes", "", "primary") }}
        {{ button("Cancel", "/dashboard", "secondary") }}
    </form>
{% endblock %}
```

Expected output (simplified):

```
<div class="alert alert-success">
    Your profile has been updated.
</div>

<form method="POST" action="/profile">
    <div class="form-group">
        <label for="name">Full Name</label>
        <input type="text" id="name" name="name" value="Alice">
    </div>
    <div class="form-group">
        <label for="email">Email Address</label>
        <input type="email" id="email" name="email" value="alice@example.com">
    </div>
    <div class="form-group">
        <label for="phone">Phone Number</label>
        <input type="tel" id="phone" name="phone" value="">
    </div>

    <a href="#" class="btn btn-primary">Save Changes</a>
    <a href="/dashboard" class="btn btn-secondary">Cancel</a>
</form>
```

---

9. Special Tags

{% raw %} -- Literal Output

When you need to output literal `{{ }}` or `{% %}` (for Vue.js or Angular templates):

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

Outputs the literal text `{{ message }}` without processing it as a Frond expression.

{% spaceless %} -- Remove Whitespace

Strip whitespace between HTML tags:

```
{% spaceless %}
    <div>
```

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

Output:

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

Useful for inline elements where whitespace creates unwanted gaps.

{% autoescape %} -- Control Escaping

Override auto-escaping for a block:

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

Everything inside outputs without HTML escaping. Equivalent to `| raw` on every variable, but more convenient for large blocks of trusted content.

Comments

Template comments stay invisible in the output:

```
{# This comment will not appear in the HTML output #}

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

---

10. tina4css Integration

Every Tina4 project includes `tina4.css` -- a built-in CSS utility framework available at `/css/tina4.css`. Layout, typography, common UI patterns. No external dependencies.

Include it in your base template:

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

Layout Classes

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

Common Components

```
<!-- Buttons -->
<button class="btn btn-primary">Primary</button>
<button class="btn btn-secondary">Secondary</button>
<button class="btn btn-danger">Danger</button>

<!-- Cards -->
<div class="card">
    <div class="card-header">Title</div>
    <div class="card-body">Content here</div>
    <div class="card-footer">Footer</div>
</div>

<!-- Alerts -->
<div class="alert alert-success">Operation completed.</div>
<div class="alert alert-danger">Something went wrong.</div>
<div class="alert alert-warning">Please review your input.</div>

<!-- Forms -->
<div class="form-group">
    <label for="name">Name</label>
    <input type="text" id="name" class="form-control">
</div>
```

Utility Classes

```
<p class="text-center">Centered text</p>
<p class="text-right">Right-aligned text</p>
<div class="mt-4">Margin top</div>
<div class="p-3">Padding all around</div>
<span class="text-muted">Gray text</span>
<span class="text-primary">Primary color text</span>
```

No Bootstrap needed. No Tailwind needed. If you prefer those, swap the `<link>` tag. tina4css stays out of the way.

---

11. Exercise: Build a Product Catalog Page

Build a product catalog page with a base layout, product cards, category filters, and a reusable card macro.

Requirements

- Create a base layout at `src/templates/catalog-base.twig` with blocks for `title`, `content`, and `scripts`
- Create a macro file at `src/templates/catalog-macros.twig` with:
- A `productCard(product)` macro that renders a styled card with name, category, price, stock status, and optional featured badge
- A `categoryFilter(categories, active)` macro that renders filter buttons
- Create a page template at `src/templates/catalog.twig` that:
- Extends the base layout

- Uses the macros
- Shows a heading with total product count
- Shows category filter buttons (All, and one per unique category)
- Shows product cards in a grid
- Shows featured products with a distinct style
- Handles empty filter results
- Create a route at `GET /catalog` that accepts an optional `?category=` filter

Data

Use this product list in your route handler:

```
products = [
  { name: "Espresso Machine", category: "Kitchen", price: 299.99, in_stock: true, featured: true
  { name: "Yoga Mat", category: "Fitness", price: 29.99, in_stock: true, featured: false },
  { name: "Standing Desk", category: "Office", price: 549.99, in_stock: true, featured: true },
  { name: "Blender", category: "Kitchen", price: 89.99, in_stock: false, featured: false },
  { name: "Running Shoes", category: "Fitness", price: 119.99, in_stock: true, featured: false }
  { name: "Desk Lamp", category: "Office", price: 39.99, in_stock: true, featured: true },
  { name: "Cast Iron Skillet", category: "Kitchen", price: 44.99, in_stock: true, featured: fals
]
```

Test with:

```
http://localhost:7147/catalog
http://localhost:7147/catalog?category=Kitchen
http://localhost:7147/catalog?category=Fitness
```

---

12. Solution

Create `src/templates/catalog-base.twig`:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>{% block title %}Product Catalog{% 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: 1000px; margin: 0 auto; padding: 20px; }
        .header { background: #2c3e50; color: white; padding: 20px; margin-bottom: 24px; }
        .header h1 { margin: 0; }
        .header p { margin: 4px 0 0; opacity: 0.8; }
        .filters { margin-bottom: 20px; }
        .filter-btn { display: inline-block; padding: 6px 14px; margin: 0 6px 6px 0; border-radi
        .filter-btn.active { background: #2c3e50; color: white; border-color: #2c3e50; }
        .product-grid { display: grid; grid-template-columns: repeat(auto-fill, minmax(280px, 1f
        .product-card { background: white; border: 2px solid #e9ecef; border-radius: 8px; paddin
        .product-card:hover { border-color: #adb5bd; }
        .product-card.featured { border-color: #f39c12; background: #fef9e7; }
        .product-name { font-size: 1.1em; font-weight: 600; margin: 0 0 4px; }
```

```
        .product-category { font-size: 0.85em; color: #6c757d; margin: 0 0 8px; }
        .product-price { font-size: 1.2em; font-weight: bold; color: #27ae60; }
        .badge { display: inline-block; padding: 2px 8px; border-radius: 4px; font-size: 0.75em;
        .badge-featured { background: #f39c12; color: white; }
        .badge-stock { background: #d4edda; color: #155724; }
        .badge-nostock { background: #f8d7da; color: #721c24; }
        .empty-state { text-align: center; padding: 40px; color: #6c757d; }
    </style>
</head>
<body>
    {% block content %}{% endblock %}

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

Create `src/templates/catalog-macros.twig`:

```
{% macro productCard(product) %}
    <div class="product-card{{ product.featured ? ' featured' : '' }}">
        <p class="product-name">
            {{ product.name }}
            {% if product.featured %}
                <span class="badge badge-featured">Featured</span>
            {% endif %}
        </p>
        <p class="product-category">{{ product.category }}</p>
        <p class="product-price">
            ${{ product.price | number_format(2) }}
            {% if product.in_stock %}
                <span class="badge badge-stock">In Stock</span>
            {% else %}
                <span class="badge badge-nostock">Out of Stock</span>
            {% endif %}
        </p>
    </div>
{% endmacro %}

{% macro categoryFilter(categories, active) %}
    <div class="filters">
        <a href="/catalog" class="filter-btn{{ active is not defined or active == '' ? ' active'
        {% for cat in categories %}
            <a href="/catalog?category={{ cat }}" class="filter-btn{{ active == cat ? ' active'
        {% endfor %}
    </div>
{% endmacro %}
```

Create `src/templates/catalog.twig`:

```
{% extends "catalog-base.twig" %}

{% from "catalog-macros.twig" import productCard, categoryFilter %}

{% block title %}{{ active_category | default("All") }} Products - Catalog{% endblock %}

{% block content %}
    <div class="header">
        <h1>Product Catalog</h1>
        <p>{{ products | length }} product{{ products | length != 1 ? 's' : '' }}{% if active_ca
```

```
    </div>

    <div class="container">
        {{ categoryFilter(categories, active_category) }}

        {% if products | length > 0 %}
            <div class="product-grid">
                {% for product in products %}
                    {{ productCard(product) }}
                {% endfor %}
            </div>
        {% else %}
            <div class="empty-state">
                <h2>No products found</h2>
                <p>Try a different category or <a href="/catalog">view all products</a>.</p>
            </div>
        {% endif %}
    </div>
{% endblock %}
```

Create `src/routes/catalog.rb`:

```
Tina4::Router.get("/catalog") do |request, response|
  all_products = [
    { name: "Espresso Machine", category: "Kitchen", price: 299.99, in_stock: true, featured: tr
    { name: "Yoga Mat", category: "Fitness", price: 29.99, in_stock: true, featured: false },
    { name: "Standing Desk", category: "Office", price: 549.99, in_stock: true, featured: true }
    { name: "Blender", category: "Kitchen", price: 89.99, in_stock: false, featured: false },
    { name: "Running Shoes", category: "Fitness", price: 119.99, in_stock: true, featured: false
    { name: "Desk Lamp", category: "Office", price: 39.99, in_stock: true, featured: true },
    { name: "Cast Iron Skillet", category: "Kitchen", price: 44.99, in_stock: true, featured: fa
  ]

  # Get unique categories
  categories = all_products.map { |p| p[:category] }.uniq.sort

  # Filter by category if specified
  active_category = request.params["category"] || ""
  products = if active_category.empty?
               all_products
             else
               all_products.select { |p| p[:category].downcase == active_category.downcase }
             end

  response.render("catalog.twig", {
    products: products,
    categories: categories,
    active_category: active_category
  })
end
```

Browser output for `/catalog`:

- A dark header with "Product Catalog" and "7 products"
- Filter buttons: All (active), Fitness, Kitchen, Office
- A grid of 7 product cards

- Three cards (Espresso Machine, Standing Desk, Desk Lamp) wear a gold border and "Featured" badge
- The Blender card wears a red "Out of Stock" badge

Browser output for `/catalog?category=Kitchen`:

- Header shows "3 products in Kitchen"
- Kitchen filter button is active
- Three cards: Espresso Machine, Blender, Cast Iron Skillet

---

13. Gotchas

1. `{% extends %}` Must Be the First Tag

Problem: Template inheritance does not work. The page renders without the base layout.

Cause: `{% extends "base.twig" %}` must be the first tag in the template. Any text, whitespace, or comment before it breaks inheritance.

Fix: Make `{% extends %}` the absolute first thing in the file. Move `{% from %}` imports after the extends tag.

2. Undefined Variables Show Nothing

Problem: `{{ username }}` renders as empty instead of raising an error.

Cause: Frond outputs nothing for undefined variables. By design. But it can hide bugs.

Fix: Use the `default` filter: `{{ username | default("Guest") }}`. Or check with `{% if username is defined %}`.

3. Auto-Escaping Prevents HTML Output

Problem: You pass HTML content but it appears as literal text in the page.

Cause: Auto-escaping converts `<` to `<` and `>` to `>` for security.

Fix: Trusted content gets `{{ content | raw }}`. Never use `raw` on user-supplied input.

4. Variable Scope in Includes

Problem: A variable defined inside a `{% for %}` loop vanishes after the loop ends.

Cause: Loop variables are scoped to the loop. They do not leak into the outer scope.

Fix: Use `{% set %}` before the loop and update inside it. Or restructure to keep logic within the loop.

5. Macro Arguments Are Positional

Problem: `{{ button("Click", style="danger") }}` does not work as expected.

Cause: Frond macros use positional arguments. Order matters.

Fix: Pass arguments in the order they are defined: `{{ button("Click", "/url", "danger") }}`. Many optional arguments? Consider passing a single object.

6. Template File Extension Does Not Matter

Problem: Should you use `.html`, `.twig`, or `.tpl`?

Cause: Frond does not care about file extensions. It processes any file in `src/templates/`.

Fix: Pick one and stay consistent. This book uses `.twig` for templates with Twig syntax and `.html` for simple HTML. Both work the same.

7. Filters Are Not Ruby Methods

Problem: `{{ items | count }}` or `{{ name | upcase }}` raises an error.

Cause: Frond filters follow Twig conventions, not Ruby conventions.

Fix: Use `{{ items | length }}` not `count`. Use `{{ name | upper }}` not `upcase`. Use `{{ text | lower }}` not `downcase`. See the filter table in section 7.

Database

1. From Arrays to Real Data

In Chapters 2 and 3, all your data lived in Ruby arrays. Server restart. Data gone. Fine for learning routing. Useless for production.

This chapter covers Tina4's database layer: raw queries, parameterised queries, transactions, schema inspection, helper methods, and migrations.

Tina4 speaks to five database engines: SQLite, PostgreSQL, MySQL, Microsoft SQL Server, and Firebird. The API is identical across all five. Switch databases by changing one line in `.env`.

---

2. Connecting to a Database

The Default: SQLite

When you scaffold with `tina4 init`, Tina4 creates a SQLite database at `data/app.db`. The default `.env` contains:

```
TINA4_DEBUG=true
```

No explicit `TINA4_DATABASE_URL`? Tina4 defaults to `sqlite:///data/app.db`. The health check at `/health` shows `"database": "connected"` with zero configuration.

Connection Strings for Other Databases

Set `TINA4_DATABASE_URL` in `.env` to use a different engine:

```
# SQLite (explicit)
TINA4_DATABASE_URL=sqlite:///data/app.db

# PostgreSQL
TINA4_DATABASE_URL=postgres://localhost:5432/myapp

# MySQL
TINA4_DATABASE_URL=mysql://localhost:3306/myapp

# Microsoft SQL Server
TINA4_DATABASE_URL=mssql://localhost:1433/myapp

# Firebird
TINA4_DATABASE_URL=firebird://localhost:3050/path/to/database.fdb
```

Firebird URL Forms

Firebird is the awkward one: 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 `URI.parse`, which is unintuitive.

Tina4 normalises five equivalent forms. Pick whichever reads best:

```
# Classic double-slash absolute path -- the URL spec way
TINA4_DATABASE_URL=firebird://SYSDBA:masterkey@localhost:3050//firebird/data/app.fdb

# Single-slash absolute path -- what most people instinctively type
TINA4_DATABASE_URL=firebird://SYSDBA:masterkey@localhost:3050/firebird/data/app.fdb

# Windows drive-letter path (also accepts /C%3A/Data/app.fdb)
TINA4_DATABASE_URL=firebird://SYSDBA:masterkey@host:3050/C:/Data/app.fdb

# Firebird alias (single token, no slashes)
TINA4_DATABASE_URL=firebird://SYSDBA:masterkey@localhost:3050/employee
```

For ops setups that keep the server URL and database location in separate config layers -- or for Windows backslash paths -- 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.

Separate Credentials

If you prefer to keep credentials out of the connection string (recommended for production), use separate environment variables:

```
TINA4_DATABASE_URL=postgres://localhost:5432/myapp
TINA4_DATABASE_USERNAME=myuser
TINA4_DATABASE_PASSWORD=secretpassword
```

Tina4 merges these with the connection string at startup. The credentials in the separate variables take precedence over any embedded in the URL.

Programmatic Connection

You can also create a database connection directly in Ruby code:

```
db = Tina4::Database.new("sqlite://app.db", username: nil, password: nil)
```

Connection Pooling

For applications that handle many concurrent requests, enable connection pooling with the `pool` parameter:

```
db = Tina4::Database.new("postgres://localhost/mydb", pool: 5)
```

The `pool` parameter controls how many database connections are maintained:

- `pool: 0` (the default) -- a single connection is used for all queries
- `pool: N` (where N > 0) -- N connections are created and rotated round-robin across queries

Pooled connections are thread-safe. Each query is dispatched to the next available connection in the pool. This eliminates contention when multiple route handlers query the database simultaneously.

Verifying the Connection

After updating `.env`, restart the server and check:

```
curl http://localhost:7147/health
```

```
{
  "status": "ok",
  "database": "connected",
  "uptime_seconds": 3,
  "version": "3.0.0",
  "framework": "tina4-ruby"
}
```

If the database is not reachable, you will see `"database": "disconnected"` with an error message.

---

3. Getting the Database Object

In your route handlers, access the database via the `Tina4::Database` class:

```
Tina4::Router.get("/api/test-db") do |request, response|
  db = Tina4.database

  result = db.fetch("SELECT 1 + 1 AS answer")

  response.json(result)
end
```

```
curl http://localhost:7147/api/test-db
```

```
{"answer": 2}
```

`Tina4.database` returns the active database connection. You call methods like `fetch`, `execute`, and `fetch_one` on this object.

---

4. Raw Queries

fetch -- Get Multiple Rows

```
db = Tina4.database

# Returns an array of hashes
products = db.fetch("SELECT * FROM products WHERE price > 50")
```

Each row is a hash with column names as keys:

```
# products looks like:
[
  { "id" => 1, "name" => "Keyboard", "price" => 79.99 },
  { "id" => 4, "name" => "Standing Desk", "price" => 549.99 }
]
```

DatabaseResult

`fetch` returns a `DatabaseResult` object. It behaves like an array 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 enumerable. Use it directly in loops:

```
result.each do |user|
  puts user["name"]
end
```

Index Access

Access rows by index like a regular array:

```
first_user = result[0]
```

Countable

`length` works on the result:

```
puts result.length  # number of records in this result set
```

Conversion Methods

```
result.to_json       # JSON string of all records
result.to_csv        # CSV string with column headers
result.to_array      # plain array of hashes
result.to_paginate   # { "records" => [...], "count" => 42, "limit" => 10, "offset" => 0 }
```

`to_paginate` is designed for building paginated API responses. It bundles the records with the total count, limit, and offset in a single hash.

Schema Metadata with column_info

`column_info` returns detailed metadata about the columns in the result set. The data is lazy-loaded -- it only queries the database schema when you call the method for the first time:

```
info = result.column_info
# [
#     { "name" => "id", "type" => "INTEGER", "size" => nil, "decimals" => nil, "nullable" => fal
#     { "name" => "name", "type" => "TEXT", "size" => nil, "decimals" => nil, "nullable" => fals
#     { "name" => "email", "type" => "TEXT", "size" => 255, "decimals" => nil, "nullable" => tru
#     ...
# ]
```

Each column entry contains:

Field	Description
`name`	Column name
`type`	Database type (e.g. `INTEGER`, `TEXT`, `REAL`)

`size`	Maximum size (or `nil` if not applicable)
`decimals`	Decimal places (or `nil`)
`nullable`	Whether the column allows `NULL`
`primary_key`	Whether the column is part of the primary key

This is useful for building dynamic forms, generating documentation, or validating data before insert.

fetch_one -- Get a Single Row

```
product = db.fetch_one("SELECT * FROM products WHERE id = 1")
# Returns: { "id" => 1, "name" => "Keyboard", "price" => 79.99 }
```

If no row matches, `fetch_one` returns `nil`.

execute -- Run a Statement

For INSERT, UPDATE, DELETE, and DDL statements that do not return rows:

```
db.execute("INSERT INTO products (name, price) VALUES ('Widget', 9.99)")
db.execute("UPDATE products SET price = 89.99 WHERE id = 1")
db.execute("DELETE FROM products WHERE id = 5")
db.execute("CREATE TABLE IF NOT EXISTS logs (id INTEGER PRIMARY KEY, message TEXT, created_at TE
```

Full Example: A Simple Query Route

```
Tina4::Router.get("/api/products") do |request, response|
  db = Tina4.database

  products = db.fetch("SELECT * FROM products ORDER BY name")

  response.json({
    products: products,
    count: products.length
  })
end
```

```
curl http://localhost:7147/api/products
```

```
{
  "products": [
    {"id": 1, "name": "Keyboard", "price": 79.99, "in_stock": 1},
    {"id": 2, "name": "Mouse", "price": 29.99, "in_stock": 1},
    {"id": 3, "name": "Monitor", "price": 399.99, "in_stock": 0}
  ],
  "count": 3
}
```

---

5. Parameterised Queries

Never concatenate user input into SQL strings. That road leads to SQL injection:

```
# NEVER do this:
db.fetch("SELECT * FROM products WHERE name = '#{user_input}'")
```

Instead, use parameterised queries. Pass parameters as the second argument:

```
db = Tina4.database

# Positional parameters with ?
product = db.fetch_one(
  "SELECT * FROM products WHERE id = ?",
  [42]
)

# Multiple parameters
products = db.fetch(
  "SELECT * FROM products WHERE price BETWEEN ? AND ? ORDER BY price",
  [10.00, 100.00]
)
```

The database driver handles escaping. Your input is never part of the SQL string.

A Safe Search Endpoint

```
Tina4::Router.get("/api/products/search") do |request, response|
  db = Tina4.database

  q = request.params["q"] || ""
  max_price = (request.params["max_price"] || 99999).to_f

  if q.empty?
    return response.json({ error: "Query parameter 'q' is required" }, 400)
  end

  products = db.fetch(
    "SELECT * FROM products WHERE name LIKE ? AND price <= ? ORDER BY name",
    ["%#{q}%", max_price]
  )

  response.json({
    query: q,
    max_price: max_price,
    results: products,
    count: products.length
  })
end
```

```
curl "http://localhost:7147/api/products/search?q=key&max_price=100"
```

```
{
  "query": "key",
  "max_price": 100,
  "results": [
    {"id": 1, "name": "Wireless Keyboard", "price": 79.99, "in_stock": 1}
  ],
  "count": 1
}
```

---

6. Transactions

Multiple operations must succeed or fail together. Transactions enforce that contract:

```
Tina4::Router.post("/api/orders") do |request, response|
  db = Tina4.database
  body = request.body

  begin
    db.transaction do |tx|
      # Create the order
      tx.execute(
        "INSERT INTO orders (customer_id, total, status) VALUES (?, ?, 'pending')",
        [body["customer_id"], body["total"]]
      )

      # Get the new order ID
      order = tx.fetch_one("SELECT last_insert_rowid() AS id")
      order_id = order["id"]

      # Create order items
      body["items"].each do |item|
        tx.execute(
          "INSERT INTO order_items (order_id, product_id, quantity, price) VALUES (?, ?, ?, ?)",
          [order_id, item["product_id"], item["quantity"], item["price"]]
        )

        # Decrease stock
        tx.execute(
          "UPDATE products SET stock = stock - ? WHERE id = ?",
          [item["quantity"], item["product_id"]]
        )
      end
    end

    response.json({ order_id: order_id, status: "created" }, 201)
  rescue => e
    response.json({ error: "Order failed: #{e.message}" }, 500)
  end
end
```

If any step inside the block fails, the transaction is automatically rolled back. If the block completes without error, the transaction is automatically committed. The database never sits in a half-finished state.

---

7. Schema Inspection

Tina4 provides methods to inspect your database structure at runtime:

tables

```
db = Tina4.database
tables = db.tables
```

Returns an array of table names:

```
["orders", "order_items", "products", "users"]
```

columns

```
columns = db.columns("products")
```

Returns an array of column definitions:

```
[
  { "name" => "id", "type" => "INTEGER", "nullable" => false, "primary" => true },
  { "name" => "name", "type" => "TEXT", "nullable" => false, "primary" => false },
  { "name" => "price", "type" => "REAL", "nullable" => true, "primary" => false },
  { "name" => "in_stock", "type" => "INTEGER", "nullable" => true, "primary" => false }
]
```

table_exists?

```
if db.table_exists?("products")
  # Table exists, safe to query
end
```

Returns `true` if the table exists, `false` otherwise.

driver_name

```
db.driver_name
# "sqlite", "postgres", "mysql", "mssql", or "firebird"
```

Returns a lowercase string identifying the active database engine. This matters when you write engine-specific SQL in a multi-database setup.

A Schema Info Endpoint

```
Tina4::Router.get("/api/schema") do |request, response|
  db = Tina4.database
  tables = db.tables

  schema = {}
  tables.each do |table|
    schema[table] = db.columns(table)
  end

  response.json({ tables: schema })
end
```

---

8. Batch Operations with execute_many

Insert or update many rows efficiently:

```
db = Tina4.database

params_list = [
  ["Widget A", 9.99],
  ["Widget B", 14.99],
  ["Widget C", 19.99],
  ["Widget D", 24.99]
]

db.execute_many(
  "INSERT INTO products (name, price) VALUES (?, ?)",
  params_list
)
```

`execute_many` prepares the statement once and executes it for each item in the array. This is significantly faster than calling `execute` in a loop because the SQL only needs to be parsed once.

---

9. Helper Methods: insert, update, delete

Tina4 provides shorthand methods so you do not have to write SQL for simple operations.

insert

```
db = Tina4.database

# Insert a single row
db.insert("products", {
  name: "Wireless Mouse",
  price: 34.99,
  in_stock: 1
})

# Insert multiple rows
db.insert("products", [
  { name: "USB Cable", price: 9.99, in_stock: 1 },
  { name: "HDMI Cable", price: 14.99, in_stock: 1 },
  { name: "DisplayPort Cable", price: 19.99, in_stock: 0 }
])
```

update

```
# Update rows matching a filter
db.update("products", { price: 39.99, in_stock: 1 }, { id: 7 })
```

The second argument is the data to set, and the third is a filter hash for the WHERE clause. Each key-value pair becomes an `AND` condition.

delete

```
# Delete rows matching a filter
db.delete("products", { id: 7 })
```

These helpers generate SQL for you. Convenient for simple CRUD. Raw queries still own complex joins, subqueries, and aggregations.

---

10. Migrations

Migrations are versioned scripts that evolve your schema over time. No manual `CREATE TABLE` statements. Write migration files. Tina4 applies them in order.

Ruby's migration system is unique among Tina4 implementations: it supports both `.sql` and `.rb` migration files.

File Naming

Two naming patterns are accepted:

Pattern	Example
Sequential	`000001_create_products_table.sql`
Timestamp	`20260324120000_create_products_table.sql`

Pick one pattern and stick with it. Do not mix sequential and timestamp naming in the same project.

Generating a Migration

Use the CLI to scaffold migration files:

```
tina4 generate migration create_products_table
```

```
Created migration: migrations/20260324120000_create_products_table.sql
Created migration: migrations/20260324120000_create_products_table.down.sql
```

The generator creates both the up and down files. The timestamp prefix ensures migrations always run in chronological order.

SQL Migrations

Edit the generated file `migrations/20260324120000_create_products_table.sql`:

```
-- migrations/20260324120000_create_products_table.sql
CREATE TABLE products (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    name TEXT NOT NULL,
    category TEXT NOT NULL DEFAULT 'Uncategorized',
    price REAL NOT NULL DEFAULT 0.00,
    in_stock INTEGER NOT NULL DEFAULT 1,
    created_at TEXT DEFAULT CURRENT_TIMESTAMP,
    updated_at TEXT DEFAULT CURRENT_TIMESTAMP
);
```

The down migration goes in the separate `.down.sql` file:

```
-- migrations/20260324120000_create_products_table.down.sql
DROP TABLE IF EXISTS products;
```

The `.down.sql` file is optional. If it does not exist, rollback for that migration is skipped.

The SQL parser supports `$$` delimited stored procedures and block comments, so you can include complex database objects in a single migration file.

Ruby Class Migrations

As an alternative to SQL files, you can write migrations as Ruby classes. This is useful when you need conditional logic, loops, or data transformations during a migration:

```
# migrations/20260324120000_create_products_table.rb
class CreateProductsTable < Tina4::MigrationBase
  def up
    execute "CREATE TABLE products (id INTEGER PRIMARY KEY, name TEXT NOT NULL, price REAL DEFAU
  end

  def down
```

```
    execute "DROP TABLE IF EXISTS products"
  end
end
```

For `.rb` migrations, the `down` method handles rollback directly in the same file. No separate `.down.sql` file is needed.

Running Migrations

```
tina4 migrate
```

```
Running migrations...
  [APPLIED] 20260324120000_create_products_table.sql
Migrations complete. 1 applied.
```

Checking Migration Status

```
tina4ruby migrate:status
```

```
Migration                                    Status     Applied At
---------                                    ------     ----------
20260324120000_create_products_table.sql      applied    2026-03-24 12:00:00
20260324130000_create_orders_table.sql        pending    -
```

Rolling Back

```
tina4ruby migrate:rollback
```

```
Rolling back last batch...
  [ROLLED BACK] 20260324130000_create_orders_table.sql
Rollback complete. 1 rolled back.
```

Rollback undoes the entire last batch of migrations. If you applied three migrations in one `tina4 migrate` run, all three are rolled back together. You can configure the number of steps if you need finer control.

For `.sql` migrations, rollback runs the corresponding `.down.sql` file. For `.rb` migrations, rollback calls the `down` method.

A Real Migration Sequence

Here is what a typical project's migrations look like:

```
migrations/
■■■ 20260324120000_create_products_table.sql
■■■ 20260324120000_create_products_table.down.sql
■■■ 20260324121000_create_users_table.sql
■■■ 20260324121000_create_users_table.down.sql
■■■ 20260324122000_create_orders_table.rb
■■■ 20260325091500_add_email_index_to_users.sql
```

Notice the mix of `.sql` and `.rb` files. Both types can coexist in the same project. The `create_orders_table.rb` file contains both `up` and `down` methods, so it does not need a separate down file.

The last SQL migration might look like:

```
-- migrations/20260325091500_add_email_index_to_users.sql
CREATE INDEX idx_users_email ON users (email);
```

With its down file:

```
-- migrations/20260325091500_add_email_index_to_users.down.sql
DROP INDEX IF EXISTS idx_users_email;
```

Migration Tracking

Migrations run in filename order. Each migration runs once. Tina4 tracks applied migrations in a `tina4_migration` table with the following columns:

Column	Description
`id`	Auto-increment primary key
`migration_name`	The filename of the migration
`batch`	The batch number (all migrations applied in one run share a batch)
`executed_at`	Timestamp of when the migration was applied

The batch system is what makes rollback work: `migrate:rollback` undoes all migrations in the highest batch number.

---

11. Query Caching

For read-heavy applications, enable query caching:

```
TINA4_DB_CACHE=true
```

When enabled, Tina4 caches the results of `fetch` and `fetch_one` calls. Identical queries with identical parameters return cached results instead of hitting the database again.

The cache is automatically invalidated when you call `execute`, `insert`, `update`, or `delete` on the same table.

You can also control caching per-query:

```
# Force a fresh query (bypass cache)
products = db.fetch("SELECT * FROM products", [], false) # third arg = use cache

# Clear the entire cache
db.cache_clear
```

---

12. Exercise: Build a Notes App

Build a notes application backed by SQLite. Create the database table via a migration and build a full CRUD API.

Requirements

- Create a migration that creates a ~~`notes` table with columns~~:

- `id` -- integer, primary key, auto-increment
- `title` -- text, not null
- `content` -- text, not null
- `tag` -- text, default "general"
- `created_at` -- text, default current timestamp
- `updated_at` -- text, default current timestamp
- Build these API endpoints:

Method	Path	Description
`GET`	`/api/notes`	List all notes. Support `?tag=` and `?search=` filters.
`GET`	`/api/notes/{id:int}`	Get a single note. 404 if not found.
`POST`	`/api/notes`	Create a note. Validate title and content are not empty.
`PUT`	`/api/notes/{id:int}`	Update a note. 404 if not found.
`DELETE`	`/api/notes/{id:int}`	Delete a note. 204 on success, 404 if not found.

Test with:

```
# Create
curl -X POST http://localhost:7147/api/notes \
  -H "Content-Type: application/json" \
  -d '{"title": "Shopping List", "content": "Milk, eggs, bread", "tag": "personal"}'

# List all
curl http://localhost:7147/api/notes

# Search
curl "http://localhost:7147/api/notes?search=shopping"

# Filter by tag
curl "http://localhost:7147/api/notes?tag=personal"

# Get one
curl http://localhost:7147/api/notes/1

# Update
curl -X PUT http://localhost:7147/api/notes/1 \
  -H "Content-Type: application/json" \
  -d '{"title": "Updated Shopping List", "content": "Milk, eggs, bread, butter"}'

# Delete
curl -X DELETE http://localhost:7147/api/notes/1
```

---

13. Solution

Migration

Generate the migration:

```
tina4 generate migration create_notes_table
```

Edit `migrations/20260324120000_create_notes_table.sql`:

```
CREATE TABLE notes (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    title TEXT NOT NULL,
    content TEXT NOT NULL,
    tag TEXT NOT NULL DEFAULT 'general',
    created_at TEXT DEFAULT CURRENT_TIMESTAMP,
    updated_at TEXT DEFAULT CURRENT_TIMESTAMP
);
```

Edit `migrations/20260324120000_create_notes_table.down.sql`:

```
DROP TABLE IF EXISTS notes;
```

Run the migration:

```
tina4 migrate
```

```
Running migrations...
  [APPLIED] 20260322143000_create_notes_table.sql
Migrations complete. 1 applied.
```

Routes

Create `src/routes/notes.rb`:

```
# List all notes with optional filters
Tina4::Router.get("/api/notes") do |request, response|
  db = Tina4.database

  tag = request.params["tag"] || ""
  search = request.params["search"] || ""

  sql = "SELECT * FROM notes"
  params = []
  conditions = []

  unless tag.empty?
    conditions << "tag = ?"
    params << tag
  end

  unless search.empty?
    conditions << "(title LIKE ? OR content LIKE ?)"
    params << "%#{search}%"
    params << "%#{search}%"
  end

  sql += " WHERE #{conditions.join(' AND ')}" unless conditions.empty?
```

```
  sql += " ORDER BY updated_at DESC"

  notes = db.fetch(sql, params)

  response.json({
    notes: notes,
    count: notes.length
  })
end

# Get a single note
Tina4::Router.get("/api/notes/{id:int}") do |request, response|
  db = Tina4.database
  id = request.params["id"]

  note = db.fetch_one("SELECT * FROM notes WHERE id = ?", [id])

  if note.nil?
    response.json({ error: "Note not found", id: id }, 404)
  else
    response.json(note)
  end
end

# Create a note
Tina4::Router.post("/api/notes") do |request, response|
  db = Tina4.database
  body = request.body

  # Validate
  errors = []
  errors << "Title is required" if body["title"].nil? || body["title"].empty?
  errors << "Content is required" if body["content"].nil? || body["content"].empty?

  unless errors.empty?
    return response.json({ errors: errors }, 400)
  end

  db.execute(
    "INSERT INTO notes (title, content, tag) VALUES (?, ?, ?)",
    [body["title"], body["content"], body["tag"] || "general"]
  )

  note = db.fetch_one("SELECT * FROM notes WHERE id = last_insert_rowid()")

  response.json(note, 201)
end

# Update a note
Tina4::Router.put("/api/notes/{id:int}") do |request, response|
  db = Tina4.database
  id = request.params["id"]
  body = request.body

  existing = db.fetch_one("SELECT * FROM notes WHERE id = ?", [id])

  if existing.nil?
    return response.json({ error: "Note not found", id: id }, 404)
  end
```

```
  db.execute(
    "UPDATE notes SET title = ?, content = ?, tag = ?, updated_at = CURRENT_TIMESTAMP WHERE id =
    [
      body["title"] || existing["title"],
      body["content"] || existing["content"],
      body["tag"] || existing["tag"],
      id
    ]
  )

  note = db.fetch_one("SELECT * FROM notes WHERE id = ?", [id])

  response.json(note)
end

# Delete a note
Tina4::Router.delete("/api/notes/{id:int}") do |request, response|
  db = Tina4.database
  id = request.params["id"]

  existing = db.fetch_one("SELECT * FROM notes WHERE id = ?", [id])

  if existing.nil?
    return response.json({ error: "Note not found", id: id }, 404)
  end

  db.execute("DELETE FROM notes WHERE id = ?", [id])

  response.json(nil, 204)
end
```

Expected output for create:

```
{
  "id": 1,
  "title": "Shopping List",
  "content": "Milk, eggs, bread",
  "tag": "personal",
  "created_at": "2026-03-22 14:30:00",
  "updated_at": "2026-03-22 14:30:00"
}
```

(Status: `201 Created`)

Expected output for list:

```
{
  "notes": [
    {
      "id": 1,
      "title": "Shopping List",
      "content": "Milk, eggs, bread",
      "tag": "personal",
      "created_at": "2026-03-22 14:30:00",
      "updated_at": "2026-03-22 14:30:00"
    }
  ],
  "count": 1
}
```

Expected output for search:

```
{
  "notes": [
    {
      "id": 1,
      "title": "Shopping List",
      "content": "Milk, eggs, bread",
      "tag": "personal",
      "created_at": "2026-03-22 14:30:00",
      "updated_at": "2026-03-22 14:30:00"
    }
  ],
  "count": 1
}
```

Expected output for validation error:

```
{"errors": ["Title is required", "Content is required"]}
```

(Status: `400 Bad Request`)

---

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. `seed_orm` and `seed_table` insert it in bulk.

FakeData

```
fake = Tina4::FakeData.new

fake.name        # "Grace Lopez"
fake.email       # "bob.anderson123@demo.net"
fake.phone       # "+1 (547) 382-9104"
fake.sentence    # "Magna exercitation lorem ipsum dolor sit."
fake.paragraph   # Three sentences of filler text
fake.integer     # 7342
fake.numeric     # 481.29
fake.date        # "2023-07-14"
fake.uuid        # "a3f1b2c4-d5e6-f7a8-b9c0-d1e2f3a4b5c6"
fake.address     # "742 Oak Avenue"
fake.boolean     # 0 or 1
fake.city        # "Tokyo"
fake.country     # "Germany"
fake.company     # "TechGlobal Inc"
fake.url         # "https://example.com/some-random-slug"
fake.password    # "xK9mPq2nRt5vWz8a"
```

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 = Tina4::FakeData.new(seed: 42)
fake.name   # Always the same name with seed 42
fake.email  # Always the same email with seed 42
```

Deterministic data means deterministic assertions. This matters for tests.

Seeding with ORM

`seed_orm` combines `FakeData` with your ORM models. It reads the model's field definitions and generates appropriate data for each column:

```
Tina4.seed_orm(User, count: 100)
```

This inserts 100 rows into the `users` table. `FakeData` inspects each column name and type to generate matching data -- email columns get email addresses, phone columns get phone numbers, name columns get names.

Overrides

Static values or custom generators apply through the `overrides` hash:

```
Tina4.seed_orm(User, count: 50, overrides: {
  role: "member",
  active: 1,
  email: ->(fake) { fake.email }
})
```

Every row gets `role = "member"` and `active = 1`. Lambda values receive the `FakeData` instance for custom generation.

Seeding a Raw Table

When you have no ORM class, use `seed_table` with a column map:

```
Tina4.seed_table("users", {
  name: :string,
  email: :string,
  phone: :string,
  bio: :text,
  age: :integer
}, count: 100)
```

The column types tell `FakeData` what kind of data to generate for each field.

Batch Seeding

Seed multiple tables in order with `seed_batch`. Pass `clear: true` to delete existing records first (in reverse order, respecting foreign keys):

```
Tina4.seed_batch([
  { orm_class: User, count: 20 },
  { orm_class: Order, count: 100, overrides: { status: "pending" } }
], clear: true)
```

Seed Files

Place seed scripts in a `seeds/` folder. Each file is a Ruby script that runs `seed_orm` or `seed_table`:

```
# seeds/001_users.rb
Tina4.seed_orm(User, count: 50, seed: 1)

# seeds/002_orders.rb
```

```
Tina4.seed_orm(Order, count: 200, overrides: {
  status: ->(f) { f.choice(%w[pending shipped delivered]) }
})
```

Run all seed files:

```
Tina4.seed(seed_folder: "seeds")
```

Files run in alphabetical order. Prefix with numbers to control sequence. Files starting with `_` are skipped.

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. Forgetting the transaction block

Problem: You run multiple related queries but they are not atomic -- some succeed while others fail, leaving the database in an inconsistent state.

Cause: Without wrapping them in a `transaction` block, each query is committed independently.

Fix: Use `db.transaction { |tx| ... }` to wrap related operations. The block automatically commits on success and rolls back on any exception.

2. Connection String Formats

Problem: The database will not connect and you see a cryptic error about the connection string.

Cause: Each database engine expects a specific URL format. A common mistake is using `mysql://user:pass@host/db` when the engine expects the port.

Fix: Always include the port: `mysql://localhost:3306/mydb`. Here are the default ports:

Engine	Default Port
PostgreSQL	5432
MySQL	3306
MSSQL	1433
Firebird	3050
SQLite	(file path, no port)

3. SQLite File Paths

Problem: SQLite creates a new empty database instead of using the existing one.

Cause: The path in `TINA4_DATABASE_URL` is relative and resolves to the wrong directory, or you used `sqlite://` (two slashes) instead of `sqlite:///` (three slashes).

Fix: Use three slashes for a relative path: `sqlite:///data/app.db`. For an absolute path, use four slashes: `sqlite:////var/data/app.db`. The third slash separates the scheme from the path; the fourth starts the absolute path.

4. Parameterised Queries with LIKE

Problem: `WHERE name LIKE ?` with `["%search%"]` works, but `WHERE name LIKE '%?%'` does not.

Cause: Parameters inside quotes are treated as literal text, not as placeholders.

Fix: Include the `%` wildcards in the parameter value, not in the SQL: `["%#{search}%"]`. The SQL should be `WHERE name LIKE ?`.

5. Boolean Values in SQLite

Problem: You insert `true` or `false` but the database stores `1` or `0`. When you read it back, you get integers, not booleans.

Cause: SQLite does not have a native boolean type. It stores booleans as integers.

Fix: Cast in your Ruby code: `row["in_stock"] == 1` or use `!!row["in_stock"]` to convert to a boolean. Ruby treats `0` as truthy (unlike some other languages), so be explicit with comparisons.

6. Migration Already Applied

Problem: You edited a migration file and ran `tina4 migrate` again, but nothing changed.

Cause: Tina4 tracks applied migrations by filename in the `tina4_migration` table. Once applied, a migration will not run again even if you change its contents.

Fix: Create a new migration for schema changes. Do not edit applied migrations. If you are in early development and want to start fresh, use `tina4ruby migrate:rollback` to undo the last batch and then `tina4 migrate` to reapply. Use `tina4ruby migrate:status` to see which migrations are applied and which are pending.

7. fetch Returns Empty Array, Not Nil

Problem: You check `if result.nil?` but it never matches, even when the table is empty.

Cause: `fetch` always returns an array. An empty result is `[]`, not `nil`. Only `fetch_one` returns `nil` when no row matches.

Fix: Check with `if result.empty?` or `if result.length == 0`.

8. SQL Injection Through String Interpolation

Problem: Your application is vulnerable to SQL injection attacks.

Cause: You used string interpolation to build SQL queries with user input: `"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 hash keys. Over and over.

Tina4's ORM turns database rows into Ruby 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 `.rb` file in that directory is auto-loaded.

Create `src/orm/note.rb`:

```
class Note < Tina4::ORM
  integer_field :id, primary_key: true, auto_increment: true
  string_field :title, nullable: false
  string_field :content, default: ""
  string_field :category, default: "general"
  boolean_field :pinned, default: false
  datetime_field :created_at
  datetime_field :updated_at
end
```

A complete model. Here is what each piece does:

- The table name defaults to the lowercase, pluralised class name (`Note` -> `notes`). Override it with `table_name "my_table"` inside the class body.
- `primary_key: true` on a field marks it as the primary key (defaults to `:id` if none is specified)
- Each field is a DSL declaration that creates a getter and setter on the model

Field Types

Field Type	Ruby Type	SQL Type	Description
`integer_field`	`Integer`	`INTEGER`	Whole numbers
`string_field`	`String`	`VARCHAR(255)`	Text strings
`text_field`	`String`	`TEXT`	Long text
`float_field`	`Float`	`REAL`	Floating-point numbers
`decimal_field`	`Float`	`REAL`	Decimal numbers (precision/scale options)
`numeric_field`	`Float`	`REAL`	Alias for float_field
`boolean_field`	`Integer`	`INTEGER` (0/1)	True/False stored as 0/1
`date_field`	`String`	`DATE`	Date values
`datetime_field`	`String`	`DATETIME`	Date and time
`timestamp_field`	`String`	`TIMESTAMP`	Timestamps
`blob_field`	`String`	`BLOB`	Binary data
`json_field`	`String`	`TEXT`	JSON stored as text

For foreign keys, use `integer_field`. There is no separate foreign key field type -- the relationship is defined through `has_many`, `has_one`, and `belongs_to` declarations instead.

Field Options

Option	Type	Description
`primary_key`	`bool`	Marks this field as the primary key
`auto_increment`	`bool`	Auto-incrementing integer
`nullable`	`bool`	Whether the field accepts nil (default: `true`)
`default`	any	Default value when not provided
`length`	`int`	String length for `string_field` (default: 255)
`precision`	`int`	Decimal precision for `decimal_field`
`scale`	`int`	Decimal scale for `decimal_field`

Field Mapping

When your Ruby attribute names do not match the database column names, use `field_mapping` to define the translation. `field_mapping` is a hash that maps Ruby attribute names to DB column names.

```
class User < Tina4::ORM
  self.field_mapping = {
    "first_name"    => "fname",      # Ruby attr -> DB column
    "last_name"     => "lname",
    "email_address" => "email"
  }

  integer_field :id, primary_key: true, auto_increment: true
  string_field :first_name, nullable: false
  string_field :last_name, nullable: false
  string_field :email_address, nullable: false

  table_name "user_accounts"
end
```

With this mapping, `user.first_name` reads from and writes to the `fname` column. The ORM handles the conversion in both directions -- on reads via `from_hash` and on writes via `to_db_hash`. 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:

```
class Account < Tina4::ORM
  self.table_name   = "ACCOUNTS"
```

```
  self.field_mapping = {
    "id"           => "CUST_ID",
    "account_no"   => "ACCOUNTNO",
    "store_name"   => "STORENAME",
    "credit_limit" => "CREDITLIMIT",
  }

  integer_field :id, primary_key: true, auto_increment: true
  string_field :account_no
  string_field :store_name
  float_field :credit_limit, default: 0.0
end
```

Ruby code uses clean snake_case names (`account.account_no`, `account.credit_limit`). The ORM maps them to the uppercase DB columns automatically.

find() vs where() -- naming convention

The two query methods have a deliberate difference in how they handle column names:

- `find(filter)` uses **Ruby attribute names**. The ORM translates them via `field_mapping`.
- `where(conditions, params)` uses **raw DB column names** in the SQL string. No translation is done.

```
# find() -- use Ruby 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 Ruby it is a no-op because Ruby 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 methods are available:

```
Tina4.snake_to_camel("first_name")   # "firstName"
Tina4.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.

```
irb
irb> require_relative "app"
irb> Note.create_table
```

---

4. CRUD Operations

save -- Create or Update

```
Tina4::Router.post "/api/notes" do |request, response|
  note = Note.new
  note.title = request.body_parsed["title"]
  note.content = request.body_parsed["content"] || ""
  note.category = request.body_parsed["category"] || "general"
  note.pinned = request.body_parsed["pinned"] || false
  result = note.save

  if result
    response.json({ message: "Note created", note: note.to_h }, status: 201)
  else
    response.json({ errors: note.errors }, status: 422)
  end
end
```

`save` detects whether the record is new (INSERT) or existing (UPDATE) based on whether the primary key has a value and the record is persisted. It returns `self` on success, so you can chain calls. It returns `false` on failure -- check `note.errors` for details.

create -- Build and Save in One Step

When you have a hash 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"
})
```

find*by*id -- Fetch One Record by Primary Key

```
Tina4::Router.get "/api/notes/{id:int}" do |request, response|
  note = Note.find_by_id(request.params["id"].to_i)

  if note.nil?
    response.json({ error: "Note not found" }, status: 404)
  else
    response.json(note.to_h)
  end
end
```

`find_by_id` takes a primary key value and returns a model instance, or `nil` if no row matches. If soft delete is enabled, it excludes soft-deleted records.

Use `find_or_fail` when you want an exception raised instead of `nil`:

```
note = Note.find_or_fail(id)  # Raises RuntimeError if not found
```

find -- Query by Filter Hash

The `find` method accepts a hash of column-value pairs and returns an array 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
```

Both hash syntax (`find({category: "work"})`) and keyword syntax (`find(category: "work")`) are accepted. Ruby attribute names are used in the filter -- the ORM applies `field_mapping` automatically when translating to SQL column names.

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

```
Tina4::Router.delete "/api/notes/{id:int}" do |request, response|
  note = Note.find_by_id(request.params["id"].to_i)

  if note.nil?
    response.json({ error: "Note not found" }, status: 404)
  else
    note.delete
    response.json(nil, status: 204)
  end
end
```

Listing Records

```
Tina4::Router.get "/api/notes" do |request, response|
  category = request.query["category"]

  if category
    notes = Note.where("category = ?", [category])
  else
    notes = Note.all
  end

  response.json({
    notes: notes.map(&:to_h),
    count: notes.length
  })
end
```

`where` takes a WHERE clause with `?` placeholders and an array of parameters. It returns an array of model instances. `all` fetches all records. Both support pagination:

```
# With pagination
notes = Note.where("category = ?", ["work"])
```

```
# Fetch all with pagination and ordering
notes = Note.all(limit: 20, offset: 0, order_by: "created_at DESC")

# 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 `nil`.

load -- Populate an Existing Instance

The `load` method fills an existing model instance from the database:

```
note = Note.new
note.id = 42
note.load  # Loads data for id=42

# Or with a filter string
note = Note.new
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*h, to*dict, to_json, and Other Serialisation

to*h and to*dict

Convert a model instance to a hash. `to_dict` is a direct alias for `to_h` -- use whichever reads more naturally in your code:

```
note = Note.find_by_id(1)

data = note.to_h
# {id: 1, title: "Shopping List", content: "Milk, eggs", category: "personal", pinned: false, cr

# to_dict is identical
data = note.to_dict
```

The `include` parameter adds relationship data to the output (see Eager Loading below). Pass an array of relationship names:

```
# Include relationships in the hash
data = note.to_h(include: [:comments])
```

```
data = note.to_dict(include: [:comments])  # same result
```

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_h(include: nil)`	`Hash`	Primary hash method with optional relationship includes
`to_hash(include: nil)`	`Hash`	Alias for `to_h`
`to_dict(include: nil)`	`Hash`	Alias for `to_h`
`to_assoc(include: nil)`	`Hash`	Alias for `to_h`
`to_object`	`Hash`	Alias for `to_h`
`to_json(include: nil)`	`String`	JSON string
`to_array`	`Array`	Flat list of values (no keys)
`to_list`	`Array`	Alias for `to_array`

---

6. Relationships

Tina4 Ruby supports two styles of relationships: declarative (class-level DSL) and imperative (instance-level method calls). Both produce the same queries. Declarative relationships enable eager loading; imperative relationships are ad-hoc.

foreign*key*field — Auto-Wired Relationships

Declaring a column with `foreign_key_field :user_id, references: User` automatically wires both sides of the relationship. The declaring class gets a `belongs_to` accessor (the column name with `_id` stripped), and the referenced class gets a `has_many` accessor (the declaring class name lowercased with `s` appended, or whatever you pass via `related_name:`).

```
class User < Tina4::ORM
  table_name "users"
  integer_field :id, primary_key: true
  string_field :name
end

class Post < Tina4::ORM
  table_name "posts"
  integer_field :id, primary_key: true
  string_field :title
```

```
  # Auto-wires post.user (belongs_to) and user.posts (has_many)
  foreign_key_field :user_id, references: User
end
```

With just the `foreign_key_field` declaration, both sides are accessible:

```
post = Post.find_by_id(1)
puts post.user.name        # "Alice"

user = User.find_by_id(1)
user.posts.each do |p|
  puts p.title
end
```

For a custom `has_many` name, pass `related_name:`:

```
foreign_key_field :user_id, references: User, related_name: :blog_posts
# user.blog_posts instead of user.posts
```

If the referenced class is defined later, the framework handles deferred wiring — as soon as the referenced class applies its own field definitions, the `has_many` is injected.

Declarative Relationships

Define relationships at the class level. The ORM creates accessor methods on each instance.

has_many

An author has many posts:

Create `src/orm/author.rb`:

```
class Author < Tina4::ORM
  integer_field :id, primary_key: true, auto_increment: true
  string_field :name, nullable: false
  string_field :email, nullable: false
  string_field :bio, default: ""
  datetime_field :created_at

  has_many :posts, class_name: "BlogPost", foreign_key: "author_id"
end
```

Create `src/orm/blog_post.rb`:

```
class BlogPost < Tina4::ORM
  integer_field :id, primary_key: true, auto_increment: true
  integer_field :author_id, nullable: false
  string_field :title, nullable: false
  string_field :slug, nullable: false
  string_field :content, default: ""
  string_field :status, default: "draft"
  datetime_field :created_at
  datetime_field :updated_at

  table_name "posts"

  belongs_to :author, class_name: "Author", foreign_key: "author_id"
end
```

Now access an author's posts:

```
Tina4::Router.get "/api/authors/{id:int}" do |request, response|
  author = Author.find_by_id(request.params["id"].to_i)

  if author.nil?
    response.json({ error: "Author not found" }, status: 404)
  else
    posts = author.posts  # Calls the has_many accessor

    data = author.to_h
    data[:posts] = posts.map(&:to_h)
    response.json(data)
  end
end
```

has_one

A user has one profile:

```
class User < Tina4::ORM
  integer_field :id, primary_key: true, auto_increment: true
  string_field :name, nullable: false

  has_one :profile, class_name: "Profile", foreign_key: "user_id"
end
```

```
profile = user.profile  # Returns a single instance or nil
```

belongs_to

A post belongs to an author:

```
Tina4::Router.get "/api/posts/{id:int}" do |request, response|
  post = BlogPost.find_by_id(request.params["id"].to_i)

  if post.nil?
    response.json({ error: "Post not found" }, status: 404)
  else
    author = post.author  # Calls the belongs_to accessor

    data = post.to_h
    data[:author] = author&.to_h
    response.json(data)
  end
end
```

Imperative Relationships

For ad-hoc queries without class-level declarations, use `query_has_many`, `query_has_one`, and `query_belongs_to` on any instance:

```
# Same result as declarative has_many, but without a class-level declaration
posts = author.query_has_many(BlogPost, foreign_key: "author_id")

# Has one
profile = user.query_has_one(Profile, foreign_key: "user_id")

# Belongs to
author = post.query_belongs_to(Author, foreign_key: "author_id")
```

These work identically to the declarative accessors but do not support eager loading.

---

7. Eager Loading

Calling relationship methods inside a loop creates the N+1 problem. Load 10 authors. Call `author.posts` for each one. That fires 11 queries -- 1 for authors, 10 for posts. The page drags.

The `include` parameter on `all`, `where`, `find`, and `select` solves this. It eager-loads relationships in bulk:

```
Tina4::Router.get "/api/authors" do |request, response|
  authors = Author.all(include: ["posts"])

  data = authors.map do |author|
    author_dict = author.to_h
    author_dict[:posts] = author.posts.map(&:to_h)
    author_dict
  end

  response.json({ authors: data })
end
```

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.

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_h with Nested Includes

When eager loading is active, `to_h(include: ...)` embeds the related data:

```
post = BlogPost.find_by_id(1)
# Manually trigger eager load first, or use select with include
posts = BlogPost.select(
  "SELECT * FROM posts WHERE id = ?", [1],
  include: ["author", "comments"]
)
post = posts.first
data = post.to_h(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.

```
class Task < Tina4::ORM
  self.soft_delete = true  # Enable soft delete

  integer_field :id, primary_key: true, auto_increment: true
  string_field :title, nullable: false
  boolean_field :completed, default: false
  integer_field :is_deleted, default: 0  # Required for soft delete (0 = active, 1 = deleted)
  string_field :created_at
end
```

When `soft_delete` is set to `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"])
```

Custom Soft Delete Field

By default, the ORM uses `:is_deleted` as the soft delete column. You can change this:

```
class Task < Tina4::ORM
  self.soft_delete = true
  self.soft_delete_field = :deleted_flag

  integer_field :id, primary_key: true, auto_increment: true
  string_field :title, nullable: false
  integer_field :deleted_flag, default: 0
end
```

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 `self.auto_crud = true` on your model class:

```
class Note < Tina4::ORM
  self.auto_crud = true  # Generates REST endpoints automatically

  integer_field :id, primary_key: true, auto_increment: true
  string_field :title, nullable: false
  string_field :content, default: ""
end
```

The moment Ruby loads this class, the ORM registers it with `AutoCrud`. Five routes appear after `Tina4::AutoCrud.generate_routes` is called.

Manual Registration

You can also register models explicitly using `AutoCrud.register`:

```
Tina4::AutoCrud.register(Note)
```

Then generate all routes at once:

```
Tina4::AutoCrud.generate_routes(prefix: "/api")
```

Both approaches produce the same result:

Method	Path	Description
`GET`	`/api/notes`	List all with pagination (`limit`, `offset`, `page`, `per_page` 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:

```
Tina4::AutoCrud.generate_routes(prefix: "/api/v2")
# Routes: /api/v2/notes, /api/v2/notes/{id}, etc.
```

What the Generated Routes Do

GET /api/notes returns paginated results with optional filtering and sorting:

```
curl "http://localhost:7147/api/notes?limit=10&offset=0"
curl "http://localhost:7147/api/notes?filter[category]=work&sort=-created_at"
```

```
{
  "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
}
```

Sorting uses the `sort` parameter. Prefix a field with `-` for descending order: `?sort=-created_at,name`.

Filtering uses `filter[field]=value` syntax: `?filter[category]=work&filter[pinned]=true`.

POST /api/notes validates input before saving:

```
curl -X POST http://localhost:7147/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 422 with error details:

```
{"errors": ["title cannot be null"]}
```

DELETE /api/notes/1 respects soft delete. If the model has `self.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.

---

10. Scopes

Scopes are reusable query filters baked into the model. Ruby supports two approaches: instance methods and the `scope` class method.

Class Methods as Scopes

```
class BlogPost < Tina4::ORM
  integer_field :id, primary_key: true, auto_increment: true
  string_field :title, nullable: false
  string_field :status, default: "draft"
  datetime_field :created_at

  table_name "posts"

  def self.published
    where("status = ?", ["published"])
  end

  def self.drafts
    where("status = ?", ["draft"])
  end

  def self.recent(days = 7)
    where(
      "created_at > datetime('now', ?)",
      ["-#{days} days"]
    )
  end
end
```

Use them in your routes:

```
Tina4::Router.get "/api/posts/published" do |request, response|
  posts = BlogPost.published
  response.json({ posts: posts.map(&:to_h) })
end

Tina4::Router.get "/api/posts/recent" do |request, response|
  days = (request.query["days"] || 7).to_i
  posts = BlogPost.recent(days)
  response.json({ posts: posts.map(&:to_h) })
end
```

Dynamic Scopes with scope()

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.

---

11. Input Validation

Field definitions carry validation rules through the `nullable` option. Call `validate` before `save` and the ORM checks every constraint:

```
class Product < Tina4::ORM
  integer_field :id, primary_key: true, auto_increment: true
  string_field :name, nullable: false
  string_field :sku, nullable: false
  float_field :price, nullable: false
  string_field :category
end

Tina4::Router.post "/api/products" do |request, response|
  product = Product.new(request.body_parsed)

  errors = product.validate
  if errors.any?
    response.json({ errors: errors }, status: 400)
  else
    product.save
    response.json({ product: product.to_h }, status: 201)
  end
end
```

If validation fails, `validate` returns an array of error messages:

```
{
  "errors": [
    "name cannot be null",
    "sku cannot be null",
    "price cannot be null"
  ]
}
```

Note that `save` also runs field validation internally and returns `false` if any required fields are missing. Check `model.errors` after a failed save.

---

12. 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), `slug` (required), `content`, `status` (default: draft), `created_at`, `updated_at`

Comment: `id`, `post_id` (integer foreign key), `author_name` (required), `author_email` (required), `body` (required), `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

---

13. Solution

Create `src/orm/author.rb`:

```
class Author < Tina4::ORM
  integer_field :id, primary_key: true, auto_increment: true
  string_field :name, nullable: false
  string_field :email, nullable: false
  string_field :bio, default: ""
  datetime_field :created_at

  has_many :posts, class_name: "BlogPost", foreign_key: "author_id"
end
```

Create `src/orm/blog_post.rb`:

```
class BlogPost < Tina4::ORM
  integer_field :id, primary_key: true, auto_increment: true
  integer_field :author_id, nullable: false
  string_field :title, nullable: false
  string_field :slug, nullable: false
  string_field :content, default: ""
  string_field :status, default: "draft"
  datetime_field :created_at
  datetime_field :updated_at

  table_name "posts"

  belongs_to :author, class_name: "Author", foreign_key: "author_id"
  has_many :comments, class_name: "Comment", foreign_key: "post_id"

  def self.published
```

```
    where("status = ?", ["published"])
  end
end
```

Create `src/orm/comment.rb`:

```
class Comment < Tina4::ORM
  integer_field :id, primary_key: true, auto_increment: true
  integer_field :post_id, nullable: false
  string_field :author_name, nullable: false
  string_field :author_email, nullable: false
  string_field :body, nullable: false
  datetime_field :created_at

  belongs_to :post, class_name: "BlogPost", foreign_key: "post_id"
end
```

Create `src/routes/blog.rb`:

```
Tina4::Router.post "/api/authors" do |request, response|
  author = Author.new
  author.name = request.body_parsed["name"]
  author.email = request.body_parsed["email"]
  author.bio = request.body_parsed["bio"] || ""

  errors = author.validate
  if errors.any?
    response.json({ errors: errors }, status: 400)
  else
    author.save
    response.json({ author: author.to_h }, status: 201)
  end
end

Tina4::Router.get "/api/authors/{id:int}" do |request, response|
  author = Author.find_by_id(request.params["id"].to_i)

  if author.nil?
    response.json({ error: "Author not found" }, status: 404)
  else
    posts = author.posts

    data = author.to_h
    data[:posts] = posts.map(&:to_h)
    response.json(data)
  end
end

Tina4::Router.post "/api/posts" do |request, response|
  body = request.body_parsed

  # Verify author exists
  author = Author.find_by_id(body["author_id"])
  if author.nil?
    next response.json({ error: "Author not found" }, status: 404)
  end

  blog_post = BlogPost.new
  blog_post.author_id = body["author_id"]
  blog_post.title = body["title"]
```

```
  blog_post.slug = body["slug"]
  blog_post.content = body["content"] || ""
  blog_post.status = body["status"] || "draft"

  errors = blog_post.validate
  if errors.any?
    response.json({ errors: errors }, status: 400)
  else
    blog_post.save
    response.json({ post: blog_post.to_h }, status: 201)
  end
end

Tina4::Router.get "/api/posts" do |request, response|
  posts = BlogPost.published
  data = []

  posts.each do |p|
    post_dict = p.to_h
    post_dict[:author] = p.author&.to_h
    data << post_dict
  end

  response.json({ posts: data, count: data.length })
end

Tina4::Router.get "/api/posts/{id:int}" do |request, response|
  blog_post = BlogPost.find_by_id(request.params["id"].to_i)

  if blog_post.nil?
    response.json({ error: "Post not found" }, status: 404)
  else
    data = blog_post.to_h
    data[:author] = blog_post.author&.to_h
    comments = blog_post.comments
    data[:comments] = comments.map(&:to_h)
    data[:comment_count] = comments.length
    response.json(data)
  end
end

Tina4::Router.post "/api/posts/{id:int}/comments" do |request, response|
  blog_post = BlogPost.find_by_id(request.params["id"].to_i)

  if blog_post.nil?
    next response.json({ error: "Post not found" }, status: 404)
  end

  comment = Comment.new
  comment.post_id = request.params["id"].to_i
  comment.author_name = request.body_parsed["author_name"]
  comment.author_email = request.body_parsed["author_email"]
  comment.body = request.body_parsed["body"]

  errors = comment.validate
  if errors.any?
    response.json({ errors: errors }, status: 400)
  else
    comment.save
```

```
    response.json({ comment: comment.to_h }, status: 201)
  end
end
```

---

14. 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 Ruby 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 nil

Problem: You call `Note.find_by_id(id)` but get `nil` instead of a note object.

Cause: `find_by_id` returns `nil` when no row matches the given primary key. If soft delete is enabled, `find_by_id` also excludes soft-deleted records.

Fix: Check for `nil` after `find_by_id`: `if note.nil?` and return 404. Use `find_or_fail` if you want an exception raised instead.

3. find takes a hash, not keyword arguments

Problem: You call `Note.find(category: "work")` expecting a filter, but get unexpected results.

Cause: `find` takes a hash argument: `find({category: "work"})`. Passing `category: "work"` as a keyword argument does not filter -- it gets interpreted differently.

Fix: Use `find({column: value})` with an explicit hash. Use `find_by_id(id)` for primary key lookups.

4. to_h includes everything

Problem: `user.to_h` includes `password_hash` in the API response.

Cause: `to_h` includes all fields by default.

Fix: Build the response hash 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.

5. Validation runs on save, but check errors

Problem: You call `save` and it returns `false`, but you do not know why.

Cause: `save` validates required fields internally. When validation fails, it populates `model.errors` and returns `false`.

Fix: Call `errors = model.validate` before `save` for explicit error messages. Or check `model.errors` after a failed save.

6. 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` declarations, 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.

7. 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.posts` inside a loop for each author.

Fix: Use eager loading with the `include` parameter on `all`, `where`, or `select`:

```
authors = Author.all(include: ["posts"])
```

Two queries instead of 101.

8. 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.

9. Soft-deleted records appearing in queries

Problem: You soft-deleted a record, but queries still return it.

Cause: Soft delete requires `self.soft_delete = true` on the model class and an `integer_field :is_deleted, default: 0` field. Without both, soft delete is inactive.

Fix: Verify both the `self.soft_delete = true` flag and the `integer_field :is_deleted, default: 0` field exist on the model. The column stores `0` for active records and `1` for deleted ones.

10. Table name pluralisation

Problem: Your model `Category` maps to `categorys` instead of `categories`.

Cause: The ORM appends `s` by default unless the name already ends in `s`. It does not handle irregular plurals.

Fix: Set the table name explicitly with `table_name "categories"` inside the class body. Disable auto-pluralisation with the `TINA4_ORM_PLURAL_TABLE_NAMES=false` environment variable.

---

15. Raw SQL

The ORM handles 90% of queries. The other 10% need custom SQL -- reports, aggregations, complex joins. Drop down to the database directly:

```
result = Tina4.database.fetch("SELECT * FROM users WHERE active = ?", [1])
result.each { |row| puts row["name"] }
```

`fetch` returns an array of hashes. Column names are the keys. Combine raw SQL with ORM serialisation:

```
rows = Tina4.database.fetch(
  "SELECT u.*, COUNT(p.id) AS post_count FROM users u LEFT JOIN posts p ON p.user_id = u.id GROU
  []
)

rows.each do |row|
  puts "#{row["name"]}: #{row["post_count"]} posts"
end
```

When your raw query returns rows that match a model's shape, you can hydrate them via `from_hash`:

```
rows = Tina4.database.fetch("SELECT * FROM users WHERE active = ?", [1])
users = rows.map { |row| User.from_hash(row) }
users.each { |user| puts user.to_h }
```

`from_hash` applies `field_mapping` during hydration, so DB column names are translated to Ruby attribute names automatically.

---

16. 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
  .where("active = ?", [1])
  .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

1. Why a Query Builder?

Chapter 5 covered raw SQL. Chapter 6 covered the ORM. Both work. But raw SQL is error-prone for dynamic queries -- the more conditions you add, the more string concatenation you juggle. The ORM handles simple CRUD but steps aside for complex filtering, joins, and aggregations.

The QueryBuilder sits between the two. It gives you a fluent, chainable Ruby API that builds SQL for you. You describe *what* you want. It handles the syntax. No string concatenation. No misplaced commas. No forgotten WHERE keywords.

---

2. Creating a QueryBuilder

Every query starts with `Tina4::QueryBuilder.from`:

```
query = Tina4::QueryBuilder.from_table("users", db: db)
```

The first argument is the table name. The `db:` keyword argument is the database connection. If you omit `db:`, the QueryBuilder will fall back to `Tina4.database` when you execute the query -- but it will raise an error if no connection is available.

You can also access the QueryBuilder from an ORM model:

```
query = User.query
```

This creates a QueryBuilder pre-configured with the model's table name and the active database connection.

---

3. Selecting Columns

By default, the QueryBuilder selects all columns (`*`). Use `.select` to pick specific columns:

```
query = Tina4::QueryBuilder.from_table("users", db: db)
  .select("id", "name", "email")
```

Pass as many column names as you need. Each is a separate string argument:

```
query = Tina4::QueryBuilder.from_table("products", db: db)
  .select("name", "price", "category")
```

If you call `.select` with no arguments, the columns remain unchanged.

---

4. Filtering with where

Add conditions with `.where`. Use `?` placeholders for parameters:

```
results = Tina4::QueryBuilder.from_table("users", db: db)
  .where("active = ?", [1])
```

```
  .get
```

The first argument is the SQL condition. The second is an array of parameter values that replace the `?` placeholders. The database driver handles escaping. Your input never touches the SQL string directly.

Multiple where Calls

Chain multiple `.where` calls. They combine with AND:

```
results = Tina4::QueryBuilder.from_table("products", db: db)
  .where("price > ?", [10.00])
  .where("category = ?", ["Electronics"])
  .get
```

This generates:

```
SELECT * FROM products WHERE price > ? AND category = ?
```

The first `.where` produces the condition directly. Every subsequent `.where` prepends `AND`.

---

5. OR Conditions with or_where

Use `.or_where` when you need OR logic:

```
results = Tina4::QueryBuilder.from_table("products", db: db)
  .where("category = ?", ["Electronics"])
  .or_where("category = ?", ["Books"])
  .get
```

This generates:

```
SELECT * FROM products WHERE category = ? OR category = ?
```

You can mix `.where` and `.or_where` freely:

```
results = Tina4::QueryBuilder.from_table("products", db: db)
  .where("price > ?", [10.00])
  .where("in_stock = ?", [1])
  .or_where("featured = ?", [1])
  .get
```

This generates:

```
SELECT * FROM products WHERE price > ? AND in_stock = ? OR featured = ?
```

If you need grouped conditions (parentheses), write the group as a single condition string:

```
results = Tina4::QueryBuilder.from_table("products", db: db)
  .where("price > ?", [10.00])
  .where("(category = ? OR category = ?)", ["Electronics", "Books"])
  .get
```

---

6. Joins

Inner Join

```
results = Tina4::QueryBuilder.from_table("orders", db: db)
  .select("orders.id", "users.name", "orders.total")
  .join("users", "users.id = orders.user_id")
  .get
```

This generates:

```
SELECT orders.id, users.name, orders.total FROM orders INNER JOIN users ON users.id = orders.use
```

Left Join

```
results = Tina4::QueryBuilder.from_table("users", db: db)
  .select("users.name", "orders.total")
  .left_join("orders", "orders.user_id = users.id")
  .get
```

This generates:

```
SELECT users.name, orders.total FROM users LEFT JOIN orders ON orders.user_id = users.id
```

Multiple Joins

Chain as many joins as you need:

```
results = Tina4::QueryBuilder.from_table("order_items", db: db)
  .select("products.name", "order_items.quantity", "orders.status")
  .join("products", "products.id = order_items.product_id")
  .join("orders", "orders.id = order_items.order_id")
  .where("orders.status = ?", ["completed"])
  .get
```

---

7. Ordering

Use `.order_by` to sort results:

```
results = Tina4::QueryBuilder.from_table("products", db: db)
  .order_by("price ASC")
  .get
```

Pass the column name and direction as a single string. Chain multiple `.order_by` calls for multi-column sorting:

```
results = Tina4::QueryBuilder.from_table("products", db: db)
  .order_by("category ASC")
  .order_by("price DESC")
  .get
```

This generates:

```
SELECT * FROM products ORDER BY category ASC, price DESC
```

---

8. Limiting and Offsetting

Use `.limit` to cap the number of rows returned:

```
results = Tina4::QueryBuilder.from_table("products", db: db)
  .order_by("created_at DESC")
  .limit(10)
  .get
```

Pass a second argument for the offset:

```
# Skip 20 rows, return the next 10
results = Tina4::QueryBuilder.from_table("products", db: db)
  .order_by("created_at DESC")
  .limit(10, 20)
  .get
```

If you do not call `.limit`, the QueryBuilder defaults to 100 rows with an offset of 0 when you call `.get`. This prevents accidental full-table dumps.

---

9. Grouping and Having

Group By

```
results = Tina4::QueryBuilder.from_table("orders", db: db)
  .select("status", "COUNT(*) AS order_count")
  .group_by("status")
  .get
```

This generates:

```
SELECT status, COUNT(*) AS order_count FROM orders GROUP BY status
```

Having

Filter grouped results with `.having`:

```
results = Tina4::QueryBuilder.from_table("products", db: db)
  .select("category", "AVG(price) AS avg_price")
  .group_by("category")
  .having("AVG(price) > ?", [50.00])
  .get
```

This generates:

```
SELECT category, AVG(price) AS avg_price FROM products GROUP BY category HAVING AVG(price) > ?
```

Multiple `.having` calls combine with AND:

```
results = Tina4::QueryBuilder.from_table("products", db: db)
  .select("category", "COUNT(*) AS cnt", "AVG(price) AS avg_price")
  .group_by("category")
  .having("COUNT(*) > ?", [5])
  .having("AVG(price) < ?", [100.00])
  .get
```

---

10. Executing Queries

The QueryBuilder provides four execution methods.

get -- Multiple Rows

```
results = Tina4::QueryBuilder.from_table("users", db: db)
  .where("active = ?", [1])
  .order_by("name ASC")
  .limit(25)
  .get
```

Returns a `DatabaseResult` object. You can iterate over it, access rows by index, call `.to_json`, `.to_csv`, or `.to_paginate` -- everything covered in Chapter 5.

first -- Single Row

```
user = Tina4::QueryBuilder.from_table("users", db: db)
  .where("email = ?", ["alice@example.com"])
  .first
```

Returns a single hash, or `nil` if no row matches. Useful when you expect exactly one result.

count -- Row Count

```
total = Tina4::QueryBuilder.from_table("users", db: db)
  .where("active = ?", [1])
  .count
```

Returns an integer. The QueryBuilder rewrites your columns to `COUNT(*) AS cnt` internally, so it works regardless of what you passed to `.select`.

exists? -- Boolean Check

```
if Tina4::QueryBuilder.from_table("users", db: db)
     .where("email = ?", ["alice@example.com"])
     .exists?
  puts "User found"
end
```

Returns `true` if at least one matching row exists, `false` otherwise. Calls `.count` under the hood.

---

11. Inspecting the SQL with to_sql

Call `.to_sql` to see the generated SQL without executing it:

```
query = Tina4::QueryBuilder.from_table("products", db: db)
  .select("name", "price")
  .where("category = ?", ["Electronics"])
  .where("price > ?", [50.00])
  .order_by("price DESC")
  .limit(10)

puts query.to_sql
```

Output:

```
SELECT name, price FROM products WHERE category = ? AND price > ? ORDER BY price DESC
```

This is invaluable for debugging. If a query returns unexpected results, print the SQL and check the logic. Note that `.to_sql` shows the SQL with `?` placeholders -- the actual parameter values are bound at execution time by the database driver.

---

12. Using QueryBuilder in Routes

A Filtered API Endpoint

```
Tina4::Router.get("/api/products") do |request, response|
  db = Tina4.database

  query = Tina4::QueryBuilder.from_table("products", db: db)

  # Apply filters from query string
  category = request.params["category"]
  min_price = request.params["min_price"]
  max_price = request.params["max_price"]
  search = request.params["search"]

  query = query.where("category = ?", [category]) if category
  query = query.where("price >= ?", [min_price.to_f]) if min_price
  query = query.where("price <= ?", [max_price.to_f]) if max_price
  query = query.where("name LIKE ?", ["%#{search}%"]) if search

  # Pagination
  page = (request.params["page"] || 1).to_i
  per_page = (request.params["per_page"] || 20).to_i
  offset = (page - 1) * per_page

  total = query.count

  results = query
    .order_by("name ASC")
    .limit(per_page, offset)
    .get

  response.call({
    products: results,
    total: total,
    page: page,
    per_page: per_page
  }, Tina4::HTTP_OK)
end
```

```
curl "http://localhost:7147/api/products?category=Electronics&min_price=50&page=2&per_page=10"
```

A Dashboard Summary Endpoint

```
Tina4::Router.get("/api/dashboard/summary") do |request, response|
  db = Tina4.database

  # Total users
  total_users = Tina4::QueryBuilder.from_table("users", db: db).count

  # Active users
  active_users = Tina4::QueryBuilder.from_table("users", db: db)
    .where("active = ?", [1])
    .count

  # Revenue by category
  revenue = Tina4::QueryBuilder.from_table("order_items", db: db)
    .select("products.category", "SUM(order_items.quantity * order_items.price) AS revenue")
    .join("products", "products.id = order_items.product_id")
    .group_by("products.category")
    .order_by("revenue DESC")
    .get

  response.call({
    total_users: total_users,
    active_users: active_users,
    revenue_by_category: revenue
  }, Tina4::HTTP_OK)
end
```

---

13. Exercise: Build a Product Search API

Build a product search endpoint that uses the QueryBuilder for all database access.

Requirements

- Assume the `products` table from Chapter 5 exists with columns: `id`, `name`, `category`, `price`, `in_stock`, `created_at`.
- Build these API endpoints:

Method	Path	Description
`GET`	`/api/products/search`	Search products. Support `?q=`, `?category=`, `?min_price=`, `?max_price=`, `?in_stock=`, `?sort=`, `?page=`, `?per_page=`.
`GET`	`/api/products/stats`	Return category statistics: product count and average price per category. Only include categories with more than 2 products.

Test with:

```
# Full-text search with price range
curl "http://localhost:7147/api/products/search?q=wireless&min_price=20&max_price=100"

# Category filter with pagination
curl "http://localhost:7147/api/products/search?category=Electronics&page=1&per_page=5&sort=pric

# In-stock only
curl "http://localhost:7147/api/products/search?in_stock=1&sort=name_asc"

# Category stats
curl "http://localhost:7147/api/products/stats"
```

---

14. Solution

Search Endpoint

Create `src/routes/product_search.rb`:

```
Tina4::Router.get("/api/products/search") do |request, response|
  db = Tina4.database

  query = Tina4::QueryBuilder.from_table("products", db: db)

  # Text search
  q = request.params["q"]
  query = query.where("name LIKE ?", ["%#{q}%"]) if q && !q.empty?

  # Category filter
  category = request.params["category"]
  query = query.where("category = ?", [category]) if category && !category.empty?

  # Price range
  min_price = request.params["min_price"]
  query = query.where("price >= ?", [min_price.to_f]) if min_price

  max_price = request.params["max_price"]
  query = query.where("price <= ?", [max_price.to_f]) if max_price

  # In-stock filter
  in_stock = request.params["in_stock"]
  query = query.where("in_stock = ?", [in_stock.to_i]) if in_stock

  # Total count before pagination
  total = query.count

  # Sorting
  sort = request.params["sort"] || "name_asc"
  order = case sort
          when "price_asc"  then "price ASC"
          when "price_desc" then "price DESC"
          when "name_desc"  then "name DESC"
          when "newest"     then "created_at DESC"
          else "name ASC"
          end
```

```
  # Pagination
  page = (request.params["page"] || 1).to_i
  per_page = (request.params["per_page"] || 20).to_i
  offset = (page - 1) * per_page

  results = query
    .order_by(order)
    .limit(per_page, offset)
    .get

  response.call({
    products: results,
    total: total,
    page: page,
    per_page: per_page,
    sort: sort
  }, Tina4::HTTP_OK)
end
```

Stats Endpoint

```
Tina4::Router.get("/api/products/stats") do |request, response|
  db = Tina4.database

  stats = Tina4::QueryBuilder.from_table("products", db: db)
    .select("category", "COUNT(*) AS product_count", "ROUND(AVG(price), 2) AS avg_price")
    .group_by("category")
    .having("COUNT(*) > ?", [2])
    .order_by("product_count DESC")
    .get

  response.call({
    categories: stats
  }, Tina4::HTTP_OK)
end
```

Expected output for search:

```
{
  "products": [
    {"id": 5, "name": "Wireless Keyboard", "category": "Electronics", "price": 79.99, "in_stock"
    {"id": 8, "name": "Wireless Mouse", "category": "Electronics", "price": 34.99, "in_stock": 1
  ],
  "total": 2,
  "page": 1,
  "per_page": 20,
  "sort": "name_asc"
}
```

Expected output for stats:

```
{
  "categories": [
    {"category": "Electronics", "product_count": 12, "avg_price": 149.99},
    {"category": "Books", "product_count": 8, "avg_price": 24.50},
    {"category": "Home", "product_count": 5, "avg_price": 67.25}
  ]
}
```

---

15. NoSQL: MongoDB Queries

The QueryBuilder can generate MongoDB-compatible query documents with `to_mongo`. This returns a hash containing the filter, projection, sort, limit, and skip -- ready to pass to the Mongo Ruby 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

```
query = Tina4::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 hash:

```
{
  filter: { "age" => { "$gt" => 25 }, "status" => "active" },
  projection: { "name" => 1, "email" => 1 },
  sort: { "name" => 1 },
  limit: 10,
  skip: 5,
}
```

Pass it directly to the Mongo Ruby driver:

```
collection = client[:users]
cursor = collection.find(mongo[:filter])
  .projection(mongo[:projection])
  .sort(mongo[:sort])
```

```
    .limit(mongo[:limit])
    .skip(mongo[:skip])
```

---

16. Gotchas

1. Default Limit of 100

Problem: You expect all rows but only get 100.

Cause: When you call `.get` without calling `.limit`, the QueryBuilder defaults to `limit: 100, offset: 0`. This is a safety net to prevent accidental full-table dumps.

Fix: Call `.limit` explicitly if you need more rows: `.limit(500)`. For truly unbounded queries, use raw SQL via `db.fetch`.

2. Parameter Order Matters

Problem: Your query returns wrong results or the database throws a type error.

Cause: The `?` placeholders are positional. Parameters from `.where`, `.or_where`, and `.having` are collected in the order you chain them. If you reorder your calls, the parameter positions change.

Fix: Keep your chained calls in a consistent order. Use `.to_sql` to verify the generated SQL matches your parameter array.

3. No Database Connection

Problem: You get `QueryBuilder: No database connection provided.`

Cause: You did not pass `db:` to `.from`, and `Tina4.database` is not configured.

Fix: Either pass the connection explicitly with `Tina4::QueryBuilder.from_table("users", db: db)` or ensure your `.env` has a valid `TINA4_DATABASE_URL` so the framework configures `Tina4.database` at startup.

4. count Replaces Your Columns

Problem: You call `.count` after setting `.select("name", "price")` and get confused about what SQL runs.

Cause: `.count` temporarily replaces your columns with `COUNT(*) AS cnt`, executes the query, then restores your original columns. This is transparent -- your QueryBuilder object is unchanged after the call.

Fix: No action needed. This is by design. If you need both the count and the rows, call `.count` first and then `.get`. The QueryBuilder is reusable.

5. to_sql Does Not Show Parameter Values

Problem: You call `.to_sql` and see `?` placeholders instead of actual values.

Cause: `.to_sql` returns the SQL template. The parameter values are bound separately by the database driver at execution time. This is correct behaviour -- it is how parameterised queries work.

Fix: To debug parameter values, inspect the arrays you pass to `.where` and `.having`. The parameters are applied in order, left to right, matching the `?` placeholders in the SQL output.

6. or_where as First Condition

Problem: Your SQL starts with `WHERE OR condition` and the database rejects it.

Cause: You used `.or_where` as the first filter. The QueryBuilder prepends OR to every `.or_where` condition except when it is the first condition in the list -- in that case, the connector is dropped. However, the intent is unclear. Starting a WHERE clause with OR is semantically wrong.

Fix: Always start with `.where` for the first condition. Use `.or_where` only for subsequent conditions that need OR logic.

Authentication

1. Locking the Door

Every endpoint you have built is public. Anyone with the URL can read, create, update, and delete data. Fine for a tutorial. Reckless for production. A real application needs to know who is making a request and whether they have permission.

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 containing a payload -- user ID, role, whatever you need. The server mints the token at login. The client sends it with every request. The server verifies it without touching the database.

Generating a Token

```
payload = {
  user_id: 42,
  email: "alice@example.com",
  role: "admin"
}

token = Tina4::Auth.get_token(payload)
```

`get_token` signs the payload 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.

> ***Legacy aliases:*** `Tina4::Auth.create_token` *still works as an alias for* `Tina4::Auth.get_token`*. Use the primary name in new code.*

Token Expiry

By default, tokens expire after 60 minutes. Configure the expiry when generating the token:

```
# Default: 60 minutes
token = Tina4::Auth.get_token(payload)

# Custom: 24 hours (1440 minutes)
token = Tina4::Auth.get_token(payload, expires_in: 1440)

# Custom: 7 days (10080 minutes)
token = Tina4::Auth.get_token(payload, expires_in: 10080)
```

The `expires_in` 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 = Tina4::Auth.valid_token(token)
# Returns the payload hash if valid, nil 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 `nil` if the token is invalid or expired.

> ***Legacy alias:*** `Tina4::Auth.validate_token` *works the same way.*

Reading the Payload

```
payload = Tina4::Auth.get_payload(token)
```

Returns the decoded payload hash **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 `nil`.

> ***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 Ruby supports two JWT algorithms, auto-detected based on your configuration:

- **HS256** (HMAC-SHA256) -- set `TINA4_SECRET` in `.env`. Uses the standard library. Zero dependencies.
- **RS256** (RSA) -- RSA keys are auto-generated in the `.keys/` folder. Requires the `jwt` gem (included by default).

```
# .env -- HS256 mode (recommended, simplest setup)
TINA4_SECRET=my-super-secret-key-at-least-32-chars
```

If `TINA4_SECRET` is set and no RSA keys exist in `.keys/`, Tina4 uses HS256. If RSA keys exist in `.keys/` instead, Tina4 uses RS256. If neither is configured, Tina4 auto-generates RSA keys in `.keys/` on first run.

Keep this key secret. If someone gets it, they can forge tokens.

---

3. Password Hashing

Plain text passwords are a security breach waiting to happen. Tina4 provides two methods for secure password handling:

Hashing a Password

```
hash = Tina4::Auth.hash_password("my-secure-password")
# Returns: "$2a$12$abc123...long-hash-string..."
```

This uses BCrypt (via the `bcrypt` gem, included by default). Each hash includes a random salt, so hashing the same password twice produces different results.

Checking a Password

```
is_correct = Tina4::Auth.check_password("my-secure-password", stored_hash)
# Returns true if the password matches the hash
```

Registration Example

```
# @noauth
Tina4::Router.post("/api/register") do |request, response|
  body = request.body

  # Validate input
  if body["name"].nil? || body["email"].nil? || body["password"].nil?
    return response.json({ error: "Name, email, and password are required" }, 400)
  end

  if body["password"].length < 8
    return response.json({ error: "Password must be at least 8 characters" }, 400)
  end

  db = Tina4.database

  # Check if email already exists
  existing = db.fetch_one("SELECT id FROM users WHERE email = ?", [body["email"]])
  unless existing.nil?
    return response.json({ error: "Email already registered" }, 409)
  end

  # Hash the password
  password_hash = Tina4::Auth.hash_password(body["password"])

  # Create the user
  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 = last_insert_rowid()")

  response.json({
    message: "Registration successful",
    user: user
  }, 201)
end
```

```
curl -X POST http://localhost:7147/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

Here is the complete login flow: the client sends credentials, the server validates them, and returns a JWT token.

```
# @noauth
Tina4::Router.post("/api/login") do |request, response|
  body = request.body

  if body["email"].nil? || body["password"].nil?
    return response.json({ error: "Email and password are required" }, 400)
  end

  db = Tina4.database

  # Find the user
  user = db.fetch_one(
    "SELECT id, name, email, password_hash FROM users WHERE email = ?",
    [body["email"]]
  )

  if user.nil?
    return response.json({ error: "Invalid email or password" }, 401)
  end

  # Check the password
  unless Tina4::Auth.check_password(body["password"], user["password_hash"])
    return response.json({ error: "Invalid email or password" }, 401)
  end

  # Generate a token
  token = Tina4::Auth.get_token({
    user_id: user["id"],
    email: user["email"],
    name: user["name"]
  })

  response.json({
    message: "Login successful",
    token: token,
    user: {
      id: user["id"],
      name: user["name"],
      email: user["email"]
    }
  })
end
```

Notice the `@noauth` comment. The login endpoint must be public -- you cannot require a token to get a token.

```
curl -X POST http://localhost:7147/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:7147/api/profile \
  -H "Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9..."
```

---

6. Protecting Routes

Auth Middleware

Create a reusable auth middleware:

```
def auth_middleware(request, response, next_handler)
  auth_header = request.headers["Authorization"] || ""

  if auth_header.empty? || !auth_header.start_with?("Bearer ")
    return response.json({ error: "Authorization header required" }, 401)
  end

  token = auth_header.sub("Bearer ", "")

  payload = Tina4::Auth.valid_token(token)
  if payload.nil?
    return response.json({ error: "Invalid or expired token" }, 401)
  end

  request.user = payload  # Attach user data to the request

  next_handler.call(request, response)
end
```

Applying Middleware to Routes

```
Tina4::Router.get("/api/profile", middleware: "auth_middleware") do |request, response|
  response.json({
    user_id: request.user["user_id"],
    email: request.user["email"],
    name: request.user["name"]
  })
end
```

```
# Without token -- 401
curl http://localhost:7147/api/profile
```

```
{"error":"Authorization header required"}
```

```
# With valid token -- 200
curl http://localhost:7147/api/profile \
  -H "Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9..."
```

```
{
  "user_id": 1,
  "email": "alice@example.com",
  "name": "Alice"
}
```

Applying Middleware to a Group

```
Tina4::Router.group("/api/admin", middleware: "auth_middleware") do

  Tina4::Router.get("/stats") do |request, response|
    response.json({ active_users: 42 })
  end

  Tina4::Router.get("/logs") do |request, response|
    response.json({ logs: [] })
  end

end
```

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:

```
# @noauth
Tina4::Router.get("/api/public/health") do |request, response|
  response.json({ status: "ok" })
end

# @noauth
Tina4::Router.post("/api/login") do |request, response|
  # Login logic
end
```

```
# @noauth
Tina4::Router.post("/api/register") do |request, response|
  # Registration logic
end
```

@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:

```
# @secured
Tina4::Router.get("/api/me") do |request, response|
  # This GET route requires authentication
  response.json(request.user)
end
```

Role-Based Authorization

Combine auth middleware with role checks:

```
def require_role(role)
  lambda do |request, response, next_handler|
    # First check authentication
    auth_header = request.headers["Authorization"] || ""
    if auth_header.empty? || !auth_header.start_with?("Bearer ")
      return response.json({ error: "Authorization required" }, 401)
    end

    token = auth_header.sub("Bearer ", "")
    payload = Tina4::Auth.valid_token(token)
    if payload.nil?
      return response.json({ error: "Invalid or expired token" }, 401)
    end

    # Check role
    if (payload["role"] || "") != role
      return response.json({ error: "Forbidden. Required role: #{role}" }, 403)
    end

    request.user = payload
    next_handler.call(request, response)
  end
end
```

---

8. CSRF Protection

For traditional form-based applications (not SPAs), Tina4 provides CSRF protection with 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:

```
Tina4::Router.post("/profile/update") do |request, response|
  # Validate CSRF token
  unless Tina4::Auth.validate_form_token(request.body["_token"] || "")
    return response.json({ error: "Invalid form token. Please refresh and try again." }, 403)
  end

  # Process the form...
  response.redirect("/profile")
end
```

---

9. Sessions

Tina4 supports server-side sessions for storing per-user state between requests. Sessions work alongside JWT tokens -- use JWTs for API authentication and sessions for stateful web pages.

Using Sessions

Access session data via `request.session`:

```
Tina4::Router.post("/login-form") do |request, response|
  # After validating credentials...
  request.session["user_id"] = 42
  request.session["user_name"] = "Alice"
  request.session["logged_in"] = true

  response.redirect("/dashboard")
end

Tina4::Router.get("/dashboard") do |request, response|
  if request.session["logged_in"].nil?
    return response.redirect("/login")
  end

  response.render("dashboard.html", {
    user_name: request.session["user_name"]
  })
end

Tina4::Router.post("/logout") do |request, response|
  # Clear all session data
  request.session.clear
```

```
  response.redirect("/login")
end
```

---

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:7147/api/register \
  -H "Content-Type: application/json" \
  -d '{"name": "Alice", "email": "alice@example.com", "password": "securePass123"}'

# Login
curl -X POST http://localhost:7147/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:7147/api/profile \
  -H "Authorization: Bearer YOUR_TOKEN_HERE"

# Update profile
curl -X PUT http://localhost:7147/api/profile \
  -H "Authorization: Bearer YOUR_TOKEN_HERE" \
  -H "Content-Type: application/json" \
  -d '{"name": "Alice Smith"}'

# Change password
curl -X PUT http://localhost:7147/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:7147/api/profile
```

---

11. Solution

Migration

Create `migrations/20260322160000_create_auth_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;
```

Middleware

Create `src/routes/middleware.rb`:

```
def auth_middleware(request, response, next_handler)
```

```
  auth_header = request.headers["Authorization"] || ""

  if auth_header.empty? || !auth_header.start_with?("Bearer ")
    return response.json({ error: "Authorization required. Send: Authorization: Bearer <token>"
  end

  token = auth_header.sub("Bearer ", "")

  payload = Tina4::Auth.valid_token(token)
  if payload.nil?
    return response.json({ error: "Invalid or expired token. Please login again." }, 401)
  end

  request.user = payload

  next_handler.call(request, response)
end
```

Routes

Create `src/routes/auth.rb`:

```
# @noauth
Tina4::Router.post("/api/register") do |request, response|
  body = request.body

  errors = []
  errors << "Name is required" if body["name"].nil? || body["name"].empty?
  errors << "Email is required" if body["email"].nil? || body["email"].empty?
  if body["password"].nil? || body["password"].empty?
    errors << "Password is required"
  elsif body["password"].length < 8
    errors << "Password must be at least 8 characters"
  end

  unless errors.empty?
    return response.json({ errors: errors }, 400)
  end

  db = Tina4.database

  existing = db.fetch_one("SELECT id FROM users WHERE email = ?", [body["email"]])
  unless existing.nil?
    return response.json({ error: "Email already registered" }, 409)
  end

  hash = Tina4::Auth.hash_password(body["password"])

  db.execute(
    "INSERT INTO users (name, email, password_hash) VALUES (?, ?, ?)",
    [body["name"], body["email"], hash]
  )

  user = db.fetch_one("SELECT id, name, email, role, created_at FROM users WHERE id = last_inser

  response.json({ message: "Registration successful", user: user }, 201)
end

# @noauth
```

```
Tina4::Router.post("/api/login") do |request, response|
  body = request.body

  if body["email"].nil? || body["password"].nil?
    return response.json({ error: "Email and password are required" }, 400)
  end

  db = Tina4.database

  user = db.fetch_one(
    "SELECT id, name, email, password_hash, role FROM users WHERE email = ?",
    [body["email"]]
  )

  if user.nil? || !Tina4::Auth.check_password(body["password"], user["password_hash"])
    return response.json({ error: "Invalid email or password" }, 401)
  end

  token = Tina4::Auth.get_token({
    user_id: user["id"],
    email: user["email"],
    name: user["name"],
    role: user["role"]
  })

  response.json({
    message: "Login successful",
    token: token,
    user: { id: user["id"], name: user["name"], email: user["email"], role: user["role"] }
  })
end

# Get current user profile
Tina4::Router.get("/api/profile", middleware: "auth_middleware") do |request, response|
  db = Tina4.database

  user = db.fetch_one(
    "SELECT id, name, email, role, created_at FROM users WHERE id = ?",
    [request.user["user_id"]]
  )

  if user.nil?
    return response.json({ error: "User not found" }, 404)
  end

  response.json(user)
end

# Update profile
Tina4::Router.put("/api/profile", middleware: "auth_middleware") do |request, response|
  db = Tina4.database
  body = request.body
  user_id = request.user["user_id"]

  if body["email"]
    existing = db.fetch_one(
      "SELECT id FROM users WHERE email = ? AND id != ?",
      [body["email"], user_id]
    )
```

```
    unless existing.nil?
      return response.json({ error: "Email already in use by another account" }, 409)
    end
  end

  current = db.fetch_one("SELECT * FROM users WHERE id = ?", [user_id])

  db.execute(
    "UPDATE users SET name = ?, email = ? WHERE id = ?",
    [body["name"] || current["name"], body["email"] || current["email"], user_id]
  )

  updated = db.fetch_one(
    "SELECT id, name, email, role, created_at FROM users WHERE id = ?",
    [user_id]
  )

  response.json({ message: "Profile updated", user: updated })
end

# Change password
Tina4::Router.put("/api/profile/password", middleware: "auth_middleware") do |request, response|
  db = Tina4.database
  body = request.body
  user_id = request.user["user_id"]

  if body["current_password"].nil? || body["new_password"].nil?
    return response.json({ error: "Current password and new password are required" }, 400)
  end

  if body["new_password"].length < 8
    return response.json({ error: "New password must be at least 8 characters" }, 400)
  end

  user = db.fetch_one("SELECT password_hash FROM users WHERE id = ?", [user_id])

  unless Tina4::Auth.check_password(body["current_password"], user["password_hash"])
    return response.json({ error: "Current password is incorrect" }, 401)
  end

  new_hash = Tina4::Auth.hash_password(body["new_password"])

  db.execute(
    "UPDATE users SET password_hash = ? WHERE id = ?",
    [new_hash, user_id]
  )

  response.json({ message: "Password changed successfully" })
end
```

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`)

Expected output for successful password change:

```
{"message":"Password changed successfully"}
```

---

12. Gotchas

1. Token Expiry Confusion

Problem: Tokens that worked yesterday now return 401.

Cause: The default token lifetime is 1 hour (3600 seconds). 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 RSA keys in `.keys/`. Or the key files were 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` annotation to bypass it.

Fix: Add `# @noauth` before 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: BCrypt/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 & Cookies

1. State in a Stateless World

Your e-commerce site needs a shopping cart that persists across page loads. A language preference that sticks. Flash messages after form submissions. But HTTP is stateless. Every request arrives with no memory of what came before. Sessions and cookies give the server a way to remember.

Chapter 8 introduced sessions for authentication. This chapter goes deeper. Session backends. Flash messages. Cookies. Remember-me tokens. Security configuration.

---

2. How Sessions Work

Sessions are auto-started. Every route handler receives `request.session` ready to use. No manual setup required.

When a user visits your site for the first time, Tina4 generates a unique session ID (a long random string), stores it in a cookie named `tina4_session` (`HttpOnly`, `SameSite=Lax`) on the user's browser, and creates a server-side storage entry keyed by that ID. On every subsequent request, the browser sends the cookie, Tina4 looks up the session data, and makes it available via `request.session`.

The flow looks like this:

- Browser sends first request (no session cookie)
- Tina4 generates session ID: `abc123def456`
- Tina4 sets cookie: `tina4_session=abc123def456`
- Tina4 creates empty session storage for `abc123def456`
- Browser sends second request with cookie `tina4_session=abc123def456`
- Tina4 loads session data for `abc123def456`
- Your route handler reads and writes `request.session`
- At the end of the request, Tina4 saves the updated session data

The session data is stored server-side. The browser only has the session ID -- it never sees the actual data.

---

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 hash

---

4. File Sessions (Default)

Out of the box, Tina4 stores sessions in files. No configuration needed.

```
Tina4::Router.get("/visit-counter") do |request, response|
  count = (request.session.get("visit_count", 0)) + 1
  request.session.set("visit_count", count)

  response.json({
    visit_count: count,
    message: "You have visited this page #{count} time#{count == 1 ? '' : 's'}"
  })
end
```

```
curl http://localhost:7147/visit-counter -c cookies.txt -b cookies.txt
```

```
{"visit_count":1,"message":"You have visited this page 1 time"}
```

```
curl http://localhost:7147/visit-counter -c cookies.txt -b cookies.txt
```

```
{"visit_count":2,"message":"You have visited this page 2 times"}
```

The `-c cookies.txt` flag tells curl to save cookies to a file, and `-b cookies.txt` tells it to send them back. This simulates how a browser works.

---

5. Redis Sessions

For production deployments with multiple servers (behind a load balancer), you need a shared session store. Redis is the most common choice.

```
TINA4_SESSION_BACKEND=redis
TINA4_SESSION_REDIS_HOST=localhost
TINA4_SESSION_REDIS_PORT=6379
TINA4_SESSION_REDIS_PASSWORD=your-redis-password
```

That is the only change. Your code stays exactly the same. `request.session` works identically whether sessions are stored in files, Redis, MongoDB, or Valkey.

---

6. MongoDB Sessions

```
TINA4_SESSION_BACKEND=mongodb
TINA4_SESSION_REDIS_HOST=localhost
TINA4_SESSION_REDIS_PORT=27017
TINA4_SESSION_MONGO_DB=myapp
TINA4_SESSION_MONGO_COLLECTION=sessions
```

---

7. Valkey Sessions

```
TINA4_SESSION_BACKEND=valkey
TINA4_SESSION_REDIS_HOST=localhost
TINA4_SESSION_REDIS_PORT=6379
```

---

8. 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. Works with all 5 database engines (SQLite, PostgreSQL, MySQL, MSSQL, Firebird).

---

9. Reading and Writing Session Data

Session data is a simple key-value store. You read and write it through `request.session`:

```
# Write to session
Tina4::Router.post("/api/preferences") do |request, response|
  body = request.body

  request.session.set("language", body["language"] || "en")
  request.session.set("theme", body["theme"] || "light")
  request.session.set("items_per_page", (body["items_per_page"] || 20).to_i)

  response.json({
```

```
    message: "Preferences saved",
    preferences: {
      language: request.session.get("language"),
      theme: request.session.get("theme"),
      items_per_page: request.session.get("items_per_page")
    }
  })
end

# Read from session
Tina4::Router.get("/api/preferences") do |request, response|
  response.json({
    language: request.session.get("language", "en"),
    theme: request.session.get("theme", "light"),
    items_per_page: request.session.get("items_per_page", 20)
  })
end

# Delete a specific key
Tina4::Router.delete("/api/preferences/{key}") do |request, response|
  key = request.params["key"]
  request.session.delete(key)

  response.json({ message: "Preference '#{key}' removed" })
end

# Clear all session data
Tina4::Router.post("/api/session/clear") do |request, response|
  request.session.clear

  response.json({ message: "Session cleared" })
end
```

Storing Complex Data

Sessions can hold arrays and nested structures:

```
Tina4::Router.post("/api/cart/add") do |request, response|
  body = request.body

  cart = request.session.get("cart", [])

  cart << {
    product_id: body["product_id"].to_i,
    name: body["name"],
    price: body["price"].to_f,
    quantity: (body["quantity"] || 1).to_i,
    added_at: Time.now.iso8601
  }

  request.session.set("cart", cart)

  total = cart.sum { |item| item[:price] * item[:quantity] }

  response.json({
    message: "Added to cart",
    cart_items: cart.length,
    cart_total: total
  })
```

```
end
```

---

10. Flash Messages

Flash messages are session data that lives for one request. Set a flash before redirecting. The next request reads it. Then it vanishes.

Setting a Flash Message

```
Tina4::Router.post("/profile/update") do |request, response|
  body = request.body

  # Update the profile (database logic here)

  request.session.flash("message", "Profile updated successfully")
  request.session.flash("message_type", "success")

  response.redirect("/profile")
end
```

Reading and Clearing Flash Messages

```
Tina4::Router.get("/profile") do |request, response|
  flash_message = request.session.flash("message")
  flash_type = request.session.flash("message_type") || "info"

  response.render("profile.html", {
    user: { name: "Alice", email: "alice@example.com" },
    flash_message: flash_message,
    flash_type: flash_type
  })
end
```

Calling `request.session.flash(key)` with only a key reads the value and removes it in one step.

Using Flash Messages in Templates

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

{% block content %}
    {% if flash_message %}
        <div class="alert alert-{{ flash_type }}">
            {{ flash_message }}
        </div>
    {% endif %}

    <h1>Profile</h1>
    <p>Name: {{ user.name }}</p>
    <p>Email: {{ user.email }}</p>
{% endblock %}
```

---

11. Setting and Reading Cookies

Cookies are small pieces of data stored in the browser. Unlike sessions, the data is stored client-side.

Setting a Cookie

```
Tina4::Router.post("/api/set-language") do |request, response|
  language = request.body["language"] || "en"

  response.cookie("language", language, {
    expires: Time.now + (365 * 24 * 60 * 60),  # 1 year
    path: "/",
    http_only: false,
    secure: false,
    same_site: "Lax"
  })

  response.json({ message: "Language set to #{language}" })
end
```

Reading a Cookie

```
Tina4::Router.get("/api/get-language") do |request, response|
  language = request.cookies["language"] || "en"

  response.json({ language: language })
end
```

Deleting a Cookie

```
Tina4::Router.post("/api/clear-language") do |request, response|
  response.cookie("language", "", {
    expires: Time.now - 3600,
    path: "/"
  })

  response.json({ message: "Language cookie cleared" })
end
```

When to Use Cookies vs Sessions

Use Cookies For	Use Sessions For
Language preference	Shopping cart contents
Theme preference (light/dark)	User authentication state
"Remember this device" flag	Flash messages
Analytics tracking consent	Form wizard progress
Non-sensitive, long-lived data	Sensitive, short-lived data

---

12. Remember Me Functionality

The "remember me" pattern uses a long-lived cookie to re-authenticate users after their session expires.

```
# @noauth
Tina4::Router.post("/login") do |request, response|
  body = request.body
  db = Tina4.database

  user = db.fetch_one(
    "SELECT id, name, email, password_hash FROM users WHERE email = ?",
    [body["email"]]
  )

  if user.nil? || !Tina4::Auth.check_password(body["password"], user["password_hash"])
    return response.json({ error: "Invalid email or password" }, 401)
  end

  request.session.set("user_id", user["id"])
  request.session.set("user_name", user["name"])

  if body["remember_me"]
    remember_token = SecureRandom.hex(32)

    db.execute(
      "UPDATE users SET remember_token = ? WHERE id = ?",
      [Digest::SHA256.hexdigest(remember_token), user["id"]]
    )

    response.cookie("remember_me", remember_token, {
      expires: Time.now + (30 * 24 * 60 * 60),  # 30 days
      path: "/",
      http_only: true,
      secure: true,
      same_site: "Lax"
    })
  end

  response.json({
    message: "Login successful",
    user: { id: user["id"], name: user["name"] }
  })
end
```

---

13. Session Security

Configuration Options

```
TINA4_SESSION_TTL=3600
TINA4_SESSION_SAMESITE=Lax
```

`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.

Session Regeneration

After a user logs in, regenerate the session ID to prevent session fixation attacks:

```
Tina4::Router.post("/login") do |request, response|
  # Validate credentials...

  request.session.regenerate

  request.session.set("user_id", user["id"])

  response.redirect("/dashboard")
end
```

Destroy a Session

To completely destroy a session (not just clear its data):

```
Tina4::Router.post("/logout") do |request, response|
  request.session.destroy
  response.redirect("/login")
end
```

---

14. Exercise: Build a Shopping Cart with Session Storage

Build a shopping cart that stores items in the session. No database needed -- the cart lives entirely in session data.

Requirements

Method	Path	Description
`POST`	`/api/cart/add`	Add an item to the cart. Body: `{"product_id": 1, "name": "Widget", "price": 9.99, "quantity": 2}`
`GET`	`/api/cart`	View the cart. Show items, quantities, item subtotals, and cart total.
`PUT`	`/api/cart/{product_id:int}`	Update quantity. Body: `{"quantity": 3}`. Remove item if quantity is 0.
`DELETE`	`/api/cart/{product_id:int}`	Remove an item from the cart.
`DELETE`	`/api/cart`	Clear the entire cart.

Business Rules

- If adding a product that already exists in the cart, increment the quantity instead of adding a duplicate
- Cart total should be calculated dynamically
- Return the full cart state after every operation

Test with:

```
# Add first item
curl -X POST http://localhost:7147/api/cart/add \
  -H "Content-Type: application/json" \
  -d '{"product_id": 1, "name": "Wireless Keyboard", "price": 79.99, "quantity": 1}' \
  -c cookies.txt -b cookies.txt

# Add second item
curl -X POST http://localhost:7147/api/cart/add \
  -H "Content-Type: application/json" \
  -d '{"product_id": 2, "name": "USB-C Hub", "price": 49.99, "quantity": 2}' \
  -c cookies.txt -b cookies.txt

# View cart
curl http://localhost:7147/api/cart -b cookies.txt

# Clear cart
curl -X DELETE http://localhost:7147/api/cart -b cookies.txt -c cookies.txt
```

---

15. Solution

Create `src/routes/cart.rb`:

```
def get_cart(session)
  session.get("cart", [])
end

def cart_response(cart)
  total = 0.0
  item_count = 0
  items = []

  cart.each do |item|
    subtotal = item[:price] * item[:quantity]
    total += subtotal
    item_count += item[:quantity]
    items << item.merge(subtotal: subtotal)
  end

  {
    items: items,
    item_count: item_count,
    unique_items: cart.length,
    total: total.round(2)
  }
end

# Add item to cart
Tina4::Router.post("/api/cart/add") do |request, response|
  body = request.body

  if body["product_id"].nil? || body["name"].nil? || body["price"].nil?
    return response.json({ error: "product_id, name, and price are required" }, 400)
  end

  cart = get_cart(request.session)
  product_id = body["product_id"].to_i
  quantity = (body["quantity"] || 1).to_i

  existing = cart.find { |item| item[:product_id] == product_id }

  if existing
    existing[:quantity] += quantity
  else
    cart << {
      product_id: product_id,
      name: body["name"],
      price: body["price"].to_f,
      quantity: quantity
    }
  end

  request.session.set("cart", cart)

  response.json(cart_response(cart))
end

# View cart
```

```
Tina4::Router.get("/api/cart") do |request, response|
  cart = get_cart(request.session)
  response.json(cart_response(cart))
end

# Update quantity
Tina4::Router.put("/api/cart/{product_id:int}") do |request, response|
  product_id = request.params["product_id"]
  quantity = (request.body["quantity"] || 0).to_i
  cart = get_cart(request.session)

  index = cart.index { |item| item[:product_id] == product_id }

  if index.nil?
    return response.json({ error: "Product not in cart" }, 404)
  end

  if quantity <= 0
    cart.delete_at(index)
  else
    cart[index][:quantity] = quantity
  end

  request.session.set("cart", cart)

  response.json(cart_response(cart))
end

# Remove item
Tina4::Router.delete("/api/cart/{product_id:int}") do |request, response|
  product_id = request.params["product_id"]
  cart = get_cart(request.session)

  index = cart.index { |item| item[:product_id] == product_id }

  if index.nil?
    return response.json({ error: "Product not in cart" }, 404)
  end

  cart.delete_at(index)
  request.session.set("cart", cart)

  response.json(cart_response(cart))
end

# Clear cart
Tina4::Router.delete("/api/cart") do |request, response|
  request.session.set("cart", [])

  response.json(cart_response([]))
end
```

Expected output after adding two items:

```
{
  "items": [
    {"product_id": 1, "name": "Wireless Keyboard", "price": 79.99, "quantity": 1, "subtotal": 79
    {"product_id": 2, "name": "USB-C Hub", "price": 49.99, "quantity": 2, "subtotal": 99.98}
  ],
```

```
   "item_count": 3,
   "unique_items": 2,
   "total": 179.97
 }
```

---

16. Gotchas

1. Sessions Do Not Work with curl Without Cookie Flags

Problem: Each curl request sees an empty session, as if it is a new user.

Cause: curl does not automatically save or send cookies.

Fix: Use `-c cookies.txt -b cookies.txt` with curl.

2. Session Data Disappears After Server Restart

Problem: All session data is gone after restarting the dev server.

Cause: File sessions stored in the system temp directory may be cleared on restart.

Fix: Set `TINA4_SESSION_PATH` to a persistent directory. For production, use Redis or Valkey.

3. Session Cookie Not Sent Over HTTP in Production

Problem: Sessions work locally but not in production.

Cause: `TINA4_SESSION_SECURE=true` means the cookie is only sent over HTTPS.

Fix: Ensure your reverse proxy sets `X-Forwarded-Proto: https`, or set `TINA4_SESSION_SECURE=false` for HTTP.

4. Flash Messages Show Twice

Problem: The flash message appears, then appears again on the next page load.

Cause: You read the flash message but did not clear it from the session.

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.

5. Large Session Data Causes Slow Requests

Problem: Pages load slowly and performance degrades over time.

Cause: You are storing large amounts of data in the session.

Fix: Keep session data small. Store IDs and references, not entire objects.

6. Remember Me Token Not Invalidated on Password Change

Problem: After a user changes their password, their "remember me" cookies on other devices still work.

Cause: The remember-me token in the database was not cleared when the password changed.

Fix: Clear the `remember_token` column whenever the password is updated.

7. Session Fixation

Problem: An attacker can hijack a user's session by setting a known session ID before the user logs in.

Cause: The session ID is not regenerated after login.

Fix: Call `request.session.regenerate` after successful login.

Middleware

1. The Gatekeepers

Your API needs CORS headers for the React frontend. Rate limiting for public endpoints. Auth checking for admin routes. All without cluttering your handlers.

You could copy-paste 10 lines of CORS code into every handler. That breaks the moment you forget one. You could pile all the checks into a giant `if` tree. That buries business logic under boilerplate.

Middleware solves this. Wrap routes with reusable logic that runs before or after the handler. Each middleware does one job -- check a token, set CORS headers, log the request, enforce rate limits -- then passes control to the next layer. Route handlers stay focused on their actual purpose.

Chapter 2 introduced middleware. This chapter goes deep. Built-in middleware. Custom middleware. Execution order. Short-circuiting. Real-world patterns.

---

2. What Middleware Is

Middleware is code that runs before or after your route handler. It sits in the HTTP pipeline between the incoming request and the response. Every request can pass through multiple middleware layers before reaching the handler.

Tina4 Ruby supports two styles of middleware:

Method-based middleware receives the request, the response, and a `next_handler` callable. Call `next_handler` to continue. Skip it to short-circuit.

```
def passthrough(request, response, next_handler)
  next_handler.call(request, response)
end

def block_everything(request, response, next_handler)
  response.json({ error: "Service unavailable" }, 503)
end
```

Class-based middleware uses naming conventions. Class methods prefixed with `before_` run before the handler. Methods prefixed with `after_` run after it. Each method receives `(request, response)` and returns `[request, response]`.

```
class MyMiddleware
  class << self
    def before_check(request, response)
      # Runs before the route handler
      [request, response]
    end

    def after_cleanup(request, response)
      # Runs after the route handler
      [request, response]
    end
```

```
  end
end
```

Register class-based middleware globally with `Middleware.use`:

```
Tina4::Middleware.use(Tina4::CorsClassMiddleware)
Tina4::Middleware.use(Tina4::RateLimiterMiddleware)
Tina4::Middleware.use(Tina4::RequestLoggerMiddleware)
```

If a `before_*` method returns a response with status >= 400, the handler is skipped (short-circuit).

---

3. Built-in CorsMiddleware

Cross-Origin Resource Sharing (CORS) is the browser mechanism that controls which domains can call your API. Tina4 provides a built-in `CorsMiddleware` that handles this. Configure it in `.env`:

```
TINA4_CORS_ORIGINS=http://localhost:3000,https://myapp.com
TINA4_CORS_METHODS=GET,POST,PUT,PATCH,DELETE,OPTIONS
TINA4_CORS_HEADERS=Content-Type,Authorization,Accept
TINA4_CORS_MAX_AGE=86400
TINA4_CORS_CREDENTIALS=true
```

Apply it to a group:

```
Tina4::Router.group("/api", middleware: "CorsMiddleware") do

  Tina4::Router.get("/products") do |request, response|
    response.json({ products: [] })
  end

  Tina4::Router.post("/products") do |request, response|
    response.json({ created: true }, 201)
  end

end
```

---

4. Built-in RateLimiter

Rate limiting prevents a single client from overwhelming your API. Configure it in `.env`:

```
TINA4_RATE_LIMIT=60
TINA4_RATE_WINDOW=60
```

This means 60 requests per 60 seconds per IP. Apply it:

```
Tina4::Router.group("/api/public", middleware: "RateLimiter") do

  Tina4::Router.get("/search") do |request, response|
    q = request.params["q"] || ""
    response.json({ query: q, results: [] })
  end

end
```

When a client exceeds the limit:

```
{"error":"Rate limit exceeded. Try again in 42 seconds.","retry_after":42}
```

Custom Limits Per Group

```
# Public endpoints: 30 requests per minute
Tina4::Router.group("/api/public", middleware: "RateLimiter:30") do
  Tina4::Router.get("/search") do |request, response|
    response.json({ results: [] })
  end
end

# Authenticated endpoints: 120 requests per minute
Tina4::Router.group("/api/v1", middleware: ["auth_middleware", "RateLimiter:120"]) do
  Tina4::Router.get("/data") do |request, response|
    response.json({ data: [] })
  end
end
```

Built-in RequestLoggerMiddleware

The `RequestLoggerMiddleware` logs every request with its timing. It uses two hooks:

- `before_log` stamps the start time before the handler runs
- `after_log` calculates elapsed time and writes a log entry

Register it globally:

```
Tina4::Middleware.use(Tina4::RequestLoggerMiddleware)
```

The log output looks like:

```
[RequestLogger] GET /api/users -> 200 (12.345ms)
[RequestLogger] POST /api/products -> 201 (45.678ms)
```

Built-in SecurityHeadersMiddleware

The `SecurityHeadersMiddleware` adds standard security headers to every response. Register it globally:

```
Tina4::Middleware.use(Tina4::SecurityHeadersMiddleware)
```

It sets the following headers by default:

Header	Default Value
`X-Frame-Options`	`DENY`
`Content-Security-Policy`	`default-src 'self'`
`Strict-Transport-Security`	`max-age=31536000; includeSubDomains`
`Referrer-Policy`	`strict-origin-when-cross-origin`
`Permissions-Policy`	`camera=(), microphone=(), geolocation=()`

`X-Content-Type-Options`	`nosniff`

Override any header via environment variables in `.env`:

```
TINA4_FRAME_OPTIONS=SAMEORIGIN
TINA4_CSP=default-src 'self'; script-src 'self' https://cdn.example.com
TINA4_HSTS=max-age=63072000; includeSubDomains; preload
TINA4_REFERRER_POLICY=no-referrer
TINA4_PERMISSIONS_POLICY=camera=(), microphone=(), geolocation=(self)
```

Combining All Four Built-In Middleware

A common production setup registers all four globally:

```
Tina4::Middleware.use(Tina4::CorsClassMiddleware)
Tina4::Middleware.use(Tina4::RateLimiterMiddleware)
Tina4::Middleware.use(Tina4::RequestLoggerMiddleware)
Tina4::Middleware.use(Tina4::SecurityHeadersMiddleware)
```

Order matters. CORS handles preflight first. The rate limiter only counts real requests. The logger measures total time including the other middleware. Security headers are added to every response.

---

5. Writing Custom Middleware

Custom middleware follows the same pattern. You can write method-based or class-based middleware.

Request Logging Middleware

```
def log_request(request, response, next_handler)
  start = Process.clock_gettime(Process::CLOCK_MONOTONIC)
  method = request.method
  path = request.path
  ip = request.ip

  $stderr.puts "[#{Time.now.strftime('%Y-%m-%d %H:%M:%S')}] #{method} #{path} from #{ip}"

  result = next_handler.call(request, response)

  duration = ((Process.clock_gettime(Process::CLOCK_MONOTONIC) - start) * 1000).round(2)
  $stderr.puts "  Completed in #{duration}ms"

  result
end
```

Request Timing Middleware

```
def add_timing(request, response, next_handler)
  start = Process.clock_gettime(Process::CLOCK_MONOTONIC)

  result = next_handler.call(request, response)

  duration = ((Process.clock_gettime(Process::CLOCK_MONOTONIC) - start) * 1000).round(2)
  response.header("X-Response-Time", "#{duration}ms")

  result
end
```

IP Whitelist Middleware

```
def ip_whitelist(request, response, next_handler)
  allowed_ips = (ENV["ALLOWED_IPS"] || "127.0.0.1").split(",")

  unless allowed_ips.include?(request.ip)
    return response.json({
      error: "Access denied",
      your_ip: request.ip
    }, 403)
  end

  next_handler.call(request, response)
end
```

Request Validation Middleware

```
def require_json(request, response, next_handler)
  if %w[POST PUT PATCH].include?(request.method)
    content_type = request.headers["Content-Type"] || ""

    unless content_type.include?("application/json")
      return response.json({
        error: "Content-Type must be application/json",
        received: content_type
      }, 415)
    end
  end

  next_handler.call(request, response)
end
```

Writing Class-Based Middleware

For middleware that needs both before and after hooks, use the class-based pattern:

```
class InputSanitizer
  class << self
    def before_sanitize(request, response)
      if request.body.is_a?(Hash)
        request.body = sanitize_hash(request.body)
      end
      [request, response]
    end

    private

    def sanitize_hash(data)
```

```
      data.transform_values do |value|
        case value
        when String
          CGI.escapeHTML(value)
        when Hash
          sanitize_hash(value)
        else
          value
        end
      end
    end
  end
end
```

Register globally or apply to groups:

```
# Global
Tina4::Middleware.use(InputSanitizer)

# On a specific group
Tina4::Router.group("/api", middleware: "InputSanitizer") do
  # routes here
end
```

JWT Authentication Middleware (Class-Based)

```
class JwtAuthMiddleware
  class << self
    def before_verify_token(request, response)
      auth_header = request.headers["Authorization"] || ""

      unless auth_header.start_with?("Bearer ")
        response.json({ error: "Authorization header required" }, 401)
        return [request, response]
      end

      token = auth_header[7..]
      payload = Tina4::Auth.valid_token(token)

      if payload.nil?
        response.json({ error: "Invalid or expired token" }, 401)
        return [request, response]
      end

      request.user = payload
      [request, response]
    end
  end
end
```

Apply it to protected routes:

```
Tina4::Router.group("/api/protected", middleware: "JwtAuthMiddleware") do

  Tina4::Router.get("/profile") do |request, response|
    response.json({ user: request.user })
  end

  Tina4::Router.post("/settings") do |request, response|
    user_id = request.user["sub"]
```

```
    response.json({ updated: true, user_id: user_id })
  end

end
```

---

6. Applying Middleware to Individual Routes

Pass middleware as a keyword argument to any route method:

```
Tina4::Router.get("/api/data", middleware: "log_request") do |request, response|
  response.json({ data: [1, 2, 3] })
end

Tina4::Router.post("/api/data", middleware: ["log_request", "require_json"]) do |request, respon
  response.json({ created: true }, 201)
end
```

For a single middleware, pass a string. For multiple, pass an array. Each middleware runs in the order listed.

---

7. Route Groups with Shared Middleware

Groups apply middleware to every route inside them:

```
# Public API -- rate limited, CORS enabled
Tina4::Router.group("/api/public", middleware: ["CorsMiddleware", "RateLimiter:30"]) do

  Tina4::Router.get("/products") do |request, response|
    response.json({ products: [] })
  end

  Tina4::Router.get("/categories") do |request, response|
    response.json({ categories: [] })
  end

end

# Admin API -- auth required, IP restricted, logged
Tina4::Router.group("/api/admin", middleware: ["log_request", "ip_whitelist", "auth_middleware"]

  Tina4::Router.get("/users") do |request, response|
    response.json({ users: [] })
  end

  Tina4::Router.delete("/users/{id:int}") do |request, response|
    id = request.params["id"]
    response.json({ deleted: id })
  end

end
```

---

8. Middleware Execution Order

When you stack middleware, they execute from outer to inner -- like layers of an onion. The first middleware listed runs first on the way in and last on the way out.

Consider this setup:

```
def middleware_a(request, response, next_handler)
  $stderr.puts "A: before"
  result = next_handler.call(request, response)
  $stderr.puts "A: after"
  result
end

def middleware_b(request, response, next_handler)
  $stderr.puts "B: before"
  result = next_handler.call(request, response)
  $stderr.puts "B: after"
  result
end

def middleware_c(request, response, next_handler)
  $stderr.puts "C: before"
  result = next_handler.call(request, response)
  $stderr.puts "C: after"
  result
end

Tina4::Router.get("/api/test", middleware: ["middleware_a", "middleware_b", "middleware_c"]) do
  $stderr.puts "Handler"
  response.json({ ok: true })
end
```

Server log output:

```
A: before
B: before
C: before
Handler
C: after
B: after
A: after
```

The request flows inward: A, B, C, Handler. The response flows outward: C, B, A.

---

9. Short-Circuiting

When middleware does not call `next_handler`, the chain stops. No subsequent middleware runs and the route handler is never called.

Maintenance Mode

```
def maintenance_mode(request, response, next_handler)
  is_maintenance = (ENV["MAINTENANCE_MODE"] || "false") == "true"

  if is_maintenance
    if request.path == "/health"
      return next_handler.call(request, response)
    end

    return response.json({
      error: "Service is undergoing maintenance",
      retry_after: 300
    }, 503)
  end

  next_handler.call(request, response)
end
```

---

10. Modifying Requests in Middleware

Middleware can add data to the request before passing it to the handler:

```
def add_request_id(request, response, next_handler)
  request_id = SecureRandom.hex(8)
  request.request_id = request_id

  result = next_handler.call(request, response)

  response.header("X-Request-Id", request_id)

  result
end
```

---

11. Real-World Middleware Stack

Here is a realistic middleware setup for a production API:

```
# src/routes/middleware.rb

def add_request_id(request, response, next_handler)
  request.request_id = SecureRandom.hex(8)
  result = next_handler.call(request, response)
  response.header("X-Request-Id", request.request_id)
  result
end

def log_request(request, response, next_handler)
  start = Process.clock_gettime(Process::CLOCK_MONOTONIC)
  $stderr.puts "[#{request.request_id}] #{request.method} #{request.path}"

  result = next_handler.call(request, response)

  duration = ((Process.clock_gettime(Process::CLOCK_MONOTONIC) - start) * 1000).round(2)
```

```
  $stderr.puts "[#{request.request_id}] Completed in #{duration}ms"

  result
end

def require_api_key(request, response, next_handler)
  api_key = request.headers["X-API-Key"] || ""
  valid_keys = (ENV["API_KEYS"] || "").split(",")

  unless valid_keys.include?(api_key)
    return response.json({
      error: "Invalid or missing API key",
      request_id: request.request_id
    }, 401)
  end

  next_handler.call(request, response)
end
```

```
# src/routes/api.rb

Tina4::Router.group("/api/v1", middleware: ["add_request_id", "log_request", "CorsMiddleware", "

  Tina4::Router.get("/products") do |request, response|
    response.json({ products: [
      { id: 1, name: "Widget", price: 9.99 },
      { id: 2, name: "Gadget", price: 19.99 }
    ]})
  end

end
```

---

12. Exercise: Build an API Key Middleware

Build a middleware called `validate_api_key` that:

- Checks for an `X-API-Key` header on every request
- Validates the key against a comma-separated list stored in the `API_KEYS` environment variable
- If the key is missing, returns `401` with `{"error": "API key required"}`
- If the key is invalid, returns `403` with `{"error": "Invalid API key"}`
- If the key is valid, attaches the key to `request.api_key` and continues
- Apply this middleware to a route group with at least two endpoints

Setup

Add this to your `.env`:

```
API_KEYS=key-alpha-001,key-beta-002,key-gamma-003
```

Test with:

```
# No API key -- should get 401
curl http://localhost:7147/api/partner/data

# Invalid API key -- should get 403
curl http://localhost:7147/api/partner/data \
  -H "X-API-Key: wrong-key"

# Valid API key -- should get 200
curl http://localhost:7147/api/partner/data \
  -H "X-API-Key: key-alpha-001"
```

---

13. Solution

Create `src/routes/api_key_middleware.rb`:

```
def validate_api_key(request, response, next_handler)
  api_key = request.headers["X-API-Key"] || ""

  if api_key.empty?
    return response.json({ error: "API key required" }, 401)
  end

  valid_keys = (ENV["API_KEYS"] || "").split(",").map(&:strip)

  unless valid_keys.include?(api_key)
    return response.json({ error: "Invalid API key" }, 403)
  end

  request.api_key = api_key

  next_handler.call(request, response)
end

Tina4::Router.group("/api/partner", middleware: "validate_api_key") do

  Tina4::Router.get("/data") do |request, response|
    response.json({
      authenticated_with: request.api_key,
      data: [
        { id: 1, value: "alpha" },
        { id: 2, value: "beta" }
      ]
    })
  end

  Tina4::Router.get("/stats") do |request, response|
    response.json({
      authenticated_with: request.api_key,
      stats: {
        total_requests: 1423,
        avg_response_ms: 42
      }
    })
  end
```

```
end
```

Expected output -- no key:

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

(Status: `401 Unauthorized`)

Expected output -- valid key:

```
{
  "authenticated_with": "key-alpha-001",
  "data": [
    {"id": 1, "value": "alpha"},
    {"id": 2, "value": "beta"}
  ]
}
```

---

14. Gotchas

1. Middleware Must Be a Named Method

Problem: You pass an anonymous lambda as middleware and get an error.

Cause: Tina4 expects middleware to be referenced by a string name, not as an inline block. The string is resolved to a method at runtime.

Fix: Define your middleware as a named method: `def my_middleware(request, response, next_handler) ... end` and pass `"my_middleware"` as a string.

2. Forgetting to Return next_handler Result

Problem: Your middleware runs but the route handler never executes. The response is empty or a 500 error.

Cause: You called `next_handler.call(request, response)` but did not return the result.

Fix: Always return the result of `next_handler.call(request, response)`. Without the return, the middleware discards the response from the handler and returns `nil`.

3. Middleware Order Matters

Problem: Your logging middleware does not see the request ID, even though `add_request_id` is in the middleware list.

Cause: `log_request` runs before `add_request_id`. Middleware executes in the order listed.

Fix: Put `add_request_id` before `log_request` in the array: `["add_request_id", "log_request"]`.

4. CORS Preflight Returns 404

Problem: The browser's preflight `OPTIONS` request gets a 404, but `GET` and `POST` work fine when tested with curl.

Cause: You did not apply `CorsMiddleware` to the route, so the `OPTIONS` method is not handled.

Fix: Apply `CorsMiddleware` to the group. It automatically handles `OPTIONS` requests.

5. Rate Limiter Counts Preflight Requests

Problem: Your frontend hits the rate limit faster than expected because every `POST` request counts as two requests.

Cause: The rate limiter counts all requests, including `OPTIONS`.

Fix: Put `CorsMiddleware` before `RateLimiter` in the middleware chain.

6. Middleware File Not Auto-Loaded

Problem: You defined middleware in a file but get "method not found" when referencing it.

Cause: The file is not in `src/routes/`. Tina4 auto-loads all `.rb` files in `src/routes/`, but middleware defined outside that directory is not discovered.

Fix: Put your middleware methods in a file inside `src/routes/`, such as `src/routes/middleware.rb`.

7. Short-Circuiting Skips Cleanup Middleware

Problem: Your timing middleware logs the start time but never logs the completion time for blocked requests.

Cause: When an inner middleware short-circuits, the outer middleware's code after `next_handler.call` still runs. But if the short-circuiting middleware is listed before the timing middleware, the timing middleware never executes at all.

Fix: Put cleanup-dependent middleware (timing, logging) at the outermost layer.

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 method call to remember. The framework enforces this before your handler runs.

```
Tina4::Router.post "/api/orders" do |request, response|
  # This handler ONLY runs if the request carries a valid Bearer token.
  # Without one, the framework returns 401 before your code executes.
  response.call({ created: true }, 201)
end
```

Test it without a token:

```
curl -X POST http://localhost:7147/api/orders \
  -H "Content-Type: application/json" \
  -d '{"product": "widget"}'
# 401 Unauthorized
```

Test it with a valid token:

```
curl -X POST http://localhost:7147/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. Chain `.no_auth`:

```
Tina4::Router.post("/api/webhooks/stripe").no_auth do |request, response|
  # No token required. Stripe can POST here freely.
  response.call({ received: true })
end
```

Protecting a GET Route

Admin dashboards, user profiles, account settings — some pages need protection even though they only read data. Chain `.secure`:

```
Tina4::Router.get("/api/admin/users").secure do |request, response|
  # Requires a valid Bearer token, even though it's a GET.
  response.call({ users: [] })
end
```

The Rule

Method	Default	Override
GET, HEAD, OPTIONS	Public	`.secure` to protect
POST, PUT, PATCH, DELETE	Auth required	`.no_auth` to open

Two chainable methods. 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 `.no_auth`** — 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

```
Tina4::Router.post("/api/login").no_auth do |request, response|
  email = request.body["email"].to_s.strip
  password = request.body["password"].to_s.strip

  if email.empty? || password.empty?
    return response.call({ error: "Email and password required" }, 400)
  end

  # Look up user (replace with your database query)
  user = db.fetch_one(
    "SELECT id, email, password_hash, role FROM users WHERE email = ?",
    [email]
  )

  if user.nil?
    return response.call({ error: "Invalid credentials" }, 401)
  end

  # Verify password
  unless Tina4::Auth.check_password(password, user["password_hash"])
    return response.call({ error: "Invalid credentials" }, 401)
  end

  # Generate token with user claims
  token = Tina4::Auth.get_token(
    { sub: user["id"], email: user["email"], role: user["role"] }
  )

  # Store user in session
  request.session["user_id"] = user["id"]
  request.session["email"] = user["email"]
  request.session["role"] = user["role"]
  request.session.save

  response.call({ token: token, user: { id: user["id"], email: user["email"] } })
end
```

The `.no_auth` chain 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

```
Tina4::Router.get "/dashboard" do |request, response|
  user_id = request.session["user_id"]

  if user_id.nil?
    return response.redirect("/login")
  end

  response.render("dashboard.twig", {
    email: request.session["email"],
    role: request.session["role"]
  })
end
```

Logout — Destroying the Session

```
Tina4::Router.post("/api/logout").no_auth do |request, response|
  request.session.destroy
  response.call({ logged_out: true })
end
```

---

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.

```
require "cgi"

Tina4::Router.get "/account/settings" do |request, response|
  user_id = request.session["user_id"]

  if user_id.nil?
    # Remember where they wanted to go
    return_url = CGI.escape(request.url)
    return response.redirect("/login?redirect=#{return_url}")
  end

  response.render("settings.twig", { user_id: user_id })
end
```

The login handler reads the `redirect` parameter after successful authentication:

```
Tina4::Router.post("/api/login").no_auth do |request, response|
  # ... validate credentials ...

  redirect_url = request.params["redirect"] || "/dashboard"

  response.call({
    token: token,
    redirect: redirect_url
  })
end
```

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:

```
class LoginRateLimit
  def self.before_rate_check(request, response)
    # Custom rate limiter: 5 attempts per 60 seconds for login
    ip = request.ip
    # ... implement per-route rate limiting ...
    [request, response]
  end
end

Tina4::Router.post("/api/login").no_auth.middleware(LoginRateLimit) do |request, response|
  # ... login logic ...
end
```

---

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 `.no_auth` and validates credentials before issuing tokens.
- [ ] Session is regenerated after login (prevents session fixation).
- [ ] Passwords are hashed with `Tina4::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: Chain `.no_auth` onto the route definition 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 chain `.no_auth` onto 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 (`.no_auth`).
- 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.rb

Tina4::Router.get "/contact" do |request, response|
  response.render("contact.twig", { title: "Contact Us" })
end

Tina4::Router.post("/api/contact").no_auth do |request, response|
  name = request.body["name"].to_s.strip
  email = request.body["email"].to_s.strip
  message = request.body["message"].to_s.strip

  if name.empty? || email.empty? || message.empty?
    return response.call({ error: "All fields are required" }, 400)
  end

  db.insert("contact_messages", {
    name: name,
    email: email,
    message: message
  })

  response.call({ success: true, message: "Thank you for your message" })
end
```

```
{# 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 `.no_auth` chain 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 cache lives 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.

```
Tina4::Router.get("/api/products", middleware: ["ResponseCache:300"]) do |request, response|
  # This handler runs 12 database queries and takes 800ms
  # With ResponseCache, it only runs once every 5 minutes

  puts "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 }
  ]

  response.json({ products: products, generated_at: Time.now.utc.iso8601 })
end
```

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:7147/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": "2026-03-22T14:30:00Z"
```

```
}
```

Call it again within 5 minutes. The `generated_at` timestamp stays the same. The handler did not run -- the response came from cache.

Cache Headers

The `ResponseCache` middleware 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

The cache key includes the full URL with query parameters. `/api/products?page=1` and `/api/products?page=2` are cached separately:

```
curl "http://localhost:7147/api/products?page=1"  # Cache MISS, stores for page=1
curl "http://localhost:7147/api/products?page=2"  # Cache MISS, stores for page=2
curl "http://localhost:7147/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

---

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 works well 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=redis://localhost:6379
```

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 JSON 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 `cache_get`, `cache_set`, and `cache_delete` directly through the `Tina4` module:

cache_set

```
# Cache a value for 300 seconds
Tina4.cache_set("product:42", {
  id: 42,
  name: "Wireless Keyboard",
  price: 79.99,
  in_stock: true
}, ttl: 300)

# Cache a string
Tina4.cache_set("exchange_rate:USD_EUR", "0.92", ttl: 3600)
```

cache_get

```
product = Tina4.cache_get("product:42")
# Returns the cached value, or nil if not found or expired

if product.nil?
  # Cache miss -- fetch from database
  product = fetch_product_from_database(42)
  Tina4.cache_set("product:42", product, ttl: 300)
end
```

cache_delete

```
# Delete a specific key
Tina4.cache_delete("product:42")

# Delete multiple keys
Tina4.cache_delete("product:42")
Tina4.cache_delete("product:43")
Tina4.cache_delete("product:44")
```

Real-World Pattern: Cache-Aside

The most common caching pattern is cache-aside (also called lazy loading):

```
Tina4::Router.get("/api/products/{id:int}") do |request, response|
  product_id = request.params["id"]
  cache_key = "product:#{product_id}"

  # 1. Try the cache first
  product = Tina4.cache_get(cache_key)

  unless product.nil?
    # Cache hit -- return immediately
    return response.json(product.merge(source: "cache"))
  end

  # 2. Cache miss -- fetch from database
  db = Tina4.database
  product = db.fetch_one(
    "SELECT id, name, category, price, in_stock FROM products WHERE id = ?",
    [product_id]
  )

  if product.nil?
    return response.json({ error: "Product not found" }, 404)
  end

  # 3. Store in cache for next time
  Tina4.cache_set(cache_key, product, ttl: 600)  # Cache for 10 minutes

  response.json(product.merge(source: "database"))
end
```

```
curl http://localhost:7147/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:

```
Tina4::Router.get("/api/categories") do |request, response|
  db = Tina4.database

  # First call: executes the query (20ms)
  # Subsequent calls within 300 seconds: returns cached result (0.1ms)
  categories = db.fetch("SELECT * FROM categories ORDER BY name")

  response.json({ categories: categories })
end
```

The cache key derives from the SQL query and its parameters. Different queries or different parameters produce different cache keys:

```
# These are cached separately:
db.fetch("SELECT * FROM products WHERE category = ?", ["Electronics"])
db.fetch("SELECT * FROM products WHERE category = ?", ["Fitness"])
```

Any write operation (`insert`, `update`, `delete`, `execute`) automatically invalidates the query cache.

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

---

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:

```
Tina4.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:

```
Tina4::Router.put("/api/products/{id:int}") do |request, response|
  product_id = request.params["id"]
  body = request.body
  db = Tina4.database

  db.execute(
    "UPDATE products SET name = ?, price = ? WHERE id = ?",
    [body["name"], body["price"], product_id]
  )

  # Invalidate the cache for this product
  Tina4.cache_delete("product:#{product_id}")

  # Also invalidate list caches that might include this product
  Tina4.cache_delete("products:all")
  Tina4.cache_delete("products:featured")

  updated = db.fetch_one("SELECT * FROM products WHERE id = ?", [product_id])

  response.json(updated)
end
```

This is the most accurate strategy -- the cache is always fresh after a write. The downside: 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:

```
Tina4::Router.put("/api/products/{id:int}") do |request, response|
  product_id = request.params["id"]
  body = request.body
  db = Tina4.database

  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)
  Tina4.cache_set("product:#{product_id}", updated, ttl: 600)

  response.json(updated)
end
```

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:

```
def get_cached_product(product_id)
  cache_key = "product:#{product_id}"
  product = Tina4.cache_get(cache_key)

  return product unless product.nil?

  product = fetch_product_from_database(product_id)
```

```
  # Popular products: shorter TTL (more likely to change)
  # Inactive products: longer TTL (rarely change)
  ttl = product["view_count"].to_i > 1000 ? 60 : 3600

  Tina4.cache_set(cache_key, product, ttl: ttl)

  product
end
```

---

10. Cache Statistics

Monitor cache performance to verify that caching helps:

```
Tina4::Router.get("/api/cache/stats") do |request, response|
  response.json(Tina4.cache_stats)
end
```

```
curl http://localhost:7147/api/cache/stats
```

```
{
  "backend": "memory",
  "size": 42,
  "hits": 15234,
  "misses": 891
}
```

Hit rate above 90%: your caching strategy works. Below 80%: TTLs are too short, cache is too small, or you are caching data that does not benefit from caching.

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:

```
Tina4::Router.get("/api/catalog", middleware: ["ResponseCache:60"]) do |request, response|
  page = (request.params["page"] || 1).to_i
  cache_key = "catalog:page:#{page}"

  # Layer 1: Check application cache
  catalog = Tina4.cache_get(cache_key)

  unless catalog.nil?
    return response.json(catalog.merge(cache: "application"))
  end

  # Layer 2: Database query (with DB-level caching if TINA4_DB_CACHE=true)
  db = Tina4.database
  limit = 20
  offset = (page - 1) * limit

  products = db.fetch(
    "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: limit, offset: offset
   )

   total = db.fetch_one("SELECT COUNT(*) as count FROM products WHERE active = 1")

   catalog = {
     products: products,
     page: page,
     total: total["count"],
     pages: (total["count"].to_f / limit).ceil,
     generated_at: Time.now.utc.iso8601
   }

   # Store in application cache
   Tina4.cache_set(cache_key, catalog, ttl: 300)

   response.json(catalog.merge(cache: "none"))
 end
```

This creates three cache layers:

- **ResponseCache** (60 seconds): The entire HTTP response is cached. No Ruby 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 (`Tina4.cache_get`/`Tina4.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:7147/api/store/products?category=Electronics&page=1"

# Second call -- cache hit, fast
curl "http://localhost:7147/api/store/products?category=Electronics&page=1"

# Different category -- cache miss
curl "http://localhost:7147/api/store/products?category=Fitness&page=1"

# Create a product -- should invalidate cache
curl -X POST http://localhost:7147/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:7147/api/store/products?category=Electronics&page=1"

# Check cache stats
curl http://localhost:7147/api/store/cache-stats
```

---

13. Solution

Create `src/routes/store_cached.rb`:

```
require "digest"
require "json"

PRODUCT_STORE = [
  { id: 1, name: "Wireless Keyboard", category: "Electronics", price: 79.99, in_stock: true },
  { 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: "Electronics", price: 549.99, in_stock: true },
  { id: 5, name: "Running Shoes", category: "Fitness", price: 119.99, in_stock: true },
  { id: 6, name: "Bluetooth Speaker", category: "Electronics", price: 39.99, in_stock: true },
  { id: 7, name: "Resistance Bands", category: "Fitness", price: 14.99, in_stock: true },
  { id: 8, name: "French Press", category: "Kitchen", price: 34.99, in_stock: true }
]

Tina4::Router.get("/api/store/products") do |request, response|
  category = request.params["category"]
  page = (request.params["page"] || 1).to_i
  limit = (request.params["limit"] || 20).to_i

  # Build cache key from query parameters
  key_data = JSON.generate({ category: category, page: page, limit: limit })
  cache_key = "store:products:#{Digest::MD5.hexdigest(key_data)}"

  # Try cache first
  cached = Tina4.cache_get(cache_key)

  unless cached.nil?
    return response.json(cached.merge("source" => "cache"))
  end
```

```
  # Simulate expensive database query
  sleep(0.1)

  products = PRODUCT_STORE.dup

  # Filter by category
  if category
    products = products.select { |p| p[:category].downcase == category.downcase }
  end

  total = products.length
  offset = (page - 1) * limit
  products = products[offset, limit] || []

  result = {
    products: products,
    page: page,
    limit: limit,
    total: total,
    pages: (total.to_f / limit).ceil,
    generated_at: Time.now.utc.iso8601
  }

  # Cache for 5 minutes
  Tina4.cache_set(cache_key, result, ttl: 300)

  response.json(result.merge(source: "database"))
end

Tina4::Router.post("/api/store/products") do |request, response|
  body = request.body

  if body["name"].nil? || body["name"].empty?
    return response.json({ error: "Name is required" }, 400)
  end

  product = {
    id: rand(100..9999),
    name: body["name"],
    category: body["category"] || "General",
    price: (body["price"] || 0).to_f,
    in_stock: true
  }

  # Invalidate all product list caches
  Tina4.cache_clear

  response.json({
    message: "Product created",
    product: product,
    cache_invalidated: true
  }, 201)
end

Tina4::Router.get("/api/store/cache-stats") do |request, response|
  response.json(Tina4.cache_stats)
end
```

Expected output -- first call (cache miss):

```
{
  "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": "2026-03-22T14:30:00Z",
  "source": "database"
}
```


Expected output -- second call (cache hit):

Same data, but `source` changes 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: `Tina4.cache_set("profile:#{user_id}", data, ttl: 300)`.

2. Cache Stampede

Problem: When a popular cache key expires, hundreds of requests hit the database simultaneously.

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.

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.

Fix: For production, use Redis cache (`TINA4_CACHE_BACKEND=redis`). It persists across server restarts. Or implement a cache warmup script that pre-populates 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 `Tina4.cache_delete("product:#{product_id}")` after an update, or use write-through caching 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.

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 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` without a TTL. The entry lives until the server restarts.

Fix: Always set a TTL: `Tina4.cache_set("key", value, ttl: 300)`. Even for data that "never changes," set a long TTL like 86400 (24 hours). This prevents unbounded memory growth.

Queue System

1. Do Not Make the User Wait

Your app sends welcome emails on signup. Generates PDF invoices. Resizes uploaded images. Each task takes 2 to 30 seconds. Run them inside the HTTP request and the user stares at a spinner while the server grinds through email delivery, invoice rendering, and image resizing.

Queues move slow work to a background process. The HTTP handler drops a job onto a queue and responds to the user in under 100 milliseconds. A separate consumer picks up the job and does the work at its own pace. The user sees "Welcome! Check your email." The email arrives 5 seconds later.

Tina4's queue system works out of the box with a file-based backend. No Redis. No RabbitMQ. No external services. Add jobs and process them.

---

2. Why Queues Matter

Without queues:

```
User clicks "Sign Up"
  -> Server validates input (10ms)
  -> Server creates user in database (20ms)
  -> Server sends welcome email (3000ms)
  -> Server generates PDF welcome kit (2000ms)
  -> Server resizes avatar (1500ms)
  -> User sees response (6530ms later)
```

With queues:

```
User clicks "Sign Up"
  -> Server validates input (10ms)
  -> Server creates user in database (20ms)
  -> Server queues: send welcome email (1ms)
  -> Server queues: generate PDF (1ms)
  -> Server queues: resize avatar (1ms)
  -> User sees response (33ms later)

Meanwhile, in the background:
  -> Consumer sends welcome email
  -> Consumer generates PDF
  -> Consumer resizes avatar
```

6.5 seconds becomes 33 milliseconds. The user feels the difference.

---

3. File Queue (Default)

The file-based backend is the default. No configuration needed.

Creating a Queue and Pushing a Job

```
queue = Tina4::Queue.new(topic: "emails")

# Push a job
queue.push({
  to: "alice@example.com",
  subject: "Order Confirmation",
  body: "Your order #1234 has been confirmed."
})
```

Convenience Method: produce

The `produce` method pushes to a specific topic without creating a separate Queue instance:

```
queue = Tina4::Queue.new(topic: "emails")
queue.produce("invoices", { order_id: 101, format: "pdf" })
```

Push with Priority

Jobs default to priority 0 (normal). Higher numbers are popped first:

```
# Normal priority (default)
queue.push({ to: "alice@example.com", subject: "Newsletter" })

# High priority -- processed before normal jobs
queue.push({ to: "alice@example.com", subject: "Password Reset" }, priority: 10)
```

Queue Size

Check how many pending messages are in the queue:

```
count = queue.size
```

Pass a status keyword to count jobs in a specific state:

```
failed = queue.size(status: "failed")
```

---

4. Pushing from Route Handlers

```
Tina4::Router.post("/api/register") do |request, response|
  body = request.body

  user_id = 42 # Simulated

  queue = Tina4::Queue.new(topic: "emails")

  queue.push({
    user_id: user_id,
    to: body["email"],
    name: body["name"],
    subject: "Welcome!"
  })

  response.json({
    message: "Registration successful. Welcome email will arrive shortly.",
    user_id: user_id
  }, 201)
end
```

```
curl -X POST http://localhost:7147/api/register \
  -H "Content-Type: application/json" \
  -d '{"name": "Alice", "email": "alice@example.com", "password": "securePass123"}'
```

```
{
  "message": "Registration successful. Welcome email will arrive shortly.",
  "user_id": 42
}
```

---

5. Consuming Jobs

The `consume` method yields jobs one at a time via a block. Each job must be explicitly completed or failed:

```
queue = Tina4::Queue.new(topic: "emails")

queue.consume("emails") do |job|
  begin
    send_email(job.payload[:to], job.payload[:subject], job.payload[:body])
    job.complete
  rescue => e
    job.fail(e.message)
  end
end
```

Retry with Delay

If a job fails but you want to retry it after a cooldown instead of marking it as failed:

```
queue.consume("emails") do |job|
  begin
    send_email(job.payload[:to], job.payload[:subject], job.payload[:body])
    job.complete
  rescue => e
    # Retry after 30 seconds instead of failing immediately
```

```
    job.retry(queue: queue, delay_seconds: 30)
  end
end
```

Manual Pop

For more control, pop a single message:

```
job = queue.pop

unless job.nil?
  begin
    send_email(job.payload[:to], job.payload[:subject])
    job.complete
  rescue => e
    job.fail(e.message)
  end
end
```

---

6. Job Lifecycle

Every job moves through states:

```
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`
- `job.retry(queue: queue, delay_seconds: 30)` -- re-queue the job after a delay (in seconds). The job goes back to PENDING after the delay elapses. You must pass the `queue` reference.

Job Properties

- `job.topic` -- the topic this job belongs to. Useful when consuming from multiple topics.

Always call `complete`, `fail`, or `retry` on every job. If you do not, the job stays reserved.

---

7. Retry and Dead Letters

Max Retries

The default `max_retries` is 3. When a job's attempt count reaches `max_retries`, `retry_failed` skips it.

Retrying Failed Jobs

```
# 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

dead_jobs.each do |job|
  puts "Dead job: #{job.id}"
  puts "  Payload: #{job.payload}"
  puts "  Error: #{job.error}"
end
```

Purging Jobs

Remove jobs by status:

```
queue.purge("completed")
queue.purge("failed")
```

---

8. Producing Multiple Jobs

Sometimes a single action triggers multiple background tasks:

```
Tina4::Router.post("/api/orders") do |request, response|
  body = request.body
  order_id = 101

  queue = Tina4::Queue.new(topic: "emails")

  queue.push({
    order_id: order_id,
    to: body["email"],
    subject: "Order Confirmation"
  })

  queue.produce("invoices", {
    order_id: order_id,
    format: "pdf"
  })

  queue.produce("inventory", {
    items: body["items"]
  })
```

```
  queue.produce("warehouse", {
    order_id: order_id,
    shipping_address: body["shipping_address"]
  })

  response.json({
    message: "Order placed successfully",
    order_id: order_id
  }, 201)
end
```

Four jobs are queued in under 5 milliseconds. The user gets an instant response.

---

9. Switching Backends

Switching backends is a config change, not a code change.

Default: File

```
# No config needed -- file is the default
# Optionally set a custom storage path (defaults to ./queue)
TINA4_QUEUE_PATH=./data/queue
```

RabbitMQ

```
TINA4_QUEUE_BACKEND=rabbitmq
TINA4_QUEUE_URL=amqp://user:pass@localhost:5672
```

Kafka

```
TINA4_QUEUE_BACKEND=kafka
TINA4_QUEUE_URL=localhost:9092
```

MongoDB

```
TINA4_QUEUE_BACKEND=mongodb
TINA4_QUEUE_URL=mongodb://user:pass@localhost:27017/tina4
```

Install the MongoDB driver:

```
gem install mongo
```

Your code does not change. The same `queue.push` and `queue.consume` calls work with every backend.

---

10. Separate Producer and Consumer Patterns

Use separate `Tina4::Queue` instances in different files or services for clarity:

```
# Producer side (e.g., in a route handler)
queue = Tina4::Queue.new(topic: "emails")
queue.push({ to: "alice@example.com", subject: "Hello" })

# Consumer side (e.g., in a worker script)
queue = Tina4::Queue.new(topic: "emails")
```

```
queue.consume("emails") do |job|
  process(job)
  job.complete
end
```

The same `Tina4::Queue` class handles both producing and consuming. Separate instances in different files make intent clearer.

---

11. Exercise: Build an Email Queue

Build a queue-based email system with failure handling.

Requirements

- Create these endpoints:

Method	Path	Description
`POST`	`/api/emails/send`	Queue an email for sending
`GET`	`/api/emails/queue`	Show 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

Test with:

```
curl -X POST http://localhost:7147/api/emails/send \
  -H "Content-Type: application/json" \
  -d '{"to": "alice@example.com", "subject": "Welcome!", "body": "Thanks for signing up."}'

curl http://localhost:7147/api/emails/queue

curl http://localhost:7147/api/emails/dead

curl -X POST http://localhost:7147/api/emails/retry
```

---

12. Solution

Create `src/routes/email_queue.rb`:

```
queue = Tina4::Queue.new(topic: "emails")

# @noauth
Tina4::Router.post("/api/emails/send") do |request, response|
  body = request.body

  errors = []
  errors << "'to' is required" if body["to"].nil? || body["to"].empty?
```

```
  errors << "'subject' is required" if body["subject"].nil? || body["subject"].empty?
  errors << "'body' is required" if body["body"].nil? || body["body"].empty?

  unless errors.empty?
    return response.json({ errors: errors }, 400)
  end

  message_id = queue.push({
    to: body["to"],
    subject: body["subject"],
    body: body["body"]
  })

  response.json({
    message: "Email queued for sending",
    message_id: message_id
  }, 201)
end

Tina4::Router.get("/api/emails/queue") do |request, response|
  count = queue.size
  response.json({ pending: count })
end

Tina4::Router.get("/api/emails/dead") do |request, response|
  dead_jobs = queue.dead_letters
  items = dead_jobs.map do |job|
    { id: job.id, payload: job.payload, error: job.error }
  end
  response.json({ dead_letters: items, count: items.length })
end

Tina4::Router.post("/api/emails/retry") do |request, response|
  queue.retry_failed
  response.json({ message: "Failed emails re-queued for retry" })
end
```

Create a separate consumer file `src/workers/email_worker.rb`:

```
queue = Tina4::Queue.new(topic: "emails")

queue.consume("emails") do |job|
  payload = job.payload

  puts "Sending email to #{payload[:to]}..."
  puts "  Subject: #{payload[:subject]}"

  begin
    # Simulate sending (replace with real email logic)
    sleep(1)

    # Simulate failure for a specific address
    if payload[:to] == "bad@example.com"
      raise "SMTP connection refused"
    end

    puts "  Email sent to #{payload[:to]} successfully!"
    job.complete

  rescue => e
```

```
    puts "  Failed: #{e.message}"
    job.fail(e.message)
  end
end
```

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.

---

13. Gotchas

1. Always call complete or fail

Problem: Jobs stay in reserved status forever.

Cause: Your consumer block 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 block.

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 in `queue.consume`.

3. Payload must be serializable

Problem: `queue.push` throws an error.

Cause: You passed a non-serializable object (database connection, file handle, etc.).

Fix: Only pass simple data types in the payload. If you need to reference a database record, pass the ID. The consumer looks up records by ID.

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, fix, then call `queue.retry_failed` or `queue.purge("failed")`.

5. File backend for production

Problem: Multiple workers cause contention on the file backend.

Cause: The file backend is designed for single-worker setups.

Fix: For production with multiple workers, switch to RabbitMQ, Kafka, or MongoDB via the `TINA4_QUEUE_BACKEND` environment variable.

6. Consumer block returns nothing useful

Problem: Jobs are processed but the system does not track completion.

Cause: Your consumer block does not call `job.complete`. Without an explicit call, the job stays reserved.

Fix: Always call `job.complete` when the job succeeds and `job.fail(reason)` when it fails.

Events

1. Decouple With Events

Your user registration handler validates input, creates the account, sends a welcome email, creates an activity log entry, and notifies an analytics service. Seven responsibilities crammed into one method. Adding an eighth means editing that method again.

Events decouple producers from consumers. The registration handler fires `user.registered`. Any number of listeners react to it — email, analytics, logging — without the handler knowing they exist. Adding a new reaction is adding a new listener.

Tina4's event system is in-process, synchronous by default, and zero dependency. No broker, no serialization, no network.

---

2. Basic Usage

Registering a Listener

```
Tina4::Events.on("user.registered") do |payload|
  puts "Welcome #{payload[:name]}!"
end
```

Emitting an Event

```
Tina4::Events.emit("user.registered", {
  id: 42,
  name: "Alice",
  email: "alice@example.com"
})
```

Output:

```
Welcome Alice!
```

Listeners receive whatever hash you pass to `emit`. Use symbol keys consistently.

---

3. Multiple Listeners

Register as many listeners as you like for the same event. All fire in registration order by default.

```
Tina4::Events.on("order.placed") do |payload|
  puts "Sending confirmation email to #{payload[:email]}"
end

Tina4::Events.on("order.placed") do |payload|
  puts "Reserving inventory for order ##{payload[:order_id]}"
end

Tina4::Events.on("order.placed") do |payload|
  puts "Notifying warehouse for shipping"
```

```
end

Tina4::Events.emit("order.placed", {
  order_id: 101,
  email: "alice@example.com",
  items: [{ sku: "KB-100", qty: 1 }]
})
```

Output:

```
Sending confirmation email to alice@example.com
Reserving inventory for order #101
Notifying warehouse for shipping
```

---

4. Priority

Control listener order with `priority`. Higher numbers run first.

```
Tina4::Events.on("payment.received", priority: 10) do |payload|
  puts "FRAUD CHECK first (priority 10)"
end

Tina4::Events.on("payment.received", priority: 5) do |payload|
  puts "Fulfill order second (priority 5)"
end

Tina4::Events.on("payment.received", priority: 1) do |payload|
  puts "Send receipt last (priority 1)"
end

Tina4::Events.emit("payment.received", { order_id: 201, amount: 99.99 })
```

Output:

```
FRAUD CHECK first (priority 10)
Fulfill order second (priority 5)
Send receipt last (priority 1)
```

Default priority is `0`. Listeners with the same priority run in registration order.

---

5. Once: Fire a Listener One Time

`once` registers a listener that fires exactly once and then removes itself.

```
Tina4::Events.once("app.started") do |payload|
  puts "App started at #{payload[:time]} -- this fires only once"
end

Tina4::Events.emit("app.started", { time: Time.now.utc.iso8601 })
Tina4::Events.emit("app.started", { time: Time.now.utc.iso8601 })
```

Output:

```
App started at 2026-04-02T09:00:00Z -- this fires only once
```

The second emit produces no output. The listener is gone after the first call.

Use `once` for initialization tasks, welcome messages, or any logic that must run exactly one time.

---

6. off: Remove a Listener

To remove a specific listener, keep a reference to the block.

```
logger = Tina4::Events.on("request.received") do |payload|
  puts "Request: #{payload[:method]} #{payload[:path]}"
end

Tina4::Events.emit("request.received", { method: "GET", path: "/api/users" })

# Remove the listener
Tina4::Events.off("request.received", logger)

Tina4::Events.emit("request.received", { method: "POST", path: "/api/users" })
```

Output:

```
Request: GET /api/users
```

The second emit fires no listener -- it was removed.

---

7. clear: Remove All Listeners

Remove all listeners for an event, or all listeners for all events.

```
# Clear listeners for a specific event
Tina4::Events.clear("order.placed")

# Clear all listeners for all events
Tina4::Events.clear
```

Use `clear` in tests to avoid listener bleed between test cases.

```
RSpec.describe "Order processing" do
  after(:each) { Tina4::Events.clear }

  it "sends confirmation email" do
    received = []
    Tina4::Events.on("order.placed") { |p| received << p[:email] }
    Tina4::Events.emit("order.placed", { email: "alice@example.com" })
    expect(received).to eq(["alice@example.com"])
  end
end
```

---

8. Events in Route Handlers

Wire events into your HTTP layer to keep handlers thin.

```
# src/routes/orders.rb

# @noauth
Tina4::Router.post("/api/orders") do |request, response|
  body = request.body

  order = {
    id: rand(1000..9999),
    email: body["email"],
    items: body["items"],
    total: body["total"]
  }

  Tina4::Events.emit("order.placed", order)

  response.json({ message: "Order received", order_id: order[:id] }, 201)
end
```

```
# src/listeners/order_listeners.rb

Tina4::Events.on("order.placed") do |order|
  puts "Email: Confirmation sent to #{order[:email]}"
end

Tina4::Events.on("order.placed") do |order|
  puts "Inventory: #{order[:items].length} item type(s) reserved"
end

Tina4::Events.on("order.placed", priority: 10) do |order|
  puts "Audit: Order #{order[:id]} logged"
end
```

```
curl -X POST http://localhost:7147/api/orders \
  -H "Content-Type: application/json" \
  -d '{"email":"alice@example.com","items":[{"sku":"KB-100","qty":1}],"total":79.99}'
```

```
{ "message": "Order received", "order_id": 5823 }
```

Server output:

```
Audit: Order 5823 logged
Email: Confirmation sent to alice@example.com
Inventory: 1 item type(s) reserved
```

---

9. Listing Registered Events

Inspect what listeners are registered.

```
Tina4::Events.listeners("order.placed").each do |listener|
  puts "Priority #{listener[:priority]}"
end
```

---

10. Gotchas

1. Listeners run synchronously

Events are in-process and synchronous. If a listener takes 3 seconds, `emit` blocks for 3 seconds. For slow work, push to a queue inside the listener instead of doing the work directly.

```
Tina4::Events.on("order.placed") do |order|
  queue = Tina4::Queue.new(topic: "emails")
  queue.push({ to: order[:email], subject: "Order Confirmation" })
end
```

2. Exceptions in listeners

If a listener raises an exception, subsequent listeners for the same event do not run. Wrap listener logic in `rescue` when reliability matters.

```
Tina4::Events.on("payment.received") do |payload|
  begin
    process_payment(payload)
  rescue => e
    puts "Payment listener error: #{e.message}"
  end
end
```

3. clear in tests

Always call `Tina4::Events.clear` in `after(:each)` blocks. Listeners registered in one test persist into the next unless explicitly removed.

4. once is not thread-safe by design

`once` is intended for single-threaded initialization sequences. Do not use it in concurrent request handlers -- the listener may fire multiple times before the first call completes removal.

Localization

1. Your App Speaks One Language

Every string in your app is hard-coded English. A French user sees "Order Confirmation". A German user sees "Error: Invalid email". A Japanese user sees dates as "April 2, 2026" instead of "2026■4■2■".

Localization separates translatable strings from code. Add a locale file per language. Switch the active locale per request. Every `t("key")` call returns the right string for the current user.

---

2. Locale Files

Store locale files as JSON in `src/locales/`. Name them by locale code.

`src/locales/en.json`:

```
{
  "greeting": "Hello, {name}!",
  "order": {
    "confirmation": "Order Confirmation",
    "placed": "Your order #{number} has been placed.",
    "total": "Total: {currency}{amount}"
  },
  "errors": {
    "invalid_email": "Invalid email address.",
    "required": "{field} is required."
  },
  "nav": {
    "home": "Home",
    "account": "My Account",
    "logout": "Log Out"
  }
}
```

`src/locales/fr.json`:

```
{
  "greeting": "Bonjour, {name}!",
  "order": {
    "confirmation": "Confirmation de commande",
    "placed": "Votre commande #{number} a été passée.",
    "total": "Total: {currency}{amount}"
  },
  "errors": {
    "invalid_email": "Adresse e-mail invalide.",
    "required": "{field} est requis."
  },
  "nav": {
    "home": "Accueil",
    "account": "Mon compte",
    "logout": "Se déconnecter"
  }
}
```

---

3. Basic Translation

```
i18n = Tina4::I18n.new(locale: "en", path: "src/locales")

puts i18n.t("greeting", name: "Alice")
# => Hello, Alice!

puts i18n.t("order.confirmation")
# => Order Confirmation

puts i18n.t("nav.logout")
# => Log Out
```

Dot notation navigates nested keys. `"order.confirmation"` maps to `locales["order"]["confirmation"]`.

---

4. Interpolation

Use `{placeholder}` in locale values. Pass named keyword arguments to `t`.

```
puts i18n.t("order.placed", number: "1042")
# => Your order #1042 has been placed.

puts i18n.t("order.total", currency: "$", amount: "79.99")
# => Total: $79.99

puts i18n.t("errors.required", field: "Email")
# => Email is required.
```

Missing interpolation variables are left as-is in the output string.

---

5. Switching Locales

Create one `I18n` instance per locale, or call `locale=` to switch.

```
i18n = Tina4::I18n.new(locale: "en", path: "src/locales")

puts i18n.t("greeting", name: "Alice")
# => Hello, Alice!

i18n.locale = "fr"

puts i18n.t("greeting", name: "Alice")
# => Bonjour, Alice!
```

---

6. Fallback Locale

When a key is missing from the active locale, fall back to a default.

```
i18n = Tina4::I18n.new(
  locale: "de",
  path: "src/locales",
  fallback: "en"
)

# "de" locale has no "nav.logout" key
puts i18n.t("nav.logout")
# => Log Out  (falls back to English)
```

Fallback prevents missing key errors in partially translated locales.

---

7. Per-Request Locale in Routes

Read the user's preferred locale from a header, cookie, or query param and set it on a per-request `I18n` instance.

```
# @noauth
Tina4::Router.get("/api/greeting") do |request, response|
  locale = request.headers["Accept-Language"]&.split(",")&.first&.strip || "en"
  locale = locale[0..1]  # normalize "fr-FR" to "fr"

  i18n = Tina4::I18n.new(locale: locale, path: "src/locales", fallback: "en")

  response.json({
    message: i18n.t("greeting", name: "World"),
    locale: locale
  })
end
```

```
curl -H "Accept-Language: fr" http://localhost:7147/api/greeting
```

```
{ "message": "Bonjour, World!", "locale": "fr" }
```

```
curl -H "Accept-Language: de" http://localhost:7147/api/greeting
```

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

German falls back to English because `de.json` does not exist.

---

8. Available Locales

List all locale files present in the locales directory.

```
i18n = Tina4::I18n.new(locale: "en", path: "src/locales")
puts i18n.available_locales.inspect
# => ["en", "fr"]
```

Use this to build a language switcher UI.

```
# @noauth
Tina4::Router.get("/api/locales") do |request, response|
  i18n = Tina4::I18n.new(locale: "en", path: "src/locales")
  response.json({ locales: i18n.available_locales })
end
```

---

9. Locale in Templates

Pass the `I18n` instance to your template context.

```
Tina4::Router.get("/dashboard") do |request, response|
  locale = request.cookies["locale"] || "en"
  i18n = Tina4::I18n.new(locale: locale, path: "src/locales", fallback: "en")

  response.render("dashboard.html", { i18n: i18n, user: { name: "Alice" } })
end
```

In `src/templates/dashboard.html` (Frond template):

```
<nav>
  <a href="/">{i18n.t("nav.home")}</a>
  <a href="/account">{i18n.t("nav.account")}</a>
  <a href="/logout">{i18n.t("nav.logout")}</a>
</nav>

<h1>{i18n.t("greeting", name: user.name)}</h1>
```

---

10. Date and Number Formatting

Add locale-specific formatting helpers to your locale files or handle them with Ruby's standard library.

```
i18n = Tina4::I18n.new(locale: "en", path: "src/locales")

# Format a date string for the current locale
date = Time.now

formatted = case i18n.locale
            when "en" then date.strftime("%B %-d, %Y")
            when "fr" then date.strftime("%-d %B %Y")
            when "de" then date.strftime("%-d. %B %Y")
            else date.strftime("%Y-%m-%d")
            end

puts formatted
# en: April 2, 2026
# fr: 2 avril 2026
# de: 2. April 2026
```

---

11. Gotchas

1. Missing locale file returns fallback silently

If the requested locale file does not exist and a fallback is set, the fallback runs without warning. Log the missing locale in development.

2. Nested key typo

`i18n.t("order.confimation")` (typo) returns the key string `"order.confimation"` unchanged. If you see keys appearing raw in the UI, check for spelling mistakes in your `t()` calls.

3. Reload locale files in development

Locale file changes do not reload automatically in a running server. Restart the server or implement a dev-mode watch pattern.

4. Interpolation keys are case-sensitive

`{Name}` in the locale file is not the same as `name:` in the `t()` call. Keep all interpolation placeholders lowercase.

Structured Logging

1. puts Is Not a Logging Strategy

`puts` dumps raw text to stdout. No timestamps. No severity levels. No structured fields. Searching through 50,000 lines of `puts` output for a production error at 2:47 AM is the pain that structured logging prevents.

Structured logging records every entry as a consistent, parseable event: timestamp, level, message, and any additional fields you provide. Log aggregators (Datadog, Elastic, CloudWatch) can filter, group, and alert on structured logs automatically. Plain text output cannot.

Tina4's logger writes structured entries, supports configurable levels, and is available everywhere in your app.

---

2. Log Levels

Five levels, in order of increasing severity:

Level	Method	When to use
DEBUG	`Tina4::Log.debug`	Detailed trace — request internals, query params, timings
INFO	`Tina4::Log.info`	Normal operations — request received, user logged in
WARNING	`Tina4::Log.warning`	Something unexpected but recoverable — deprecated API called
ERROR	`Tina4::Log.error`	Something broke but the app is still running
FATAL	`Tina4::Log.fatal`	Unrecoverable error — app cannot continue

---

3. Basic Usage

```
Tina4::Log.debug("Cache miss", key: "products:list")
Tina4::Log.info("User logged in", user_id: 42, email: "alice@example.com")
Tina4::Log.warning("Deprecated endpoint called", path: "/api/v1/products")
Tina4::Log.error("Database query failed", table: "orders", error: "connection timeout")
Tina4::Log.fatal("Out of memory -- shutting down")
```

Output (text format):

```
[2026-04-02 09:00:01 UTC] DEBUG  Cache miss | key=products:list
[2026-04-02 09:00:02 UTC] INFO   User logged in | user_id=42 email=alice@example.com
[2026-04-02 09:00:03 UTC] WARN   Deprecated endpoint called | path=/api/v1/products
[2026-04-02 09:00:04 UTC] ERROR  Database query failed | table=orders error=connection timeout
[2026-04-02 09:00:05 UTC] FATAL  Out of memory -- shutting down
```

Each entry has a timestamp, level, message, and any keyword arguments you passed as structured fields.

---

4. Controlling the Log Level

Set the minimum log level via the `TINA4_LOG_LEVEL` environment variable. Entries below this level are silently dropped.

```
TINA4_LOG_LEVEL=info
```

Valid values (case-insensitive): `debug`, `info`, `warning`, `error`, `fatal`.

With `TINA4_LOG_LEVEL=info`, `Tina4::Log.debug(...)` calls produce no output. With `TINA4_LOG_LEVEL=error`, only `error` and `fatal` entries appear.

Default is `info` in production and `debug` in development (detected via `TINA4_ENV`).

---

5. Setting the Level in Code

```
Tina4::Log.level = :debug
Tina4::Log.level = :info
Tina4::Log.level = :warning
Tina4::Log.level = :error
Tina4::Log.level = :fatal
```

The environment variable takes precedence over the in-code setting at startup. Changing it at runtime affects all subsequent log calls.

---

6. Logging in Route Handlers

```
Tina4::Router.get("/api/products/:id") do |request, response|
  id = request.params["id"].to_i

  Tina4::Log.debug("Product lookup", product_id: id)

  product = Product.find(id)

  if product.nil?
    Tina4::Log.warning("Product not found", product_id: id)
    next response.json({ error: "Product not found" }, 404)
  end

  Tina4::Log.info("Product retrieved", product_id: id, name: product.name)
  response.json({ product: product.to_h })
end
```

---

7. Logging Errors with Exceptions

Pass the exception message as a structured field.

```
Tina4::Router.post("/api/orders") do |request, response|
  begin
    body = request.body
    order = create_order(body)

    Tina4::Log.info("Order created", order_id: order[:id], total: order[:total])
    response.json({ order_id: order[:id] }, 201)

  rescue ArgumentError => e
    Tina4::Log.warning("Invalid order payload", error: e.message)
    response.json({ error: e.message }, 400)

  rescue => e
    Tina4::Log.error("Order creation failed",
      error: e.message,
      backtrace: e.backtrace.first(3).join(" | ")
    )
    response.json({ error: "Internal server error" }, 500)
  end
end
```

---

8. Request Logging Middleware

Add the `RequestLogger` middleware to log every inbound request automatically.

```
Tina4::Router.get("/api/users", middleware: ["RequestLogger"]) do |request, response|
  response.json({ users: [] })
end
```

The middleware logs:

```
[2026-04-02 09:01:00 UTC] INFO  Request | method=GET path=/api/users ip=127.0.0.1
[2026-04-02 09:01:00 UTC] INFO  Response | method=GET path=/api/users status=200 duration_ms=4
```

Apply it globally via config to log every route:

```
# config/app.rb
Tina4::Config.middleware = ["RequestLogger"]
```

---

9. JSON Output Format

For log aggregators, switch to JSON output:

```
TINA4_LOG_FORMAT=json
```

The same calls now produce:

```
{"timestamp":"2026-04-02T09:00:02Z","level":"INFO","message":"User logged in","user_id":42,"emai
{"timestamp":"2026-04-02T09:00:04Z","level":"ERROR","message":"Database query failed","table":"o
```

One JSON object per line. Compatible with Datadog, Elastic, CloudWatch, and any log drain that understands NDJSON.

---

10. Writing to a File

```
TINA4_LOG_FILE=logs/app.log
```

The logger writes to the file and to stdout simultaneously. Log rotation is handled by the OS log rotation tool (logrotate on Linux, newsyslog on macOS).

---

11. Checking the Current Level

```
if Tina4::Log.debug?
  # Build expensive debug payload only if debug logging is active
  Tina4::Log.debug("Query plan", plan: db.explain(query).inspect)
end
```

Available predicates: `debug?`, `info?`, `warning?`, `error?`, `fatal?`.

---

12. Gotchas

1. Never log passwords or tokens

```
# Wrong
Tina4::Log.info("Login attempt", password: params["password"])

# Right
Tina4::Log.info("Login attempt", email: params["email"])
```

2. Avoid string interpolation in the message

```
# Slower -- builds the string even if the level is filtered
Tina4::Log.debug("User #{user.id} loaded #{records.count} records")

# Faster -- fields are only serialized if the level passes
Tina4::Log.debug("Records loaded", user_id: user.id, count: records.count)
```

3. DEBUG in production floods logs

Set `TINA4_LOG_LEVEL=info` in production. Debug output at scale produces millions of lines per hour and obscures real errors.

4. fatal does not exit

`Tina4::Log.fatal(...)` logs at the fatal level but does not call `exit`. Call `abort` or `Process.exit(1)` explicitly after logging an unrecoverable condition.

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. Bounce handling. The details compound. Tina4's `Messenger` class owns 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:

```
# Uses .env defaults
mailer = Tina4::Messenger.new

# Override specific settings
marketing_mailer = Tina4::Messenger.new(
  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:

```
Tina4::Router.post("/api/contact") do |request, response|
  body = request.body

  mail = Tina4::Messenger.new
  result = mail.send(
    to: body["email"],
    subject: "Contact Form Submission",
    body: "Name: #{body['name']}\nEmail: #{body['email']}\nMessage:\n#{body['message']}"
  )

  if result[:success]
    response.json({ message: "Email sent successfully" })
  else
    response.json({ error: "Failed to send email", details: result[:message] }, 500)
  end
end
```

```
curl -X POST http://localhost:7147/api/contact \
  -H "Content-Type: application/json" \
  -d '{"name": "Alice", "email": "alice@example.com", "message": "Hello!"}'
```

```
{"message":"Email sent successfully"}
```

The `send` method returns a hash with three keys: `success` (boolean), `message` (string), and `id` (message ID string or nil).

The send Method Signature

```
mail.send(
  to:,              # Recipient(s) -- string or array of strings
  subject:,         # Email subject line
  body:,            # Email body (plain text or HTML)
  html: false,      # If true, body is treated as HTML
  text: nil,        # Plain text alternative (when body is HTML)
  cc: [],           # CC recipient(s) -- string or array
  bcc: [],          # BCC recipient(s) -- string or array
  reply_to: nil,    # Reply-To address
  attachments: [],  # List of file paths or hashes
  headers: {}       # Additional email headers (hash)
)
```

---

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:

```
mail = Tina4::Messenger.new

html_body = <<~HTML
  <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>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>
HTML

text_body = <<~TEXT
  Hi Alice,

  Thank you for creating your account. We are excited to have you!

  Here is what you can do next:
  - Browse our product catalog: https://mystore.com/products
  - Set up your profile: https://mystore.com/profile
  - Check out our current deals: https://mystore.com/deals

  Cheers,
  The My Store Team
TEXT

result = mail.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:

```
mail = Tina4::Messenger.new

result = mail.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.

Multiple Attachments

```
mail = Tina4::Messenger.new
result = mail.send(
  to: "alice@example.com",
  subject: "Monthly Report",
  body: "Here are this month's reports.",
  attachments: [
    "/reports/sales.pdf",
    "/reports/analytics.xlsx",
    "/reports/summary.csv"
  ]
)
```

---

7. CC, BCC, and Reply-To

```
mail = Tina4::Messenger.new

result = mail.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 an array of strings.

---

8. Custom Headers

Messenger supports custom headers through the `headers` parameter on `send`:

```
result = mail.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"
  }
```

```
)
```

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 = Tina4::Messenger.new(imap_host: "imap.gmail.com", imap_port: 993)
```

Listing Inbox Messages

The `inbox` method fetches message headers from the mailbox:

```
Tina4::Router.get("/api/inbox") do |request, response|
  mailer = Tina4::Messenger.new

  emails = mailer.inbox(limit: 20, offset: 0)

  messages = emails.map do |email|
    {
      uid: email[:uid],
      from: email[:from],
      subject: email[:subject],
      date: email[:date],
      snippet: email[:snippet],
      seen: email[:seen]
    }
  end

  response.json({ messages: messages, count: messages.length })
end
```

```
curl http://localhost:7147/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

```
Tina4::Router.get("/api/inbox/{uid}") do |request, response|
  mailer = Tina4::Messenger.new
  uid = request.params["uid"]

  email = mailer.read(uid, mark_read: true)

  if email.nil?
    return response.json({ error: "Email not found" }, 404)
  end

  response.json({
    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: (email[:attachments] || []).map { |a|
      { filename: a[:filename], size: a[:size], content_type: a[:content_type] }
    }
  })
end
```

The `read` method fetches the full message including body and attachments. Pass `mark_read: false` to leave the message unread.

Searching Messages

```
Tina4::Router.get("/api/inbox/search") do |request, response|
  mailer = Tina4::Messenger.new

  results = mailer.search(
    subject: request.params["q"],
    sender: request.params["from"],
    unseen_only: request.params["unread"] == "true",
    limit: 20
  )

  response.json({ messages: results, count: results.length })
end
```

The `search` method accepts `subject`, `sender`, `since` (date string), `before`, and `unseen_only` as filters. All filters combine with AND logic.

Other IMAP Operations

```
mailer = Tina4::Messenger.new

# Count unread messages
count = mailer.unread

# List all mailbox folders
folders = mailer.folders
# ["INBOX", "Sent", "Drafts", "Trash", "Spam"]
```

---

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 click "Emails" to see:

- Recipient, subject, and timestamp
- Full HTML preview
- Plain text fallback
- Attachments list
- Headers

This means you can test email functionality without configuring SMTP during development.

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 Ruby 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. 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>
        </table>

        <p>Get started:</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>
    </div>
</body>
</html>
```

Rendering and Sending

Use `html_template` and `template_data` on the Messenger instance:

```
Tina4::Router.post("/api/register") do |request, response|
  body = request.body

  # Create user (database logic)
  user_id = 42

  mail = Tina4::Messenger.new
  mail.to = body["email"]
  mail.subject = "Welcome to My Store, #{body['name']}!"
  mail.html_template = "emails/welcome.html"
  mail.template_data = {
    name: body["name"],
    email: body["email"],
    user_id: user_id,
    base_url: ENV["APP_URL"] || "http://localhost:7147",
    app_name: "My Store",
    promo_code: "WELCOME10"
  }
  mail.send

  response.json({ message: "Registration successful", user_id: user_id }, 201)
```

```
end
```

```
curl -X POST http://localhost:7147/api/register \
  -H "Content-Type: application/json" \
  -d '{"name": "Alice", "email": "alice@example.com", "password": "securePass123"}'
```

```
{
  "message": "Registration successful",
  "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:

```
# In the route handler, queue the email
Tina4::Router.post("/api/register") do |request, response|
  body = request.body
  user_id = 42  # Simulated

  Tina4::Queue.produce("send-email", {
    to: body["email"],
    subject: "Welcome to My Store, #{body['name']}!",
    template: "emails/welcome.html",
    data: {
      name: body["name"],
      email: body["email"],
      user_id: user_id,
      base_url: "http://localhost:7147",
      app_name: "My Store",
      promo_code: "WELCOME10"
    }
  })

  response.json({ message: "Registration successful", user_id: user_id }, 201)
end

# The consumer sends the actual email
Tina4::Queue.consume("send-email") do |job|
  mail = Tina4::Messenger.new
  mail.to = job.payload["to"]
  mail.subject = job.payload["subject"]

  if job.payload["template"]
    mail.html_template = job.payload["template"]
    mail.template_data = job.payload["data"] || {}
  else
    mail.body = job.payload["body"] || ""
  end

  mail.send
```

```
  true
end
```

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 an Email Notification System

Build an email system that sends different types of notifications.

Requirements

- `POST /api/notify/welcome` -- Send a welcome email with an HTML template
- `POST /api/notify/order` -- Send an order confirmation with order details
- `POST /api/notify/reset` -- Send a password reset email with a token link

Each endpoint should queue the email rather than sending it directly.

Test with:

```
curl -X POST http://localhost:7147/api/notify/welcome \
  -H "Content-Type: application/json" \
  -d '{"name": "Alice", "email": "alice@example.com"}'

curl -X POST http://localhost:7147/api/notify/order \
  -H "Content-Type: application/json" \
  -d '{"email": "alice@example.com", "order_id": 101, "total": 159.98}'

curl -X POST http://localhost:7147/api/notify/reset \
  -H "Content-Type: application/json" \
  -d '{"email": "alice@example.com", "reset_token": "abc123def456"}'
```

---

14. Solution

Create `src/routes/notifications.rb`:

```
# @noauth
Tina4::Router.post("/api/notify/welcome") do |request, response|
  body = request.body

  Tina4::Queue.produce("send-email", {
    to: body["email"],
    subject: "Welcome to My Store, #{body['name']}!",
    template: "emails/welcome.html",
    data: { name: body["name"], login_url: "https://mystore.com/login", year: Time.now.year }
  })

  response.json({ message: "Welcome email queued" })
end

# @noauth
Tina4::Router.post("/api/notify/order") do |request, response|
  body = request.body
```

```
  Tina4::Queue.produce("send-email", {
    to: body["email"],
    subject: "Order Confirmation ##{body['order_id']}",
    template: "emails/order-confirmation.html",
    data: { order_id: body["order_id"], total: body["total"] }
  })

  response.json({ message: "Order confirmation email queued" })
end

# @noauth
Tina4::Router.post("/api/notify/reset") do |request, response|
  body = request.body

  Tina4::Queue.produce("send-email", {
    to: body["email"],
    subject: "Password Reset Request",
    template: "emails/password-reset.html",
    data: { reset_url: "https://mystore.com/reset?token=#{body['reset_token']}" }
  })

  response.json({ message: "Password reset email queued" })
end
```

---

15. Gotchas

1. Gmail Requires App Passwords

Problem: Gmail login fails with "authentication error".

Fix: Enable 2FA on your Google account, then generate an App Password at https://myaccount.google.com/apppasswords. Use the app password, not your regular password.

2. Emails Go to Spam

Problem: Emails arrive in the spam folder.

Fix: Set up SPF, DKIM, and DMARC DNS records for your domain. Use a reputable email service like Mailgun, SendGrid, or Postmark.

3. HTML Email Rendering Differences

Problem: Your email looks different in Gmail, Outlook, and Apple Mail.

Fix: Use inline CSS. Avoid flexbox and grid. Use tables for layout. Test with a tool like Litmus or Email on Acid.

4. Attachment File Not Found

Problem: `attachments: ["/path/to/file.pdf"]` raises a file not found error.

Fix: Use absolute paths. Relative paths resolve from the working directory, which may differ in production.

5. SMTP Connection Timeout

Problem: Sending email hangs for 30 seconds and then times out.

Fix: Check your SMTP host, port, and encryption settings. Common mistake: using port 587 with SSL instead of TLS.

6. Dev Mode Emails Disappear on Restart

Problem: Intercepted emails in the dev dashboard vanish when you restart the server.

Fix: This is expected. Dev mode stores emails in memory. For persistent storage, configure a real SMTP server.

7. Unicode Characters Display as Question Marks

Problem: Non-ASCII characters (accents, CJK) show as `?` in the email.

Fix: Tina4 sets UTF-8 encoding by default. If you are constructing raw headers, make sure you include `Content-Type: text/html; charset=utf-8`.

8. IMAP Connection Fails

Problem: `inbox` or `read` raises a connection error.

Fix: Verify `TINA4_MAIL_IMAP_HOST` and `TINA4_MAIL_IMAP_PORT` in `.env`. Gmail uses `imap.gmail.com` on port `993`. Make sure your email provider allows IMAP access -- some providers disable it by default.

Frontend with tina4css

1. The Problem with Frontend Toolchains

Your client wants a dashboard. Install Node.js. Run `npm install`. Wait for 200MB of `node_modules`. Configure webpack or Vite. Set up PostCSS. Add a CSS framework. Maybe Tailwind with its purge config. Pray nothing breaks when you upgrade a dependency in 6 months.

Tina4 skips all of that. The framework ships with **tina4css** -- a Bootstrap-compatible CSS framework -- and **frond.js** -- a lightweight JavaScript helper library. Both land in your project when you scaffold. No npm. No webpack. No build step. Link the files. Start building.

By the end of this chapter, you will have a complete admin dashboard with a sidebar, navigation, cards, tables, modals, and dark mode support. Zero npm involvement.

---

2. What Ships with Tina4

When you run `tina4 init`, two files appear in your project:

```
src/public/
■■■ css/
■   ■■■ tina4.css        # The CSS framework
■■■ js/
■   ■■■ frond.js         # AJAX helpers, form submission, token management
■■■ scss/
    ■■■ tina4.scss       # SCSS source (optional, for customization)
```

Include them in any template:

```
<link rel="stylesheet" href="/css/tina4.css">
<script src="/js/frond.js"></script>
```

---

3. The Grid System

tina4css uses a 12-column responsive grid, compatible with Bootstrap's class names:

```
<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>
```

Responsive Breakpoints

Prefix	Min Width	Typical Device
`col-`	0px	All
`col-sm-`	576px	Phones (landscape)
`col-md-`	768px	Tablets
`col-lg-`	992px	Laptops
`col-xl-`	1200px	Desktops

---

4. Cards

```
<div class="card">
    <div class="card-header">Product Details</div>
    <div class="card-body">
        <h3 class="card-title">Wireless Keyboard</h3>
        <p class="card-text">Ergonomic keyboard with backlit keys.</p>
        <p class="price">$79.99</p>
    </div>
    <div class="card-footer">
        <button class="btn btn-primary">Add to Cart</button>
    </div>
</div>
```

Card Grid

```
<div class="row">
    {% for product in products %}
        <div class="col-md-4">
            <div class="card">
                <div class="card-body">
                    <h3 class="card-title">{{ product.name }}</h3>
                    <p class="card-text">${{ product.price | number_format(2) }}</p>
                    <button class="btn btn-primary btn-sm">View</button>
                </div>
            </div>
        </div>
    {% endfor %}
</div>
```

---

5. Tables

```
<table class="table table-striped table-hover">
    <thead>
        <tr>
            <th>ID</th>
            <th>Name</th>
            <th>Price</th>
            <th>Actions</th>
        </tr>
    </thead>
    <tbody>
        {% for product in products %}
            <tr>
                <td>{{ product.id }}</td>
                <td>{{ product.name }}</td>
                <td>${{ product.price | number_format(2) }}</td>
                <td>
                    <button class="btn btn-sm btn-primary">Edit</button>
                    <button class="btn btn-sm btn-danger">Delete</button>
                </td>
            </tr>
        {% endfor %}
    </tbody>
</table>
```

---

6. Forms

```
<form method="POST" action="/api/products">
    <div class="form-group">
        <label for="name">Product Name</label>
        <input type="text" class="form-control" id="name" name="name" required>
    </div>
    <div class="form-group">
        <label for="price">Price</label>
        <input type="number" class="form-control" id="price" name="price" step="0.01" required>
    </div>
    <div class="form-group">
        <label for="category">Category</label>
        <select class="form-control" id="category" name="category">
            <option value="Electronics">Electronics</option>
            <option value="Fitness">Fitness</option>
            <option value="Kitchen">Kitchen</option>
            <option value="Office">Office</option>
        </select>
    </div>
    <button type="submit" class="btn btn-primary">Create Product</button>
</form>
```

---

7. Buttons and Alerts

Buttons

```
<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-danger">Danger</button>
<button class="btn btn-warning">Warning</button>
<button class="btn btn-info">Info</button>
<button class="btn btn-outline-primary">Outline</button>
<button class="btn btn-sm btn-primary">Small</button>
<button class="btn btn-lg btn-primary">Large</button>
```

Alerts

```
<div class="alert alert-success">Product created successfully!</div>
<div class="alert alert-danger">Error: Name is required.</div>
<div class="alert alert-warning">Stock is running low.</div>
<div class="alert alert-info">New features are available.</div>
```

---

8. frond.js -- AJAX Without jQuery

`frond.js` provides lightweight AJAX helpers for form submission, API calls, and token management.

API Calls

```
// GET request
frond.get("/api/products", function (data) {
    console.log(data.products);
});

// POST request
frond.post("/api/products", {
    name: "Widget",
    price: 9.99
}, function (data) {
    console.log("Created:", data);
});

// PUT request
frond.put("/api/products/1", {
    name: "Updated Widget",
    price: 12.99
}, function (data) {
    console.log("Updated:", data);
});

// DELETE request
frond.del("/api/products/1", function (data) {
    console.log("Deleted");
});
```

Token Management

```
// Store token after login
frond.post("/api/login", { email: "alice@example.com", password: "pass123" }, function (data) {
    frond.setToken(data.token);
});

// All subsequent requests automatically include the Authorization header
frond.get("/api/profile", function (data) {
    console.log(data);  // Token is sent automatically
});
```

Form Submission via AJAX

```
<form id="product-form" data-frond-submit="/api/products" data-frond-method="POST">
    <div class="form-group">
        <label for="name">Name</label>
        <input type="text" class="form-control" name="name" id="name">
    </div>
    <button type="submit" class="btn btn-primary">Create</button>
</form>

<div id="result"></div>

<script>
    frond.onSubmit("#product-form", function (data) {
        document.getElementById("result").innerHTML = '<div class="alert alert-success">Product
    });
</script>
```

---

9. Dark Mode

tina4css supports dark mode via a CSS class on the `<html>` or `<body>` element:

```
<html data-theme="dark">
```

Toggle with JavaScript:

```
function toggleDarkMode() {
    const html = document.documentElement;
    const current = html.getAttribute("data-theme");
    html.setAttribute("data-theme", current === "dark" ? "light" : "dark");
    localStorage.setItem("theme", html.getAttribute("data-theme"));
}

// Load saved theme
const savedTheme = localStorage.getItem("theme") || "light";
document.documentElement.setAttribute("data-theme", savedTheme);
```

---

10. Building a Dashboard

Here is a complete admin dashboard template:

Create `src/templates/dashboard.html`:

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

{% block title %}Dashboard{% endblock %}

{% block content %}
    <h1>Dashboard</h1>

    <div class="row">
        <div class="col-md-3">
            <div class="card">
                <div class="card-body text-center">
                    <h2>{{ stats.total_products }}</h2>
                    <p class="text-muted">Products</p>
                </div>
            </div>
        </div>
        <div class="col-md-3">
            <div class="card">
                <div class="card-body text-center">
                    <h2>{{ stats.total_orders }}</h2>
                    <p class="text-muted">Orders</p>
                </div>
            </div>
        </div>
        <div class="col-md-3">
            <div class="card">
                <div class="card-body text-center">
                    <h2>${{ stats.revenue | number_format(2) }}</h2>
                    <p class="text-muted">Revenue</p>
                </div>
            </div>
        </div>
        <div class="col-md-3">
            <div class="card">
                <div class="card-body text-center">
                    <h2>{{ stats.total_users }}</h2>
                    <p class="text-muted">Users</p>
                </div>
            </div>
        </div>
    </div>

    <div class="row mt-4">
        <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>Total</th><th>Status</th></tr
                        </thead>
                        <tbody>
                            {% for order in recent_orders %}
                                <tr>
                                    <td>#{{ order.id }}</td>
                                    <td>{{ order.customer }}</td>
                                    <td>${{ order.total | number_format(2) }}</td>
                                    <td><span class="badge badge-{{ order.status_class }}">{{ or
                                </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-secondary btn-block mb-2">View Orders
                <a href="/admin/users" class="btn btn-secondary btn-block">Manage Users</a>
            </div>
        </div>
    </div>
</div>
{% endblock %}
```

Route handler:

```
Tina4::Router.get("/admin") do |request, response|
  response.render("dashboard.html", {
    stats: {
      total_products: 156,
      total_orders: 1423,
      revenue: 45678.90,
      total_users: 342
    },
    recent_orders: [
      { id: 1001, customer: "Alice Smith", total: 159.98, status: "Shipped", status_class: "succ
      { id: 1000, customer: "Bob Jones", total: 49.99, status: "Processing", status_class: "warn
      { id: 999, customer: "Charlie Brown", total: 299.99, status: "Delivered", status_class: "i
    ]
  })
end
```

---

11. 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.

---

12. frond.js 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.

tina4.min.js -- Core Utilities

Low-level helpers for AJAX page loading and form submission.

`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.

```
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));
});
```

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 management, and dynamic template loading.

AJAX Requests

```
// GET request
frond.get("/api/products", function (data) {
    console.log("Products:", data);
});

// 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);
});
```

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.

```
frond.get("/api/products", function (data) {
    renderProducts(data.products);
}, null, {
    loading: "#loadingSpinner"
});
```

```
<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 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.
```

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.

Reactive State

```
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"
```

Web Components

```
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);
```

---

13. Responsive Design

tina4css includes responsive breakpoints for adapting layouts to different screen sizes:

Breakpoint	Min Width	Class Prefix
Extra small	0px	(none -- default)
Small	576px	`sm-`
Medium	768px	`md-`
Large	992px	`lg-`
Extra large	1200px	`xl-`

Responsive Columns

```
<div class="row">
    <!-- Full width on mobile, half on medium, third on large -->
    <div class="col-12 col-md-6 col-lg-4">Product 1</div>
    <div class="col-12 col-md-6 col-lg-4">Product 2</div>
    <div class="col-12 col-md-6 col-lg-4">Product 3</div>
</div>
```

Responsive Visibility

```
<!-- Hidden on mobile, visible on medium and up -->
<div class="d-none d-md-block">Desktop Sidebar</div>

<!-- Visible on mobile only -->
<div class="d-block d-md-none">Mobile Menu</div>
```

---

14. Building a Users Page with AJAX

A single-page admin view that loads data without page reloads:

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

{% block title %}User Management{% endblock %}

{% block content %}
<h2>User Management</h2>

<div id="loadingSpinner" class="text-center p-4" style="display: none;">
    Loading users...
</div>

<div id="user-list"></div>

<script src="/js/frond.min.js"></script>
<script>
    function loadUsers() {
        frond.get("/api/users", function (data) {
            var html = '<table class="table"><thead><tr>' +
                '<th>ID</th><th>Name</th><th>Email</th><th>Actions</th>' +
                '</tr></thead><tbody>';

            data.users.forEach(function (user) {
                html += '<tr>' +
                    '<td>' + user.id + '</td>' +
                    '<td>' + user.name + '</td>' +
                    '<td>' + user.email + '</td>' +
                    '<td><button class="btn btn-sm btn-danger" ' +
                    'onclick="deleteUser(' + user.id + ')">Delete</button></td>' +
                    '</tr>';
            });

            html += '</tbody></table>';
            document.getElementById("user-list").innerHTML = html;
        }, null, { loading: "#loadingSpinner" });
    }

    function deleteUser(id) {
        frond.delete("/api/users/" + id, function () {
            loadUsers();
        });
    }

    loadUsers();
</script>
{% endblock %}
```

The page loads instantly. frond.js fetches user data in the background. Delete a user and the list refreshes without a page reload.

---

15. Exercise: Build a Product Management Page

Build a product management page with a table, add/edit form, and AJAX interactions.

Requirements

- `GET /admin/products` -- HTML page with a product table and "Add Product" form
- Use frond.js for AJAX form submission
- Display flash messages for success/error
- Include search and category filter

---

16. Solution

Create `src/templates/admin-products.html` with a table displaying products, a form for adding new ones using `data-frond-submit`, and JavaScript handlers using `frond.post` and `frond.get` for CRUD operations. The template extends `base.html` and uses tina4css for styling.

Create `src/routes/admin_products.rb`:

```
Tina4::Router.get("/admin/products") do |request, response|
  db = Tina4.database
  products = db.fetch("SELECT * FROM products ORDER BY name")

  response.render("admin-products.html", {
    products: products,
    count: products.length
  })
end
```

---

17. Gotchas

1. tina4css Classes Do Not Work

Problem: CSS classes have no effect.

Fix: Make sure `<link rel="stylesheet" href="/css/tina4.css">` is in your HTML head.

2. frond.js Not Loading

Problem: `frond is not defined` error in the browser console.

Fix: Make sure `<script src="/js/frond.js"></script>` is included before your custom scripts.

3. AJAX Calls Return HTML Instead of JSON

Problem: `frond.get` returns HTML instead of JSON.

Fix: Your route handler is returning `response.render` instead of `response.json`. API endpoints should use `response.json`.

4. Forms Submit Twice

Problem: The form submits via AJAX and then also submits normally.

Fix: Call `event.preventDefault()` in your submit handler, or use `data-frond-submit` which handles this automatically.

5. Dark Mode Resets on Page Load

Problem: Dark mode resets to light mode when navigating.

Fix: Save the theme preference in `localStorage` and apply it on page load before the page renders.

6. Bootstrap Classes Not Working

Problem: Some Bootstrap classes do not work with tina4css.

Fix: tina4css is Bootstrap-compatible but not a complete clone. Stick to the documented classes. Complex Bootstrap components (modals, tooltips) require additional JavaScript.

7. SCSS Changes Not Reflected

Problem: You edited `tina4.scss` but the browser shows the old CSS.

Fix: Compile SCSS to CSS with `tina4 scss` or `sass src/public/scss/tina4.scss src/public/css/tina4.css`. The browser loads the compiled CSS file, not the SCSS source.

---

18. HtmlElement — Programmatic HTML Builder

Build HTML in Ruby without string concatenation:

```
el = Tina4::HtmlElement.new("div", { class: "card" }, ["Hello"])
el.to_s  # => '<div class="card">Hello</div>'

# Nesting
card = Tina4::HtmlElement.new("div", { class: "card" }, [
  Tina4::HtmlElement.new("h2", {}, ["Title"]),
  Tina4::HtmlElement.new("p", {}, ["Content"]),
])

# HtmlHelpers mixin
include Tina4::HtmlHelpers
html = _div({ class: "card" }, _p("Hello"), _a({ href: "/" }, "Home"))
```

Void tags (`<br>`, `<img>`, `<input>`) render without closing tags. Boolean attributes render as bare names.

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? You have 47 routes, 12 ORM models, and 3 middleware functions. Click through every page? An hour. Run 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 passes. You deploy with confidence. Weekend intact.

Tina4 uses RSpec for testing. Tests live in `tests/`. Every `_spec.rb` file is auto-discovered when you run `tina4 test`.

---

2. Your First Test

Create `tests/basic_spec.rb`:

```
require "tina4"

RSpec.describe "Basic tests" do
  it "adds two numbers" do
    expect(2 + 2).to eq(4)
  end

  it "concatenates strings" do
    result = "Hello" + " " + "World"
    expect(result).to eq("Hello World")
  end

  it "handles nil correctly" do
    value = nil
    expect(value).to be_nil
  end
end
```

Run it:

```
tina4 test

Running tests...

  Basic tests
    [PASS] adds two numbers
    [PASS] concatenates strings
    [PASS] handles nil correctly

  3 tests, 3 passed, 0 failed (0.01s)
```

You can also use `bundle exec rspec` directly:

```
bundle exec rspec tests/
```

---

3. RSpec Matchers Reference

RSpec provides matchers for every comparison you need. Here are the ones you will use most:

Equality

```
expect(result).to eq(42)            # value equality
expect(result).not_to eq(0)         # value inequality
expect(result).to eql(42)           # type + value equality
expect(object).to equal(other)      # identity (same object)
```

Truthiness

```
expect(true).to be_truthy           # truthy (not nil, not false)
expect(nil).to be_falsy             # falsy (nil or false)
expect(nil).to be_nil               # exactly nil
expect("hello").not_to be_nil       # not nil
expect(true).to be true             # exactly true
expect(false).to be false           # exactly false
```

Comparison

```
expect(10).to be > 5                # greater than
expect(3).to be < 10                # less than
expect(5).to be >= 5                # greater or equal
expect(5).to be <= 5                # less or equal
expect(5).to be_between(1, 10)      # range check
```

Strings

```
expect("Hello World").to include("World")         # substring
expect("Hello World").to start_with("Hello")      # prefix
expect("Hello World").to end_with("World")        # suffix
expect("user@example.com").to match(/@/)          # regex match
```

Collections

```
expect([1, 2, 3]).to include(2)                   # contains element
expect([1, 2, 3]).to have_attributes(length: 3)   # size check
expect([]).to be_empty                            # empty check
expect([3, 1, 2]).to contain_exactly(1, 2, 3)     # exact elements, any order
```

Exceptions

```
expect { Integer("not-a-number") }.to raise_error(ArgumentError)
expect { 10 / 0 }.to raise_error(ZeroDivisionError)
expect { safe_operation }.not_to raise_error
```

Floating Point

```
expect(9.99).to be_within(0.01).of(10.0)          # approximate equality
```

Type Checking

```
expect("hello").to be_a(String)
expect(42).to be_an(Integer)
expect([]).to be_an(Array)
```

---

4. Testing Routes

The `Tina4::TestClient` lets you make HTTP requests to your routes without starting a server:

```
require "tina4"

RSpec.describe "Product API" do
  let(:client) { Tina4::TestClient.new }

  it "returns products list" do
    result = client.get("/api/products")

    expect(result.status).to eq(200)

    data = result.json
    expect(data).to have_key("products")
    expect(data["products"]).to be_an(Array)
  end

  it "creates a product" do
    result = client.post("/api/products", {
      name: "Test Widget",
      category: "Test",
      price: 9.99
    })

    expect(result.status).to eq(201)

    data = result.json
    expect(data["name"]).to eq("Test Widget")
    expect(data["price"]).to eq(9.99)
    expect(data["id"]).not_to be_nil
  end

  it "returns 404 for missing product" do
    result = client.get("/api/products/99999")

    expect(result.status).to eq(404)

    data = result.json
    expect(data["error"]).to eq("Product not found")
```

```
  end

  it "validates required fields" do
    result = client.post("/api/products", {})

    expect(result.status).to eq(400)

    data = result.json
    expect(data["error"]).to include("required")
  end
end
```

TestClient Methods

```
client = Tina4::TestClient.new

# GET request
result = client.get("/api/products")
result = client.get("/api/products?category=Electronics")

# POST request with JSON body
result = client.post("/api/products", { name: "Widget", price: 9.99 })

# PUT request
result = client.put("/api/products/1", { name: "Updated Widget" })

# DELETE request
result = client.delete("/api/products/1")

# With custom headers
result = client.get("/api/profile", headers: { "Authorization" => "Bearer #{token}" })
```

TestResult Properties

```
result.status      # HTTP status code (200, 201, 404, etc.)
result.json        # Parsed JSON body as a hash
result.body        # Raw response body as a string
result.headers     # Response headers hash
```

---

5. Testing ORM Models

```
require "tina4"

RSpec.describe Product do
  before(:each) do
    db = Tina4.database
    db.execute("DELETE FROM products")
  end

  it "creates and saves a product" do
    product = Product.new
    product.name = "Test Product"
    product.price = 29.99
    product.category = "Testing"
    product.save

    expect(product.id).not_to be_nil
    expect(product.name).to eq("Test Product")
  end

  it "loads a product by ID" do
    product = Product.new
    product.name = "Load Test"
    product.price = 19.99
    product.save

    loaded = Product.new
    loaded.load(product.id)

    expect(loaded.name).to eq("Load Test")
    expect(loaded.price).to eq(19.99)
  end

  it "updates a product" do
    product = Product.new
    product.name = "Before Update"
    product.price = 10.00
    product.save

    product.name = "After Update"
    product.price = 20.00
    product.save

    loaded = Product.new
    loaded.load(product.id)

    expect(loaded.name).to eq("After Update")
    expect(loaded.price).to eq(20.00)
  end

  it "deletes a product" do
    product = Product.new
    product.name = "To Delete"
    product.price = 5.00
    product.save

    id = product.id
    product.delete
```

```
      loaded = Product.new
      loaded.load(id)

      expect(loaded.id).to be_nil
    end

    it "selects products with filter" do
      product = Product.new

      p1 = Product.new
      p1.name = "Electronics Item"
      p1.category = "Electronics"
      p1.price = 99.99
      p1.save

      p2 = Product.new
      p2.name = "Fitness Item"
      p2.category = "Fitness"
      p2.price = 29.99
      p2.save

      results = Product.where("category = ?", ["Electronics"])

      expect(results.length).to eq(1)
      expect(results[0].name).to eq("Electronics Item")
    end
  end
```

---

6. Testing Authentication

```
require "tina4"

RSpec.describe "Authentication" do
  let(:client) { Tina4::TestClient.new }

  before(:each) do
    # Register a test user
    client.post("/api/register", {
      name: "Test User",
      email: "test@example.com",
      password: "securePass123"
    })
  end

  it "logs in with valid credentials" do
    result = client.post("/api/login", {
      email: "test@example.com",
      password: "securePass123"
    })

    expect(result.status).to eq(200)
    expect(result.json).to have_key("token")
    expect(result.json["user"]["email"]).to eq("test@example.com")
  end

  it "rejects invalid password" do
    result = client.post("/api/login", {
      email: "test@example.com",
      password: "wrongPassword"
    })

    expect(result.status).to eq(401)
    expect(result.json["error"]).to include("Invalid")
  end

  it "protects routes without token" do
    result = client.get("/api/profile")

    expect(result.status).to eq(401)
  end

  it "allows access with valid token" do
    login = client.post("/api/login", {
      email: "test@example.com",
      password: "securePass123"
    })

    token = login.json["token"]

    result = client.get("/api/profile", headers: {
      "Authorization" => "Bearer #{token}"
    })

    expect(result.status).to eq(200)
    expect(result.json["email"]).to eq("test@example.com")
  end
end
```

---

7. Test Database Isolation

Use a separate test database to avoid polluting development data:

```
# .env.test
TINA4_DATABASE_URL=sqlite:///data/test.db
TINA4_DEBUG=false
```

In your test helper:

```
# tests/spec_helper.rb
ENV["TINA4_ENV"] = "test"
require "tina4"

RSpec.configure do |config|
  config.before(:suite) do
    # Run migrations on test database
    system("tina4 migrate --env test")
  end

  config.after(:suite) do
    # Clean up test database
    File.delete("data/test.db") if File.exist?("data/test.db")
  end
end
```

---

8. Setup and Teardown

RSpec's `before` and `after` hooks handle test fixture management. Use them to create and clean up test data:

```
RSpec.describe "User Management" do
  let(:client) { Tina4::TestClient.new }

  before(:each) do
    # Runs before every test
    @user = User.new(name: "Test User", email: "test@example.com")
    @user.save
    @user_id = @user.id
  end

  after(:each) do
    # Runs after every test, regardless of pass/fail
    User.find(@user_id)&.delete rescue nil
  end

  it "loads the user" do
    loaded = User.find(@user_id)
    expect(loaded.name).to eq("Test User")
  end

  it "updates the user" do
    user = User.find(@user_id)
    user.name = "Updated Name"
```

```
    user.save

    reloaded = User.find(@user_id)
    expect(reloaded.name).to eq("Updated Name")
  end
end
```

`before(:each)` runs before every test method. `after(:each)` runs after every test method, regardless of whether the test passed or failed. This keeps tests isolated -- each test starts with a clean state.

Use `before(:all)` and `after(:all)` for expensive setup that applies to the entire describe block (like creating a database). But prefer `before(:each)` for data -- shared state between tests causes flaky results.

---

9. Running Tests

```
# Run all tests
tina4 test

# Run a specific test file
tina4 test tests/product_spec.rb

# Run with verbose output
tina4 test --verbose

# Using RSpec directly
bundle exec rspec tests/
bundle exec rspec tests/product_spec.rb
bundle exec rspec tests/ --format documentation
```

---

10. Testing Best Practices

Test One Thing Per Test

Each test should verify one behavior. If it fails, you know exactly what broke.

```
# Good: each test verifies one thing
it "returns 201 on create" do
  result = client.post("/api/products", { name: "Widget", price: 9.99 })
  expect(result.status).to eq(201)
end

it "returns the created product" do
  result = client.post("/api/products", { name: "Widget", price: 9.99 })
  expect(result.json["name"]).to eq("Widget")
end

# Avoid: testing multiple unrelated things in one test
it "does everything" do
  # Creates, reads, updates, deletes, checks auth, validates input...
  # When this fails, you do not know which part broke
end
```

Use Descriptive Test Names

```
# Good: tells you what the test verifies
it "returns 404 when product does not exist" do
  # ...
end

# Bad: vague name
it "works" do
  # ...
end
```

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
it "deletes a product" do
  product = Product.new(name: "Temporary", price: 1.00)
  product.save

  product.delete

  check = Product.find(product.id)
  expect(check).to be_nil
end
```

Code Coverage

RSpec integrates with `simplecov` for coverage reports:

```
# Add to Gemfile
gem "simplecov", require: false, group: :test

# Add to spec/spec_helper.rb (before any other require)
require "simplecov"
SimpleCov.start
```

```
bundle exec rspec
```

```
Coverage report generated. 88.5% covered.
```

Open `coverage/index.html` in your browser for a visual breakdown of which lines are covered.

---

11. Exercise: Write Tests for a Notes API

Write a comprehensive test suite for the Notes API from Chapter 5.

Requirements

Test these scenarios:

- Create a note with valid data (201)
- Create a note with missing title (400)

- List all notes (200)
- Get a note by ID (200)
- Get a non-existent note (404)
- Update a note (200)
- Delete a note (204)
- Search notes by content
- Filter notes by tag

---

12. Solution

Create `tests/notes_spec.rb`:

```
require "tina4"

RSpec.describe "Notes API" do
  let(:client) { Tina4::TestClient.new }

  before(:each) do
    db = Tina4.database
    db.execute("DELETE FROM notes")
  end

  it "creates a note with valid data" do
    result = client.post("/api/notes", {
      title: "Test Note",
      content: "This is a test",
      tag: "testing"
    })

    expect(result.status).to eq(201)
    expect(result.json["title"]).to eq("Test Note")
    expect(result.json["tag"]).to eq("testing")
    expect(result.json["id"]).not_to be_nil
  end

  it "rejects note with missing title" do
    result = client.post("/api/notes", {
      content: "No title here"
    })

    expect(result.status).to eq(400)
    expect(result.json["errors"]).to include("Title is required")
  end

  it "lists all notes" do
    client.post("/api/notes", { title: "Note 1", content: "Content 1" })
    client.post("/api/notes", { title: "Note 2", content: "Content 2" })

    result = client.get("/api/notes")

    expect(result.status).to eq(200)
    expect(result.json["count"]).to eq(2)
```

```
  expect(result.json["notes"].length).to eq(2)
end

it "gets a note by ID" do
  created = client.post("/api/notes", { title: "Find Me", content: "Here I am" })
  id = created.json["id"]

  result = client.get("/api/notes/#{id}")

  expect(result.status).to eq(200)
  expect(result.json["title"]).to eq("Find Me")
end

it "returns 404 for non-existent note" do
  result = client.get("/api/notes/99999")

  expect(result.status).to eq(404)
end

it "updates a note" do
  created = client.post("/api/notes", { title: "Original", content: "Original content" })
  id = created.json["id"]

  result = client.put("/api/notes/#{id}", { title: "Updated", content: "Updated content" })

  expect(result.status).to eq(200)
  expect(result.json["title"]).to eq("Updated")
end

it "deletes a note" do
  created = client.post("/api/notes", { title: "Delete Me", content: "Goodbye" })
  id = created.json["id"]

  result = client.delete("/api/notes/#{id}")

  expect(result.status).to eq(204)

  get_result = client.get("/api/notes/#{id}")
  expect(get_result.status).to eq(404)
end

it "searches notes by content" do
  client.post("/api/notes", { title: "Shopping", content: "Buy milk and eggs" })
  client.post("/api/notes", { title: "Work", content: "Finish the report" })

  result = client.get("/api/notes?search=milk")

  expect(result.status).to eq(200)
  expect(result.json["count"]).to eq(1)
  expect(result.json["notes"][0]["title"]).to eq("Shopping")
end

it "filters notes by tag" do
  client.post("/api/notes", { title: "Personal", content: "Content", tag: "personal" })
  client.post("/api/notes", { title: "Work", content: "Content", tag: "work" })

  result = client.get("/api/notes?tag=personal")

  expect(result.status).to eq(200)
```

```
    expect(result.json["count"]).to eq(1)
    expect(result.json["notes"][0]["title"]).to eq("Personal")
  end
end
```

---

13. Gotchas

1. Tests Share Database State

Problem: Tests pass individually but fail when run together.

Fix: Clean up in `before(:each)` blocks. Delete test data before each test.

2. TestClient Does Not Start a Real Server

Problem: External services (SMTP, Redis) are not available in tests.

Fix: Mock external services or use test doubles. The TestClient simulates HTTP requests without network I/O.

3. Test Order Dependency

Problem: Test B depends on data created by Test A.

Fix: Each test should set up its own data. Use `before(:each)` blocks to create required state.

4. Database Migrations Not Applied

Problem: Tests fail with "table does not exist" errors.

Fix: Run `tina4 migrate` before running tests, or add migration logic to your test setup.

5. Token Expired During Test

Problem: Auth tests fail intermittently with "token expired".

Fix: Set a long JWT expiry for tests: `TINA4_TOKEN_LIMIT=86400` in `.env.test`.

6. Floating Point Comparison

Problem: `expect(product.price).to eq(9.99)` fails with `9.990000000000001`.

Fix: Use `be_within` for floating point: `expect(product.price).to be_within(0.01).of(9.99)`.

7. Test Output Too Verbose

Problem: Test output includes log messages from the application.

Fix: Set `TINA4_LOG_LEVEL=ERROR` in `.env.test` to suppress info and debug logs.

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.

```
tina4ruby generate crud Product --fields "name:string,price:float"
```

That single command creates six files:

#	File	Purpose
1	`src/orm/product.rb`	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.rb`	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	`spec/products_spec.rb`	RSpec stubs for all CRUD operations

What Each File Contains

The **model** maps the `product` table to a Ruby class:

```
require "tina4"

class Product < Tina4::ORM
  table_name "product"

  field :id, :integer
```

```
  field :name, :string
  field :price, :float
  field :created_at, :datetime
  field :updated_at, :datetime
end
```

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:

```
require "tina4"
require_relative "../orm/product"

Tina4.get "/api/products", description: "List all products" do |request, response|
  products = Product.new.select
  response.call(products)
end

Tina4.get "/api/products/:id", description: "Get a product by ID" do |request, response|
  product = Product.new
  product.id = request.params[:id]
  product.load
  response.call(product)
end

Tina4.post "/api/products", description: "Create a product" do |request, response|
  product = Product.new(request.body)
  product.save
  response.call(product, 201)
end

Tina4.put "/api/products/:id", description: "Update a product" do |request, response|
  product = Product.new(request.body)
  product.id = request.params[:id]
  product.save
  response.call(product)
end

Tina4.delete "/api/products/:id", description: "Delete a product" do |request, response|
  product = Product.new
  product.id = request.params[:id]
  product.delete
  response.call(nil, 204)
end
```

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:

```
require "tina4"

RSpec.describe "Products API" do
  let(:client) { Tina4::App.new.test_client }

  it "creates a product" do
    response = client.post("/api/products", { name: "Widget", price: 9.99 })
    expect(response.status).to eq(201)
  end

  it "lists products" do
    response = client.get("/api/products")
    expect(response.status).to eq(200)
  end

  it "gets a product" do
    response = client.get("/api/products/1")
    expect(response.status).to eq(200)
  end

  it "updates a product" do
    response = client.put("/api/products/1", { name: "Updated Widget" })
    expect(response.status).to eq(200)
  end

  it "deletes a product" do
    response = client.delete("/api/products/1")
    expect(response.status).to eq(204)
  end
end
```

Run It

After generating, run the migration and start the server:

```
tina4ruby migrate
tina4ruby serve
```

Open Swagger UI at `http://localhost:7147/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

```
tina4ruby generate model Product --fields "name:string,price:float"
```

Creates three files: the ORM model (`src/orm/product.rb`), the UP migration, and the DOWN migration. No routes, no templates, no tests.

Route

```
tina4ruby generate route products --model Product
```

Creates one file: `src/routes/products.rb` with CRUD endpoints and Swagger annotations. The model must exist first.

Migration

```
tina4ruby 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

```
tina4ruby generate middleware AuthLog
```

Creates one file with before and after stubs:

```
require "tina4"

Tina4.middleware :before do |request|
  puts "Request: #{request.method} #{request.url}"
  request
end

Tina4.middleware :after do |request, response|
  puts "Response: #{response.status}"
  response
end
```

Test

```
tina4ruby generate test products --model Product
```

Creates one file: `spec/products_spec.rb` with RSpec CRUD stubs.

Form

```
tina4ruby generate form Product --fields "name:string,price:float"
```

Creates one file: `src/templates/products/form.html` with typed inputs and `form_token`.

View

```
tina4ruby generate view Product --fields "name:string,price:float"
```

Creates two templates: a list view and a detail view in `src/templates/products/`.

CRUD

```
tina4ruby generate crud Product --fields "name:string,price:float"
```

Shorthand for running all generators at once: model, migration, route, form, view, and test.

Auth

```
tina4ruby 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

```
Tina4::AutoCrud.register(User)
Tina4::AutoCrud.generate_routes(prefix: "/api")
```

`AutoCrud.register` wires up a single model. `generate_routes` mounts the five standard endpoints automatically — no route files needed.

The Auth Generator

Authentication needs more than one file. The auth generator creates seven:

```
tina4ruby generate auth
```

#	File	Purpose
1	`src/orm/user.rb`	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.rb`	Login, register, logout routes
5	`src/templates/auth/login.html`	Login form

6	`src/templates/auth/register.html`	Registration form
7	`spec/auth_spec.rb`	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:

```
tina4ruby migrate
tina4ruby 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:

```
tina4ruby generate model Product --fields "name:string,price:float" --with-route --with-migratio
```

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.

```
tina4ruby generate auth
```

Step 2: Scaffold the Post resource.

```
tina4ruby generate crud Post --fields "title:string,body:text,published:bool"
```

Step 3: Scaffold the Category resource.

```
tina4ruby generate crud Category --fields "name:string,description:text"
```

Step 4: Add a migration to link posts to categories.

```
tina4ruby 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.

```
tina4ruby migrate
tina4ruby 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 `tina4ruby migrate`. Generate and migrate are separate steps by design.

File Naming Matters

The generator derives file names from the model name. `Product` becomes `product.rb` for the model and `products.rb` for routes. Do not rename generated files unless you update all requires.

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.

API Documentation with Swagger

1. The 47-Endpoint Problem

Your team has 47 API endpoints. The frontend developer keeps asking "what does this endpoint accept?" You email a spreadsheet. It goes stale. You write a wiki page. Nobody touches it. You add comments to the code. Nobody reads them.

Swagger (OpenAPI) solves this for good. It generates interactive API documentation from annotations in your route files. The docs stay current because they live in the code. Your frontend developer browses every endpoint, sees expected request and response formats, and tests endpoints from the browser.

Tina4 auto-generates a Swagger UI at `/swagger` from comment annotations on your routes. No build step. No extra tooling. Write the annotations. The documentation appears.

---

2. What Swagger/OpenAPI Is

OpenAPI is a specification format for describing REST APIs. Swagger is the toolset that reads OpenAPI specs and generates documentation, client SDKs, and server stubs.

An OpenAPI spec describes:

- Every endpoint (path + HTTP method)
- What parameters each endpoint accepts (path, query, header, body)
- What each endpoint returns (response codes, response bodies)
- Data schemas (what a "User" or "Product" object looks like)
- Authentication requirements
- Grouping and tagging

Tina4 builds this spec automatically from comment annotations in your Ruby code. You never write JSON or YAML by hand.

---

3. Enabling Swagger

Swagger is available out of the box when `TINA4_DEBUG=true`. Navigate to:

```
http://localhost:7147/swagger
```

You should see the Swagger UI with any routes you have already defined. If you have not added any Swagger annotations yet, you will see the routes listed with default descriptions.

For production, you can explicitly enable or disable Swagger:

```
TINA4_SWAGGER_ENABLED=true
```

The Swagger JSON Endpoint

The raw OpenAPI spec is available at:

```
http://localhost:7147/swagger/json
```

```
curl http://localhost:7147/swagger/json
```

```
{
  "openapi": "3.0.0",
  "info": {
    "title": "My Store API",
    "version": "1.0.0"
  },
  "paths": {
    "/api/products": {
      "get": {
        "summary": "List all products",
        "responses": {
          "200": {
            "description": "Successful response"
          }
        }
      }
    }
  }
}
```

---

4. Adding Descriptions to Routes

Add Swagger annotations as comments above your route definitions:

```
# List all products
# @description Returns a paginated list of all products in the catalog
# @tags Products
Tina4::Router.get("/api/products") do |request, response|
  response.json({ products: [] })
end
```

The first comment line becomes the `summary`. The `@description` tag provides a longer explanation.

Documenting Path Parameters

```
# Get a product by ID
# @description Returns a single product with full details including inventory status
# @tags Products
# @param int $id The unique product identifier
Tina4::Router.get("/api/products/{id:int}") do |request, response|
  id = request.params["id"]
  response.json({ id: id, name: "Wireless Keyboard", price: 79.99 })
end
```

Documenting Query Parameters

```
# Search products
# @description Search the product catalog by name, category, or price range
# @tags Products
# @query string $q Search query (searches product name and description)
# @query string $category Filter by category name
# @query float $min_price Minimum price filter
# @query float $max_price Maximum price filter
# @query int $page Page number (default: 1)
# @query int $limit Items per page (default: 20, max: 100)
Tina4::Router.get("/api/products/search") do |request, response|
  q = request.params["q"] || ""
  page = (request.params["page"] || 1).to_i
  limit = [(request.params["limit"] || 20).to_i, 100].min

  response.json({
    query: q,
    page: page,
    limit: limit,
    results: [],
    total: 0
  })
end
```

---

5. Documenting Request and Response Schemas

Request Body

```
# Create a new product
# @description Creates a product in the catalog. Requires admin authentication.
# @tags Products
# @body {"name": "string", "category": "string", "price": "float", "in_stock": "bool", "descript
# @response 201 {"id": "int", "name": "string", "category": "string", "price": "float", "in_stoc
# @response 400 {"error": "string"}
Tina4::Router.post("/api/products") do |request, response|
  body = request.body

  if body["name"].nil? || body["name"].empty?
    return response.json({ error: "Name is required" }, 400)
  end

  response.json({
    id: 1,
    name: body["name"],
    category: body["category"] || "Uncategorized",
    price: (body["price"] || 0).to_f,
    in_stock: body["in_stock"] != false,
    created_at: Time.now.iso8601
  }, 201)
end
```

---

6. Tags for Grouping Endpoints

Tags group related endpoints in the Swagger UI:

```
# List all users
# @tags Users
Tina4::Router.get("/api/users") do |request, response|
  response.json({ users: [] })
end

# List all orders
# @tags Orders
Tina4::Router.get("/api/orders") do |request, response|
  response.json({ orders: [] })
end
```

In the Swagger UI, you will see sections for "Users" and "Orders". An endpoint can belong to multiple groups:

```
# Get user's orders
# @tags Users, Orders
Tina4::Router.get("/api/users/{id:int}/orders") do |request, response|
  response.json({ orders: [] })
end
```

---

7. Example Values

Add example values to make the docs more useful:

```
# Create a new product
# @description Creates a product in the catalog
# @tags Products
# @example request {"name": "Ergonomic Keyboard", "category": "Electronics", "price": 89.99, "in
# @example response {"id": 42, "name": "Ergonomic Keyboard", "category": "Electronics", "price":
Tina4::Router.post("/api/products") do |request, response|
  body = request.body

  response.json({
    id: 42,
    name: body["name"],
    category: body["category"] || "Uncategorized",
    price: (body["price"] || 0).to_f,
    in_stock: body["in_stock"] != false,
    created_at: Time.now.iso8601
  }, 201)
end
```

---

8. Try-It-Out from the Swagger UI

The Swagger UI includes a "Try it out" button on every endpoint. Clicking it:

- Expands the endpoint with editable input fields
- Pre-fills example values (if provided)

- Lets you edit the parameters, headers, and request body
- Sends the actual HTTP request to your running server
- Shows the response status, headers, and body

This is a live testing tool built into your documentation.

Authentication in Try-It-Out

If your endpoints require authentication, click the "Authorize" button at the top of the Swagger UI. Enter your JWT token or API key, and all subsequent "Try it out" requests will include the authentication header.

---

9. Customizing the Swagger Info Block

Configure the top-level API information in `.env`:

```
TINA4_SWAGGER_TITLE=My Store API
TINA4_SWAGGER_DESCRIPTION=API for managing products, orders, and users
TINA4_SWAGGER_VERSION=1.0.0
TINA4_SWAGGER_CONTACT_EMAIL=api@mystore.com
TINA4_SWAGGER_LICENSE=MIT
```

---

10. Generating Client SDKs from the Spec

The OpenAPI spec at `/swagger/json` can be used with code generation tools:

```
# Install the OpenAPI Generator CLI
npm install -g @openapitools/openapi-generator-cli

# Generate a TypeScript client
openapi-generator-cli generate \
  -i http://localhost:7147/swagger/json \
  -g typescript-fetch \
  -o ./frontend/api-client
```

---

11. A Complete Documented API

Here is a full example showing all the annotation features together:

```
# List all users
# @description Returns a paginated list of users. Supports filtering by role and searching by na
# @tags Users
# @query int $page Page number (default: 1)
# @query int $limit Items per page (default: 20)
# @query string $role Filter by role (admin, user, moderator)
# @query string $search Search by name or email
# @response 200 {"users": [{"id": "int", "name": "string", "email": "string", "role": "string"}]
# @example response {"users": [{"id": 1, "name": "Alice", "email": "alice@example.com", "role":
Tina4::Router.get("/api/users") do |request, response|
  page = (request.params["page"] || 1).to_i
```

```
  limit = (request.params["limit"] || 20).to_i

  response.json({
    users: [
      { id: 1, name: "Alice", email: "alice@example.com", role: "admin" },
      { id: 2, name: "Bob", email: "bob@example.com", role: "user" }
    ],
    total: 42,
    page: page,
    pages: (42.0 / limit).ceil
  })
end

# Create a new user
# @description Creates a user account. Email must be unique.
# @tags Users
# @body {"name": "string", "email": "string", "password": "string", "role": "string"}
# @response 201 {"id": "int", "name": "string", "email": "string", "role": "string", "created_at
# @response 400 {"errors": ["string"]}
# @response 409 {"error": "string"}
# @example request {"name": "Charlie", "email": "charlie@example.com", "password": "securePass12
# @example response {"id": 3, "name": "Charlie", "email": "charlie@example.com", "role": "user",
Tina4::Router.post("/api/users") do |request, response|
  body = request.body

  errors = []
  errors << "Name is required" if body["name"].nil? || body["name"].empty?
  errors << "Email is required" if body["email"].nil? || body["email"].empty?
  errors << "Password is required" if body["password"].nil? || body["password"].empty?

  unless errors.empty?
    return response.json({ errors: errors }, 400)
  end

  response.json({
    id: 3,
    name: body["name"],
    email: body["email"],
    role: body["role"] || "user",
    created_at: Time.now.iso8601
  }, 201)
end
```

---

12. Exercise: Document a Complete User API

Take the User API from the example above and extend it with the following endpoints. Write full Swagger annotations for each one.

Requirements

Document these additional endpoints (you can use hardcoded data in the handlers):

Method	Path	Description

PUT	/api/users/{id}	Update a user. Body: name, email, role. Response: updated user.
GET	/api/users/{id}/orders	List a user's orders. Query: status filter, pagination.
POST	/api/users/{id}/avatar	Upload user avatar. Body: avatar_url string.

Each endpoint should have:

- A summary (first comment line)
- A `@description`
- A `@tags` annotation
- `@param` for path parameters
- `@query` for query parameters (where applicable)
- `@body` for request body (where applicable)
- `@response` for each possible response code
- `@example` for request and response (where applicable)

Test by visiting:

```
http://localhost:7147/swagger
```

---

13. Solution

Create `src/routes/user_api_documented.rb`:

```
# Update a user
# @description Updates an existing user's profile information. Only provided fields are updated.
# @tags Users
# @param int $id User ID
# @body {"name": "string", "email": "string", "role": "string"}
# @response 200 {"id": "int", "name": "string", "email": "string", "role": "string", "updated_at
# @response 404 {"error": "string"}
# @example request {"name": "Alice Smith", "email": "alice.smith@example.com", "role": "admin"}
# @example response {"id": 1, "name": "Alice Smith", "email": "alice.smith@example.com", "role":
Tina4::Router.put("/api/users/{id:int}") do |request, response|
  id = request.params["id"]
  body = request.body

  if id > 100
    return response.json({ error: "User not found" }, 404)
  end

  response.json({
    id: id,
    name: body["name"] || "Alice",
```

```
    email: body["email"] || "alice@example.com",
    role: body["role"] || "user",
    updated_at: Time.now.iso8601
  })
end

# List user orders
# @description Returns a paginated list of orders for a specific user.
# @tags Users, Orders
# @param int $id User ID
# @query string $status Filter by order status (pending, processing, shipped, delivered, cancell
# @query int $page Page number (default: 1)
# @query int $limit Items per page (default: 20)
# @response 200 {"orders": [{"id": "int", "product": "string", "quantity": "int", "total": "floa
# @response 404 {"error": "string"}
# @example response {"orders": [{"id": 101, "product": "Wireless Keyboard", "quantity": 2, "tota
Tina4::Router.get("/api/users/{id:int}/orders") do |request, response|
  id = request.params["id"]
  status = request.params["status"]
  page = (request.params["page"] || 1).to_i

  if id > 100
    return response.json({ error: "User not found" }, 404)
  end

  orders = [
    { id: 101, product: "Wireless Keyboard", quantity: 2, total: 159.98, status: "shipped" },
    { id: 102, product: "USB-C Hub", quantity: 1, total: 49.99, status: "delivered" }
  ]

  orders = orders.select { |o| o[:status] == status } if status

  response.json({ orders: orders, total: orders.length, page: page })
end

# Upload user avatar
# @description Sets or updates the avatar URL for a user.
# @tags Users
# @param int $id User ID
# @body {"avatar_url": "string"}
# @response 200 {"id": "int", "avatar_url": "string", "updated_at": "string"}
# @response 400 {"error": "string"}
# @response 404 {"error": "string"}
# @example request {"avatar_url": "https://cdn.example.com/avatars/alice-2026.jpg"}
# @example response {"id": 1, "avatar_url": "https://cdn.example.com/avatars/alice-2026.jpg", "u
Tina4::Router.post("/api/users/{id:int}/avatar") do |request, response|
  id = request.params["id"]
  body = request.body

  if id > 100
    return response.json({ error: "User not found" }, 404)
  end

  if body["avatar_url"].nil? || body["avatar_url"].empty?
    return response.json({ error: "avatar_url is required" }, 400)
  end

  response.json({
    id: id,
```

```
    avatar_url: body["avatar_url"],
    updated_at: Time.now.iso8601
  })
end
```

---

14. Gotchas

1. Annotations Must Be Directly Above the Route

Problem: Your Swagger annotations do not appear in the docs.

Cause: There is a blank line or other code between the comment block and the `Tina4::Router` call.

Fix: Make sure the comments are on the lines directly before the route definition with no blank lines in between.

2. Missing @tags Makes Endpoints Hard to Find

Problem: All endpoints appear in one giant flat list in the Swagger UI.

Fix: Add `# @tags ResourceName` to every route comment block.

3. @body Must Be Valid JSON

Problem: The Swagger UI shows the body schema as empty or broken.

Fix: Validate your `@body` JSON. Every key and string value must be in double quotes.

4. Swagger Shows Routes You Did Not Annotate

Problem: Unannotated routes appear in the Swagger UI with minimal documentation.

Cause: Tina4 includes all registered routes in the Swagger spec. Add `# @hidden` to hide a route.

5. Response Examples Do Not Match Actual Responses

Problem: The example response in Swagger shows different fields than the actual API response.

Fix: Treat annotations as part of the code. When you change a handler's response format, update the annotations.

6. Swagger UI Not Available in Production

Problem: `/swagger` returns a 404 in production.

Fix: If you want Swagger in production, explicitly set `TINA4_SWAGGER_ENABLED=true` in your `.env`.

7. SDK Generation Produces Incorrect Types

Problem: The generated TypeScript client has `any` types instead of proper interfaces.

Fix: Use correct OpenAPI type format in your annotations: `"name": "string"` (lowercase), `"price": "number"`, `"in_stock": "boolean"`.

API Client

1. Talking to Other Services

Your app does not live in isolation. It calls payment gateways, weather APIs, shipping providers, CRM platforms, and internal microservices. Each call needs a base URL, auth headers, timeout handling, and response parsing.

Tina4's API client handles that boilerplate. One configured instance gives you clean `get`, `post`, `put`, and `delete` calls with a consistent response format and no external gem required.

---

2. Creating a Client

```
client = Tina4::Api.new("https://api.example.com")
```

All requests use this base URL. Paths are appended to it.

With Default Headers

```
client = Tina4::Api.new("https://api.example.com", {
  "Content-Type" => "application/json",
  "Accept"       => "application/json"
})
```

Headers set here are sent with every request.

---

3. GET Requests

```
result = client.get("/products")

if result.success?
  puts result.body.inspect
else
  puts "Error #{result.status}: #{result.body}"
end
```

With Query Parameters

```
result = client.get("/products", params: { category: "keyboards", in_stock: true })
# Requests: GET /products?category=keyboards&in_stock=true
```

Response Object

Every call returns a response object with:

- `result.success?` -- true if HTTP status is 2xx
- `result.status` -- integer HTTP status code
- `result.body` -- parsed JSON (Hash or Array) or raw string
- `result.headers` -- response headers Hash

---

4. POST Requests

```
result = client.post("/orders", body: {
  email: "alice@example.com",
  items: [{ sku: "KB-100", qty: 1 }],
  total: 79.99
})

if result.success?
  order_id = result.body["id"]
  puts "Order created: #{order_id}"
else
  puts "Failed: #{result.body["message"]}"
end
```

The body is serialized to JSON automatically. The `Content-Type: application/json` header is set if not already present.

---

5. PUT Requests

```
result = client.put("/orders/101", body: {
  status: "shipped",
  tracking_number: "1Z999AA10123456784"
})

puts result.success? ? "Updated" : "Failed: #{result.status}"
```

---

6. DELETE Requests

```
result = client.delete("/orders/101")

if result.success?
  puts "Order deleted"
else
  puts "Delete failed: #{result.status}"
end
```

---

7. Authentication Headers

Bearer Token

```
client = Tina4::Api.new("https://api.example.com", {
  "Authorization" => "Bearer #{ENV['API_TOKEN']}",
  "Content-Type"  => "application/json"
})
```

API Key Header

```
client = Tina4::Api.new("https://api.stripe.com", {
  "Authorization" => "Bearer #{ENV['STRIPE_SECRET_KEY']}",
  "Stripe-Version" => "2024-11-20"
})
```

Basic Auth

```
require "base64"

credentials = Base64.strict_encode64("#{ENV['API_USER']}:#{ENV['API_PASS']}")

client = Tina4::Api.new("https://api.example.com", {
  "Authorization" => "Basic #{credentials}"
})
```

Per-Request Header Override

Pass headers directly on a single call to override or extend the defaults:

```
result = client.get("/admin/users", headers: {
  "X-Admin-Token" => "secret-admin-key"
})
```

---

8. Timeouts

```
client = Tina4::Api.new("https://api.slow.com", {}, timeout: 10)
# Raises Tina4::ApiTimeout after 10 seconds with no response
```

Default timeout is 30 seconds.

---

9. Using the Client in Route Handlers

```
# @noauth
Tina4::Router.get("/api/weather") do |request, response|
  city = request.params["city"] || "London"

  weather_client = Tina4::Api.new("https://api.openweathermap.org", {
    "Accept" => "application/json"
  })

  result = weather_client.get("/data/2.5/weather", params: {
    q:     city,
    appid: ENV["OPENWEATHER_API_KEY"],
    units: "metric"
  })

  if result.success?
    data = result.body
    response.json({
      city:        data["name"],
      temperature: data.dig("main", "temp"),
      description: data.dig("weather", 0, "description")
    })
  else
    response.json({ error: "Weather data unavailable" }, 502)
  end
end
```

```
curl "http://localhost:7147/api/weather?city=Paris"
```

```
{
  "city": "Paris",
  "temperature": 14.3,
  "description": "partly cloudy"
}
```

---

10. Shared Client Instances

Define shared clients once and reuse them across routes.

```
# src/clients/stripe.rb
STRIPE = Tina4::Api.new("https://api.stripe.com", {
  "Authorization"  => "Bearer #{ENV['STRIPE_SECRET_KEY']}",
  "Content-Type"   => "application/json",
  "Stripe-Version" => "2024-11-20"
})

# src/routes/payments.rb
Tina4::Router.post("/api/checkout") do |request, response|
  body = request.body

  result = STRIPE.post("/v1/payment_intents", body: {
    amount:   (body["total"].to_f * 100).to_i,
    currency: "usd",
    metadata: { order_id: body["order_id"] }
  })
```

```
  if result.success?
    response.json({ client_secret: result.body["client_secret"] }, 201)
  else
    Tina4::Log.error("Stripe error", status: result.status, error: result.body["error"]["message
    response.json({ error: "Payment failed" }, 502)
  end
end
```

---

11. Error Handling

```
begin
  result = client.get("/products")

  case result.status
  when 200..299
    process(result.body)
  when 401
    raise "Unauthorized -- check API credentials"
  when 429
    Tina4::Log.warning("Rate limited", retry_after: result.headers["Retry-After"])
  when 500..599
    raise "Remote server error: #{result.status}"
  end

rescue Tina4::ApiTimeout => e
  Tina4::Log.error("API request timed out", error: e.message)
rescue => e
  Tina4::Log.error("API request failed", error: e.message)
end
```

---

12. Gotchas

1. Never hard-code credentials

Always read tokens from environment variables. Hard-coded credentials end up in version control.

2. Check success? before reading body

`result.body` on a 4xx or 5xx response contains the error payload, not the data you expect.

3. Timeout is per-request

A shared client instance with `timeout: 5` applies that timeout to every request. Override per call if some endpoints are slower than others.

4. Base URL trailing slash

`Tina4::Api.new("https://api.example.com/")` with a trailing slash and `client.get("/products")` with a leading slash double up to `https://api.example.com//products`. Use a base URL without a trailing slash.

GraphQL and SOAP

1. The Problem GraphQL Solves

Your mobile app needs a product list. 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 gems. No Apollo Server. No graphql-ruby gem. 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. Set up a schema and register the route in `src/routes/graphql.rb`:

```
schema = Tina4::GraphQLSchema.new
gql = Tina4::GraphQL.new(schema)
gql.register_route  # POST /graphql, GET /graphql (playground)
```

To change the endpoint path:

```
gql.register_route("/api/graphql")
```

To verify it is working, start the server and send a test query:

```
curl -X POST http://localhost:7147/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 Queries

Register queries on a `GraphQLSchema` instance. Each query has a name, return type, optional arguments, and a resolver block. The block receives three arguments: `root` (parent value), `args` (query arguments), and `ctx` (context hash).

Create `src/routes/graphql.rb`:

```
schema = Tina4::GraphQLSchema.new

# List all products
schema.add_query("products", type: "[Product]") do |root, args, ctx|
  db = Tina4.database
  products = db.fetch("SELECT * FROM products ORDER BY name")
  products
end

# Get a single product by ID
schema.add_query("product", type: "Product",
                 args: { "id" => { type: "Int!" } }) do |root, args, ctx|
  db = Tina4.database
  db.fetch_one("SELECT * FROM products WHERE id = ?", [args["id"].to_i])
end

gql = Tina4::GraphQL.new(schema)
gql.register_route
```

Testing the Queries

List all products:

```
curl -X POST http://localhost:7147/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}
    ]
  }
}
```

The response contains only the four fields you 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:7147/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:7147/graphql \
  -H "Content-Type: application/json" \
  -d '{"query": "{ product(id: 999) { id name } }"}'
```

```
{
  "data": {
    "product": null
  }
}
```

---

5. Mutations

Mutations are how clients create, update, or delete data. Register them with `add_mutation` on the schema.

```
schema.add_mutation("createProduct", type: "Product",
```

```
                    args: {
                      "name"     => { type: "String!" },
                      "category" => { type: "String" },
                      "price"    => { type: "Float!" },
                      "inStock"  => { type: "Boolean" }
                    }) do |root, args, ctx|
  db = Tina4.database

  db.execute(
    "INSERT INTO products (name, category, price, in_stock) VALUES (:name, :category, :price, :i
    {
      name: args["name"],
      category: args["category"] || "Uncategorized",
      price: args["price"].to_f,
      in_stock: args["inStock"] ? 1 : 0
    }
  )

  db.fetch_one("SELECT * FROM products WHERE id = last_insert_rowid()")
end

schema.add_mutation("updateProduct", type: "Product",
                    args: {
                      "id"       => { type: "Int!" },
                      "name"     => { type: "String" },
                      "category" => { type: "String" },
                      "price"    => { type: "Float" },
                      "inStock"  => { type: "Boolean" }
                    }) do |root, args, ctx|
  db = Tina4.database
  id = args["id"].to_i

  sets = []
  params = []

  if args.key?("name")
    sets << "name = ?"
    params << args["name"]
  end
  if args.key?("category")
    sets << "category = ?"
    params << args["category"]
  end
  if args.key?("price")
    sets << "price = ?"
    params << args["price"].to_f
  end
  if args.key?("inStock")
    sets << "in_stock = ?"
    params << (args["inStock"] ? 1 : 0)
  end

  unless sets.empty?
    params << id
    db.execute("UPDATE products SET #{sets.join(', ')} WHERE id = ?", params)
  end

  db.fetch_one("SELECT * FROM products WHERE id = ?", [id])
end
```

```
schema.add_mutation("deleteProduct", type: "Boolean",
                    args: { "id" => { type: "Int!" } }) do |root, args, ctx|
  db = Tina4.database
  id = args["id"].to_i
  db.execute("DELETE FROM products WHERE id = ?", [id])
  true
end
```

Testing Mutations

Create a product:

```
curl -X POST http://localhost:7147/graphql \
  -H "Content-Type: application/json" \
  -d '{
    "query": "mutation { createProduct(name: \"Desk Lamp\", category: \"Office\", price: 39.99)
  }'
```

```
{
  "data": {
    "createProduct": {
      "id": 4,
      "name": "Desk Lamp",
      "price": 39.99
    }
  }
}
```

Update a product:

```
curl -X POST http://localhost:7147/graphql \
  -H "Content-Type: application/json" \
  -d '{
    "query": "mutation { updateProduct(id: 4, price: 44.99, inStock: false) { id name price inSt
  }'
```

```
{
  "data": {
    "updateProduct": {
      "id": 4,
      "name": "Desk Lamp",
      "price": 44.99,
      "inStock": false
    }
  }
}
```

Delete a product:

```
curl -X POST http://localhost:7147/graphql \
  -H "Content-Type: application/json" \
  -d '{
    "query": "mutation { deleteProduct(id: 4) }"
  }'
```

```
{
  "data": {
    "deleteProduct": true
  }
}
```

---

6. Nested Types and Relationships

The real power of GraphQL: traversing relationships in a single query. Authors, posts, comments -- all in one request.

```
schema.add_query("posts", type: "[Post]") do |root, args, ctx|
  db = Tina4.database
  posts = db.fetch("SELECT * FROM posts WHERE published = 1 ORDER BY created_at DESC")

  posts.each do |post|
    post["author"] = db.fetch_one(
      "SELECT id, name, email FROM users WHERE id = ?",
      [post["user_id"]]
    )
    post["comments"] = db.fetch(
      "SELECT * FROM comments WHERE post_id = ? ORDER BY created_at",
      [post["id"]]
    )
    post["commentCount"] = post["comments"].length
  end

  posts
end

schema.add_query("post", type: "Post",
                 args: { "id" => { type: "Int!" } }) do |root, args, ctx|
  db = Tina4.database
  post = db.fetch_one("SELECT * FROM posts WHERE id = ?", [args["id"].to_i])

  return nil if post.nil?

  post["author"] = db.fetch_one(
    "SELECT id, name, email FROM users WHERE id = ?",
    [post["user_id"]]
  )
  post["comments"] = db.fetch(
    "SELECT * FROM comments WHERE post_id = ? ORDER BY created_at",
    [post["id"]]
  )
  post["commentCount"] = post["comments"].length

  post
end

schema.add_query("user", type: "User",
                 args: { "id" => { type: "Int!" } }) do |root, args, ctx|
  db = Tina4.database
  user = db.fetch_one("SELECT id, name, email FROM users WHERE id = ?", [args["id"].to_i])

  return nil if user.nil?

  user["posts"] = db.fetch(
    "SELECT id, title FROM posts WHERE user_id = ?",
    [user["id"]]
  )
```

```
  user
end
```

Now clients can query across relationships in a single request:

```
curl -X POST http://localhost:7147/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. The fields the client needs. No over-fetching. No under-fetching.

---

7. Auto-Generating Schema from ORM Models

If your ORM models exist, Tina4 can auto-generate GraphQL types and CRUD resolvers from them. Call `from_orm` on the schema:

```
schema = Tina4::GraphQLSchema.new
schema.from_orm(Product)
schema.from_orm(User)

gql = Tina4::GraphQL.new(schema)
gql.register_route
```

With `from_orm`, each model gets:

- A GraphQL type with fields matching the model properties
- A query to list all records (e.g., `products(limit: Int, offset: Int)`)
- A query to get one by ID (e.g., `product(id: ID!)`)
- A mutation to create (e.g., `createProduct(input: ProductInput!)`)
- A mutation to update (e.g., `updateProduct(id: ID!, input: ProductInput!)`)
- A mutation to delete (e.g., `deleteProduct(id: ID!)`)

The `from_orm` method reads field definitions from your model class. It maps Ruby types to GraphQL types:

Ruby field type	GraphQL type
`:integer`, `:int`	`Int`
`:float`, `:double`, `:decimal`	`Float`
`:boolean`, `:bool`	`Boolean`
`:string`, `:text`, `:varchar`	`String`
`:datetime`, `:date`, `:timestamp`	`String`

You can still add custom queries and mutations alongside auto-generated ones. Custom resolvers always take precedence over auto-generated ones.

Example: ORM + Custom Queries

```
schema = Tina4::GraphQLSchema.new

# Auto-generate CRUD for Product
schema.from_orm(Product)

# Add a custom query that filters by category
schema.add_query("productsByCategory", type: "[Product]",
                 args: { "category" => { type: "String!" } }) do |root, args, ctx|
  products, count = Product.where("category = ?", [args["category"]])
  products.map(&:to_hash)
end

gql = Tina4::GraphQL.new(schema)
gql.register_route
```

---

8. Authentication in GraphQL

The resolver block receives a `ctx` hash that includes the current request. Use it to check authentication:

```
schema.add_query("me", type: "User") do |root, args, ctx|
  request = ctx[:request]

  if request.nil? || request.headers["Authorization"].nil?
    raise "Authentication required"
  end

  token = request.headers["Authorization"].sub("Bearer ", "")
  payload = Tina4::Auth.decode(token)

  raise "Invalid token" if payload.nil?

  db = Tina4.database
  db.fetch_one("SELECT id, name, email, role FROM users WHERE id = ?", [payload["user_id"]])
end
```

The `ctx` hash is populated by `register_route`, which passes `{ request: request }` to every resolver.

---

9. The GraphiQL Playground

When you call `register_route`, Tina4 serves a GraphiQL interactive playground at the same endpoint via GET:

```
http://localhost:7147/graphql
```

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

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.

In production, disable the playground by using a custom route instead of `register_route`:

```
Tina4.post "/graphql", auth: false do |request, response|
  result = gql.handle_request(request.body, context: { request: request })
  response.json(result)
end
```

---

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:7147/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. The actual value comes from the `variables` object. This is safer and more efficient than string interpolation.

Variable definitions sit in the operation signature:

```
query GetProduct($id: Int!) {
  product(id: $id) {
    id
    name
    price
  }
}
```

The `$id: Int!` declaration tells GraphQL the variable name, type, and that it is required (the `!`). The executor substitutes `$id` with the value from the `variables` hash before calling the resolver.

You can define multiple variables:

```
curl -X POST http://localhost:7147/graphql \
  -H "Content-Type: application/json" \
  -d '{
    "query": "query Search($category: String!, $limit: Int) { productsByCategory(category: $cate
    "variables": {"category": "Electronics", "limit": 10}
  }'
```

---

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 queries and mutations for `User`, `Post`, and `Comment` with these 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:7147/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:7147/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:7147/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:7147/graphql \
  -H "Content-Type: application/json" \
  -d '{"query": "mutation { addComment(postId: 1, authorName: \"Carol\", body: \"Nice article!\"
```

---

12. Solution

Create `src/routes/graphql_blog.rb`:

```
schema = Tina4::GraphQLSchema.new

# ■■ Queries ■■

schema.add_query("posts", type: "[Post]") do |root, args, ctx|
  db = Tina4.database
  posts = db.fetch("SELECT * FROM posts WHERE published = 1 ORDER BY created_at DESC")

  posts.each do |post|
    post["author"] = db.fetch_one(
      "SELECT id, name, email FROM users WHERE id = ?",
      [post["user_id"]]
    )
    post["comments"] = db.fetch(
      "SELECT * FROM comments WHERE post_id = ? ORDER BY created_at",
      [post["id"]]
    )
    post["commentCount"] = post["comments"].length
  end

  posts
end

schema.add_query("post", type: "Post",
                 args: { "id" => { type: "Int!" } }) do |root, args, ctx|
  db = Tina4.database
  post = db.fetch_one("SELECT * FROM posts WHERE id = ?", [args["id"].to_i])

  return nil if post.nil?

  post["author"] = db.fetch_one(
```

```
    "SELECT id, name, email FROM users WHERE id = ?",
    [post["user_id"]]
  )
  post["comments"] = db.fetch(
    "SELECT * FROM comments WHERE post_id = ? ORDER BY created_at",
    [post["id"]]
  )
  post["commentCount"] = post["comments"].length

  post
end

schema.add_query("user", type: "User",
                 args: { "id" => { type: "Int!" } }) do |root, args, ctx|
  db = Tina4.database
  user = db.fetch_one(
    "SELECT id, name, email FROM users WHERE id = ?",
    [args["id"].to_i]
  )

  return nil if user.nil?

  user["posts"] = db.fetch(
    "SELECT id, title FROM posts WHERE user_id = ?",
    [user["id"]]
  )

  user
end

# ■■ Mutations ■■

schema.add_mutation("createPost", type: "Post",
                    args: {
                      "userId"    => { type: "Int!" },
                      "title"     => { type: "String!" },
                      "body"      => { type: "String!" },
                      "published" => { type: "Boolean" }
                    }) do |root, args, ctx|
  db = Tina4.database

  user = db.fetch_one("SELECT id FROM users WHERE id = ?", [args["userId"].to_i])
  raise "User not found" if user.nil?

  db.execute(
    "INSERT INTO posts (user_id, title, body, published) VALUES (:user_id, :title, :body, :publi
    {
      user_id: args["userId"].to_i,
      title: args["title"],
      body: args["body"],
      published: args["published"] ? 1 : 0
    }
  )

  db.fetch_one("SELECT * FROM posts WHERE id = last_insert_rowid()")
end

schema.add_mutation("addComment", type: "Comment",
                    args: {
```

```
                    "postId"     => { type: "Int!" },
                    "authorName" => { type: "String!" },
                    "body"       => { type: "String!" }
                  }) do |root, args, ctx|
  db = Tina4.database

  post = db.fetch_one("SELECT id FROM posts WHERE id = ?", [args["postId"].to_i])
  raise "Post not found" if post.nil?

  db.execute(
    "INSERT INTO comments (post_id, author_name, body) VALUES (:post_id, :author_name, :body)",
    {
      post_id: args["postId"].to_i,
      author_name: args["authorName"],
      body: args["body"]
    }
  )

  db.fetch_one("SELECT * FROM comments WHERE id = last_insert_rowid()")
end

gql = Tina4::GraphQL.new(schema)
gql.register_route
```

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 every argument against its declared type before your resolver runs. You never need to check whether a required argument is present or whether a string arrived where an integer was expected -- the engine rejects the query before your code executes.

The built-in validator covers:

- **Non-null enforcement** -- Arguments marked with `!` must be present and non-null.

• **Scalar type checking** -- `Int`, `Float`, `String`, `Boolean`, and `ID` values are verified against their declared type.

• **Type coercion** -- When safe, the engine coerces compatible values automatically (e.g., an integer `42` passed to a `Float` field becomes `42.0`, or a string `"7"` passed to an `Int` field becomes `7`).

• **List item validation** -- For `[Int!]!` arguments, the engine checks that the value is a list, that no item is null, and that every item is an integer.

Example: Missing Required Argument

Suppose you define a query that requires an `id`:

```
schema.add_query("user", type: "User",
                 args: { "id" => { type: "ID!" } }) do |root, args, ctx|
  db = Tina4.database
  db.fetch_one("SELECT * FROM users WHERE id = ?", [args["id"].to_i])
end
```

Sending a query without the required argument:

```
{
  user {
    name
    email
  }
}
```

Returns an error before the resolver runs:

```
{
  "errors": [
    {
      "message": "Argument 'id' of type 'ID!' is required but not provided.",
      "locations": [{"line": 2, "column": 3}]
    }
  ]
}
```

Your resolver never executes. No nil-check needed inside the block.

---

14. Field-Level Auth Directives

Tina4 supports three directives that control field-level access in your GraphQL schema:

Directive	Effect
`@auth`	Field resolves only when `ctx["user"]` is present (any authenticated user)
`@role(role: "admin")`	Field resolves only when `ctx["user"]["role"]` matches the specified role

`@guest`	Field resolves only when `ctx["user"]` is absent (unauthenticated visitors)

Passing Auth Context

Pass user information through the context parameter of `execute()`:

```
result = gql.execute(query, variables: {}, context: {
  "user" => { "id" => 1, "role" => "admin" }
})
```

When no user is logged in, pass an empty context or omit the `user` key:

```
result = gql.execute(query, variables: {}, context: {})
```

Schema Example

```
type User {
    id: ID!
    name: String!
    email: String! @auth
    role: String! @role(role: "admin")
}

type Query {
    me: User @auth
    publicStats: Stats @guest
    users: [User!]! @role(role: "admin")
}
```

Behavior on Failed Auth

When a directive check fails, the field resolves to `null` silently -- no error is added to the response. The rest of the query executes normally. This prevents leaking information about which fields exist behind authentication.

For example, if a non-admin user queries:

```
{
  users {
    name
    email
    role
  }
}
```

The response is:

```
{
  "data": {
    "users": null
  }
}
```

No error message. No hint that the field requires admin access. The field is simply excluded.

---

15. Gotchas

1. Resolver Not Called

Problem: A query returns `null` even though data exists.

Cause: The query name in `add_query` does not match the name the client sends. GraphQL is case-sensitive.

Fix: Verify that `schema.add_query("products", ...)` matches `{ products { ... } }` in the client query. Check capitalization.

2. Nested Data Returns Wrong Shape

Problem: `post.author` returns the entire post hash instead of the user.

Cause: You attached the nested data to a key that does not match the field name the client requests.

Fix: Use the exact field name the client will query. If the client sends `{ post { author { name } } }`, the hash must have an `"author"` key:

```
post["author"] = db.fetch_one("SELECT id, name FROM users WHERE id = ?", [post["user_id"]])
```

3. Mutation Returns Null

Problem: A mutation executes but returns null.

Cause: `db.execute` returns affected row count, not the data. Your mutation block ends with `db.execute`, so the block returns an integer.

Fix: Follow every `db.execute` with a `db.fetch_one` to return the actual record:

```
db.execute("INSERT INTO products ...")
db.fetch_one("SELECT * FROM products WHERE id = last_insert_rowid()")
```

4. N+1 Query Problem

Problem: Loading 50 posts with their authors generates 51 database queries (1 for posts + 50 for authors).

Cause: Each nested lookup runs inside a loop. When you resolve `author` for each post, that is one query per post.

Fix: Pre-fetch all related records in the parent resolver. Check `ctx["__selections"]` to see which nested fields the client requested -- if `author` is not in the selection set, skip the join. For ORM queries, use the `include` parameter to eager-load relationships in a single query. For manual queries, load all relevant users in one query, build a hash by ID, and attach them:

```
schema.add_query("posts", type: "[Post]") do |root, args, ctx|
  db = Tina4.database
  posts = db.fetch("SELECT * FROM posts ORDER BY created_at DESC")

  user_ids = posts.map { |p| p["user_id"] }.uniq
  users = db.fetch("SELECT * FROM users WHERE id IN (#{user_ids.join(',')})")
  user_map = users.each_with_object({}) { |u, h| h[u["id"]] = u }
```

```
  posts.each { |p| p["author"] = user_map[p["user_id"]] }
  posts
end
```

5. GraphQL Playground Returns 404

Problem: Visiting `/graphql` in the browser returns a 404.

Cause: You did not call `register_route`, or you registered a POST-only route manually.

Fix: Call `gql.register_route` which registers both POST (for queries) and GET (for the GraphiQL playground).

6. Type Mismatch Between Schema and Data

Problem: A field declared as `Int` receives a string from the database, causing unexpected results.

Cause: SQLite returns all values as strings. Your resolver passes them through without conversion.

Fix: The input validation layer (see section 13) now coerces compatible types automatically -- a string `"42"` passed as an argument to an `Int` parameter is converted before your resolver runs. However, resolver *return values* still need correct types. Cast values explicitly:

```
{
  "id" => record["id"].to_i,
  "price" => record["price"].to_f,
  "inStock" => record["in_stock"] == 1
}
```

The `to_hash` method on ORM models handles this for model properties with type declarations. Raw database hashes need manual casting.

7. Authentication Not Applied

Problem: GraphQL queries work without authentication.

Cause: `register_route` sets `auth: false` by default. No middleware checks the token.

Fix: Check `ctx[:request]` in resolvers that need authentication. Or register a custom POST route with `auth: true` and let your auth middleware run first:

```
Tina4.post "/graphql", auth: true do |request, response|
  result = gql.handle_request(request.body, context: { request: request })
  response.json(result)
end
```

8. Error Messages Expose Internal Details

Problem: Database error messages appear in GraphQL responses.

Cause: Exceptions raised in resolvers become error messages in the response.

Fix: Wrap resolver logic in `begin/rescue` and return clean error messages:

```
schema.add_query("product", type: "Product",
                 args: { "id" => { type: "Int!" } }) do |root, args, ctx|
  begin
```

```
    db = Tina4.database
    db.fetch_one("SELECT * FROM products WHERE id = ?", [args["id"].to_i])
  rescue => e
    raise "Product not found"
  end
end
```

---

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 from Ruby classes and type annotations. No XML authoring required. It uses REXML (part of Ruby's standard library) for XML parsing.

Defining a SOAP Service

Create a class that extends `Tina4::WSDL`. Mark each method with `wsdl_operation` before the `def`. The `output` hash describes the response fields and their types.

```
class Calculator < Tina4::WSDL
  wsdl_operation output: { Result: :int }
  def add(a, b)
    { Result: a.to_i + b.to_i }
  end

  wsdl_operation output: { Result: :int }
  def multiply(a, b)
    { Result: a.to_i * b.to_i }
  end
end

Tina4::Router.get("/calculator") do |request, response|
  service = Calculator.new(request)
  response.call(service.handle)
end

Tina4::Router.post("/calculator") do |request, response|
  service = Calculator.new(request)
  response.call(service.handle)
end
```

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

The `output` hash maps response element names to Ruby type symbols. Input parameter types default to `:string` and are converted based on the output type declarations.

Ruby type	XSD type
`:string`, `String`	`xsd:string`
`:int`, `:integer`, `Integer`	`xsd:int`
`:float`, `:double`, `Float`	`xsd:double`
`:boolean`, `:bool`	`xsd:boolean`
`:date`	`xsd:date`
`:datetime`	`xsd:dateTime`
`:base64`	`xsd:base64Binary`

A More Complete Example

```
class UserService < Tina4::WSDL
  wsdl_operation output: { Name: :string, Email: :string, Active: :boolean }
  def get_user(user_id)
    user = User.find(user_id.to_i)
    if user
      {
        Name: user.name,
        Email: user.email,
        Active: user.active == 1
      }
    else
      { Name: "", Email: "", Active: false }
    end
  end

  wsdl_operation output: { Total: :int, Average: :float }
  def sum_list(numbers)
    nums = numbers.split(",").map(&:to_i)
    {
      Total: nums.sum,
      Average: nums.sum.to_f / nums.length
    }
  end
end

Tina4::Router.get("/api/users/soap") do |request, response|
  service = UserService.new(request)
  response.call(service.handle)
end

Tina4::Router.post("/api/users/soap") do |request, response|
  service = UserService.new(request)
  response.call(service.handle)
end
```

Lifecycle Hooks

Override `on_request` and `on_result` to add validation, logging, or transformation around every operation call.

```
class AuditedService < Tina4::WSDL
  def on_request(request)
    puts "SOAP request from #{request.headers['x-forwarded-for'] || 'unknown'}"
  end

  def on_result(result)
    result[:Timestamp] = Time.now.iso8601
    result
  end

  wsdl_operation output: { Status: :string, Timestamp: :string }
  def ping
    { Status: "ok" }
  end
end
```

Testing with curl

Fetch the WSDL definition:

```
curl http://localhost:7147/calculator?wsdl
```

Call the add operation with a SOAP request:

```
curl -X POST http://localhost:7147/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. 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 should see it -- no page refresh, no polling, no waiting. But HTTP is request-response. The client asks. The server answers. The server cannot speak first.

WebSocket tears down that wall. It establishes a persistent, bi-directional connection between browser and server. Either side can send messages at any time. The connection holds until one side closes it.

Tina4 treats WebSocket as routing. Define a WebSocket handler the same way you define an HTTP route. The path determines which handler owns the connection.

---

2. What WebSocket Is

HTTP works like this:

```
Client: "Give me /api/products"
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)
```

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.
- **Real-time**: Messages arrive within milliseconds.

---

3. Router.websocket -- WebSocket as a Route

In Tina4, you define WebSocket handlers using `Tina4::Router.websocket`:

```
Tina4::Router.websocket "/ws/echo" do |connection, event, data|
  if event == :message
    connection.send("Echo: #{data}")
  end
end
```

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, broadcast, or close the connection.
- **event**: The event type as a symbol: `:open`, `:message`, or `:close`.
- **data**: The message data (only present for `:message` events).

Shorthand Syntax

The same handler can be written with the shorthand `Tina4.websocket`:

```
Tina4.websocket "/ws/echo" do |connection, event, data|
  if event == :message
    connection.send("Echo: #{data}")
  end
end
```

Both forms are identical. Use whichever reads better in your project.

Starting the Server

WebSocket runs alongside your HTTP server. It works with both Puma and WEBrick:

```
tina4 serve

  Tina4 Ruby v3.0.0
  HTTP server running at http://0.0.0.0:7147
  WebSocket server running at ws://0.0.0.0:7147
  Press Ctrl+C to stop
```

---

4. Connection Events

There are three events, passed as symbols:

Event	When it fires	argument
`:open`	A client connects	`nil`
`:message`	A client sends a message	The message string
`:close`	A client disconnects	`nil`

A Complete Handler

```
Tina4::Router.websocket "/ws/chat" do |connection, event, data|
  case event
  when :open
    $stderr.puts "[Chat] New connection: #{connection.id}"
    connection.send(JSON.generate({
      type: "system",
      message: "Welcome to the chat!",
      your_id: connection.id
    }))

  when :message
    message = JSON.parse(data)
    $stderr.puts "[Chat] #{connection.id}: #{message['text'] || data}"

    connection.send(JSON.generate({
      type: "message",
      from: connection.id,
      text: message["text"] || data,
      timestamp: Time.now.iso8601
    }))

  when :close
    $stderr.puts "[Chat] Disconnected: #{connection.id}"
  end
end
```

---

5. Connection Methods

The `connection` object provides three methods for communication:

Method	What it does
`connection.send(message)`	Send a message to this client only
`connection.broadcast(message)`	Send a message to all clients on the same path
`connection.close()`	Close this client's connection

Sending to a Single Client

`connection.send` sends a message to the specific client that triggered the event:

```
Tina4::Router.websocket "/ws/private" do |connection, event, data|
  if event == :message
    message = JSON.parse(data)
    action = message["action"] || ""

    if action == "get-time"
      connection.send(JSON.generate({
        type: "time",
        server_time: Time.now.iso8601
      }))
```

```
    end

    if action == "get-status"
      connection.send(JSON.generate({
        type: "status",
        uptime: 3600,
        connections: 42,
        memory_mb: (ObjectSpace.memsize_of_all / 1024.0 / 1024.0).round(2)
      }))
    end
  end
end
```

Closing a Connection

`connection.close` terminates the client's connection from the server side. This triggers the `:close` event:

```
Tina4::Router.websocket "/ws/secure" do |connection, event, data|
  case event
  when :open
    token = connection.params["token"]
    unless valid_token?(token)
      connection.send(JSON.generate({ error: "Invalid token" }))
      connection.close
    end
  when :message
    connection.broadcast(data)
  when :close
    $stderr.puts "Client disconnected"
  end
end
```

---

6. Broadcasting to All Clients

`connection.broadcast` sends a message to every client connected to the same WebSocket path. Broadcast is **path-scoped** -- clients on `/ws/chat/room-1` never receive broadcasts from `/ws/chat/room-2`:

```
Tina4::Router.websocket "/ws/announcements" do |connection, event, data|
  case event
  when :open
    connection.broadcast(JSON.generate({
      type: "system",
      message: "A new user joined",
      online_count: connection.connection_count
    }))

  when :message
    message = JSON.parse(data)

    connection.broadcast(JSON.generate({
      type: "announcement",
      from: connection.id,
      text: message["text"] || "",
      timestamp: Time.now.iso8601
```

```
    }))

  when :close
    connection.broadcast(JSON.generate({
      type: "system",
      message: "A user left",
      online_count: connection.connection_count
    }))
  end
end
```

Broadcast Excluding Sender

```
when :message
  message = JSON.parse(data)

  # Send to sender (confirmation)
  connection.send(JSON.generate({ type: "sent", text: message["text"] }))

  # Send to everyone else
  connection.broadcast(JSON.generate({
    type: "message",
    from: message["username"] || "Anonymous",
    text: message["text"],
    timestamp: Time.now.iso8601
  }), true)  # true = exclude sender
```

---

7. Path Parameters and Scoped Isolation

WebSocket paths support the same `{param}` syntax as HTTP routes. Access them with `connection.params["param_name"]`. Different resolved paths are completely isolated:

```
Tina4::Router.websocket "/ws/chat/{room}" do |connection, event, data|
  room = connection.params["room"]

  case event
  when :open
    $stderr.puts "[Room #{room}] New connection: #{connection.id}"
    connection.broadcast(JSON.generate({
      type: "system",
      message: "Someone joined room #{room}",
      room: room,
      online: connection.connection_count
    }))

  when :message
    message = JSON.parse(data)
    connection.broadcast(JSON.generate({
      type: "message",
      room: room,
      from: message["username"] || "Anonymous",
      text: message["text"] || "",
      timestamp: Time.now.iso8601
    }))

  when :close
    connection.broadcast(JSON.generate({
```

```
      type: "system",
      message: "Someone left room #{room}",
      room: room,
      online: connection.connection_count
    }))
  end
end
```

A client connecting to `/ws/chat/ruby` only sees messages broadcast on `/ws/chat/ruby`. A client on `/ws/chat/python` is in a completely separate space.

---

8. Building a Live Chat

WebSocket Handler

Create `src/routes/chat_ws.rb`:

```
$chat_users = {}

Tina4::Router.websocket "/ws/livechat/{room}" do |connection, event, data|
  room = connection.params["room"]

  case event
  when :open
    $chat_users[connection.id] = {
      id: connection.id,
      username: "Anonymous",
      room: room,
      joined_at: Time.now.iso8601
    }

    connection.send(JSON.generate({
      type: "welcome",
      message: "Connected to room: #{room}",
      your_id: connection.id,
      online: connection.connection_count
    }))

  when :message
    message = JSON.parse(data)
    type = message["type"] || "message"

    if type == "set-username"
      old_name = $chat_users[connection.id][:username]
      $chat_users[connection.id][:username] = message["username"]

      connection.broadcast(JSON.generate({
        type: "system",
        message: "#{old_name} is now known as #{message['username']}"
      }))
    end

    if type == "message"
      username = $chat_users[connection.id][:username]

      connection.broadcast(JSON.generate({
```

```
        type: "message",
        from: username,
        from_id: connection.id,
        text: message["text"] || "",
        timestamp: Time.now.iso8601
      }))
    end

    if type == "typing"
      username = $chat_users[connection.id][:username]

      connection.broadcast(JSON.generate({
        type: "typing",
        username: username
      }), true)  # Exclude sender
    end

  when :close
    username = $chat_users.dig(connection.id, :username) || "Unknown"
    $chat_users.delete(connection.id)

    connection.broadcast(JSON.generate({
      type: "system",
      message: "#{username} left the chat",
      online: connection.connection_count
    }))
  end
end
```

---

9. Live Notifications

WebSocket is great for pushing notifications to users in real time:

```
Tina4::Router.websocket "/ws/notifications/{user_id}" do |connection, event, data|
  user_id = connection.params["user_id"]

  case event
  when :open
    $stderr.puts "[Notifications] User #{user_id} connected"
    connection.send(JSON.generate({
      type: "connected",
      message: "Listening for notifications"
    }))
  when :message
    # Handle client acknowledgments if needed
  when :close
    $stderr.puts "[Notifications] User #{user_id} disconnected"
  end
end

# HTTP endpoint that triggers a notification
Tina4::Router.post("/api/orders/{order_id:int}/ship") do |request, response|
  order_id = request.params["order_id"]
  user_id = request.body["user_id"] || 0

  Tina4::Router.push_to_websocket("/ws/notifications/#{user_id}", JSON.generate({
    type: "notification",
```

```
    title: "Order Shipped",
    message: "Your order ##{order_id} has been shipped!",
    action_url: "/orders/#{order_id}",
    timestamp: Time.now.iso8601
  }))

  response.json({ message: "Order shipped, user notified" })
end
```

---

10. Connecting from JavaScript

Using Plain WebSocket

The browser's built-in `WebSocket` API is all you need:

```
<script>
    const ws = new WebSocket("ws://" + location.host + "/ws/chat/general");

    ws.onopen = function () {
        console.log("Connected to chat");
    };

    ws.onmessage = function (event) {
        const message = JSON.parse(event.data);
        console.log("Received:", message);
    };

    ws.onclose = function () {
        console.log("Disconnected from chat");
    };

    function sendMessage(text) {
        ws.send(JSON.stringify({ type: "message", text: text }));
    }
</script>
```

Using Frond.js

Tina4's Frond helper wraps WebSocket with convenience features:

```
<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);
    });

    ws.on("close", function () {
        console.log("Disconnected from chat");
    });
```

```
    function sendMessage(text) {
        ws.send(JSON.stringify({ type: "message", text: text }));
    }
</script>
```

Auto-Reconnect

```
const ws = frond.ws("/ws/notifications/42", {
    reconnect: true,
    reconnectInterval: 3000,
    maxReconnectAttempts: 10
});
```

---

11. A Complete Chat Page

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

    <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:
    </form>

    <script>
        const room = "{{ room }}";
        const ws = new WebSocket("ws://" + location.host + "/ws/livechat/" + room);
        const messagesDiv = document.getElementById("messages");
        let username = prompt("Enter your username:") || "Anonymous";

        ws.onopen = function () {
            document.getElementById("status").textContent = "Connected";
            ws.send(JSON.stringify({ type: "set-username", username: username }));
        };

        ws.onmessage = function (event) {
            const msg = JSON.parse(event.data);
            const div = document.createElement("div");
            div.style.marginBottom = "8px";

            if (msg.type === "message") {
                div.innerHTML = "<strong>" + msg.from + ":</strong> " + msg.text;
            } else if (msg.type === "system") {
                div.style.cssText = "color: #888; font-style: italic;";
                div.textContent = msg.message;
            }
```

```
            messagesDiv.appendChild(div);
            messagesDiv.scrollTop = messagesDiv.scrollHeight;

            if (msg.online !== undefined) {
                document.getElementById("online").textContent = "Online: " + msg.online;
            }
        };

        ws.onclose = function () {
            document.getElementById("status").textContent = "Disconnected";
        };

        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 = "";
            }
        });
    </script>
{% endblock %}
```

Create the route:

```
Tina4::Router.get("/chat/{room}") do |request, response|
  room = request.params["room"]
  response.render("chat.html", { room: room })
end
```

---

12. Server Compatibility

WebSocket works with both Puma and WEBrick. Tina4 detects the available server and sets up WebSocket handling automatically:

Server	Notes
Puma	Recommended for production. Handles concurrent connections efficiently.
WEBrick	Ships with Ruby. Good for development and testing.

For production deployments behind Nginx, configure WebSocket proxying:

```
location /ws/ {
    proxy_pass http://127.0.0.1:7147;
    proxy_http_version 1.1;
    proxy_set_header Upgrade $http_upgrade;
    proxy_set_header Connection "upgrade";
}
```

---

13. Exercise: Build a Real-Time Chat Room

Build a WebSocket chat room with username support, message broadcasting, and join/leave notifications.

Requirements

- WebSocket endpoint at `/ws/room/{room_name}` that handles `:open`, `:message`, and `:close` events
- HTTP endpoint at `GET /room/{room_name}` that serves an HTML chat page
- The chat page should prompt for a username, display messages in real time, and show online count

Test by:

- Open `http://localhost:7147/room/test` in two browser tabs
- Set different usernames
- Send messages from both tabs and verify they appear in both
- Close one tab and verify the "user left" message appears

---

14. Solution

Create `src/routes/chat_room.rb`:

```
$room_users = {}

Tina4::Router.websocket "/ws/room/{room_name}" do |connection, event, data|
  room = connection.params["room_name"]
  key = "#{room}:#{connection.id}"

  case event
  when :open
    $room_users[key] = "Anonymous"

    connection.send(JSON.generate({
      type: "system",
      message: "Welcome to room: #{room}",
      online: connection.connection_count
    }))

    connection.broadcast(JSON.generate({
      type: "system",
      message: "A new user joined",
      online: connection.connection_count
    }), true)

  when :message
    msg = JSON.parse(data)
    type = msg["type"] || "chat"

    if type == "set-name"
      old_name = $room_users[key] || "Anonymous"
```

```
      new_name = msg["name"] || "Anonymous"
      $room_users[key] = new_name

      connection.broadcast(JSON.generate({
        type: "system",
        message: "#{old_name} changed their name to #{new_name}"
      }))
    end

    if type == "chat"
      username = $room_users[key] || "Anonymous"

      connection.broadcast(JSON.generate({
        type: "chat",
        from: username,
        text: msg["text"] || "",
        timestamp: Time.now.strftime("%H:%M:%S")
      }))
    end

  when :close
    username = $room_users[key] || "Anonymous"
    $room_users.delete(key)

    connection.broadcast(JSON.generate({
      type: "system",
      message: "#{username} left the room",
      online: connection.connection_count
    }))
  end
end

Tina4::Router.get("/room/{room_name}") do |request, response|
  room = request.params["room_name"]
  response.render("room.html", { room: room })
end
```

---

15. 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 gem as an optional dependency:

```
gem install redis
```

If `TINA4_WS_BACKPLANE` is not set (the default), Tina4 broadcasts only to local connections. This is fine for single-instance deployments.

---

16. Gotchas

1. WebSocket Needs a Persistent Server

Problem: WebSocket connections drop immediately.

Fix: Use `tina4 serve` which runs a persistent server. For production with Puma, configure WebSocket proxying with Nginx.

2. Events Are Symbols, Not Strings

Problem: Your handler never matches any events.

Fix: Use symbols (`:open`, `:message`, `:close`), not strings (`"open"`, `"message"`, `"close"`).

3. Messages Are Strings, Not Objects

Problem: `data` in the message handler is a string, not a Ruby hash.

Fix: Always `JSON.parse(data)` when you expect JSON messages. Always `JSON.generate(...)` when you send structured data.

4. 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` and `/ws/chat/room-2` are counted separately.

5. Broadcasting Does Not Scale Across Servers

Problem: Users connected to different server instances do not see each other's messages.

Fix: Use a pub/sub backend like Redis to relay messages across server instances.

6. Large Messages Cause Disconnects

Problem: The connection drops when sending a large message.

Fix: Keep messages small (under 64KB). Use HTTP endpoints for bulk data transfer.

7. Memory Leak from Tracking Connected Users

Problem: The server's memory usage grows over time.

Fix: Always clean up in the `:close` handler: `$chat_users.delete(connection.id)`.

8. No Authentication on WebSocket Connect

Problem: Anyone can connect to your WebSocket endpoint.

Fix: Pass the token as a query parameter: `ws://localhost:7147/ws/chat?token=eyJ...`. Validate in the `:open` handler and call `connection.close` if invalid.

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:7147/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:7147;
    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:7147/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:7147/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. When the World Still Uses SOAP

REST gets all the attention. But banks, government portals, insurance systems, and ERP platforms built in the 2000s still speak SOAP. You need to call a mortgage calculation service. Your payment processor publishes a WSDL. Your government tax API requires SOAP envelopes.

Tina4's WSDL module lets you expose your own operations as a SOAP service and call remote WSDL services from Ruby.

---

2. What SOAP and WSDL Are

SOAP (Simple Object Access Protocol) sends XML envelopes over HTTP. A request wraps parameters in XML. The response wraps results in XML.

WSDL (Web Services Description Language) is the machine-readable contract that describes what operations a SOAP service exposes, what parameters each operation accepts, and what it returns. Clients parse the WSDL to know how to call the service.

A WSDL service call looks like this at the wire level:

```
POST /soap/calculator HTTP/1.1
Content-Type: text/xml

<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <Add>
      <a>14</a>
      <b>28</b>
    </Add>
  </soap:Body>
</soap:Envelope>
```

Response:

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <AddResponse>
      <result>42</result>
    </AddResponse>
  </soap:Body>
</soap:Envelope>
```

Tina4 handles all of this XML generation and parsing for you.

---

3. Defining a WSDL Service

Create a WSDL service class that inherits from `Tina4::WSDL` and defines operations.

```
# src/services/calculator_service.rb
```

```
class CalculatorService < Tina4::WSDL
  service_name "CalculatorService"
  namespace    "http://example.com/calculator"
  description  "Basic arithmetic operations"

  wsdl_operation :add,
    params: { a: :integer, b: :integer },
    returns: :integer do |params|
    params[:a] + params[:b]
  end

  wsdl_operation :subtract,
    params: { a: :integer, b: :integer },
    returns: :integer do |params|
    params[:a] - params[:b]
  end

  wsdl_operation :multiply,
    params: { a: :float, b: :float },
    returns: :float do |params|
    params[:a] * params[:b]
  end

  wsdl_operation :divide,
    params: { a: :float, b: :float },
    returns: :float do |params|
    raise "Division by zero" if params[:b] == 0
    params[:a] / params[:b]
  end
end
```

---

4. Mounting the Service

Mount the WSDL service at a path using the router.

```
# src/routes/soap.rb
Tina4::Router.mount_wsdl("/soap/calculator", CalculatorService)
```

This creates two endpoints:

- `GET /soap/calculator?wsdl` -- serves the generated WSDL XML document
- `POST /soap/calculator` -- handles incoming SOAP requests

---

5. Auto-Generated WSDL

Visit `/soap/calculator?wsdl` in a browser or curl:

```
curl "http://localhost:7147/soap/calculator?wsdl"
```

Tina4 generates the full WSDL document from your operation definitions:

```
<?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">

  <message name="addRequest">
    <part name="a" type="xsd:integer"/>
    <part name="b" type="xsd:integer"/>
  </message>
  <message name="addResponse">
    <part name="result" type="xsd:integer"/>
  </message>

  <portType name="CalculatorServicePortType">
    <operation name="add">
      <input message="tns:addRequest"/>
      <output message="tns:addResponse"/>
    </operation>
    <!-- ... -->
  </portType>
  <!-- binding and service elements follow -->
</definitions>
```

The WSDL is generated dynamically. It always reflects the current operation definitions.

---

6. Calling the SOAP Service

```
curl -X POST http://localhost:7147/soap/calculator \
  -H "Content-Type: text/xml" \
  -d '
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <add>
      <a>14</a>
      <b>28</b>
    </add>
  </soap:Body>
</soap:Envelope>'
```

Response:

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <addResponse>
      <result>42</result>
    </addResponse>
  </soap:Body>
</soap:Envelope>
```

---

7. Calling a Remote WSDL Service

Use `Tina4::WSDL::Client` to call an external SOAP service.

```
client = Tina4::WSDL::Client.new("https://www.w3schools.com/xml/tempconvert.asmx?wsdl")
```

```
result = client.call(:CelsiusToFahrenheit, Celsius: "100")

puts result[:CelsiusToFahrenheitResult]
# => 212
```

The client fetches and parses the WSDL on first call, then caches it.

With Authentication

```
client = Tina4::WSDL::Client.new(
  "https://secure-service.example.com/api?wsdl",
  headers: {
    "Authorization" => "Basic #{Base64.strict_encode64('user:pass')}"
  }
)
```

---

8. Complex Types

Operations can use nested types for richer request and response shapes.

```
class OrderService < Tina4::WSDL
  service_name "OrderService"
  namespace    "http://example.com/orders"

  wsdl_type :Address do
    field :street, :string
    field :city,   :string
    field :zip,    :string
    field :country, :string
  end

  wsdl_type :OrderResult do
    field :order_id,    :string
    field :status,      :string
    field :total,       :float
  end

  wsdl_operation :place_order,
    params: { customer_email: :string, shipping_address: :Address, total: :float },
    returns: :OrderResult do |params|
    {
      order_id: SecureRandom.uuid,
      status:   "pending",
      total:    params[:total]
    }
  end
end
```

---

9. Error Handling in Operations

Raise a standard Ruby exception to return a SOAP fault.

```
wsdl_operation :divide,
  params: { a: :float, b: :float },
  returns: :float do |params|
```

```
    raise ArgumentError, "Divisor cannot be zero" if params[:b] == 0
    params[:a] / params[:b]
  end
```

The caller receives:

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <soap:Fault>
      <faultcode>soap:Server</faultcode>
      <faultstring>Divisor cannot be zero</faultstring>
    </soap:Fault>
  </soap:Body>
</soap:Envelope>
```

---

10. Gotchas

1. WSDL caching in production

The remote WSDL client caches the WSDL after the first fetch. If the remote service updates its WSDL, restart your app or call `client.reload_wsdl` to force a refresh.

2. Character encoding

SOAP envelopes must be UTF-8. If your operation returns strings with special characters, ensure they are UTF-8 encoded before returning.

3. Namespace conflicts

Each WSDL service must have a unique namespace. Duplicate namespaces across mounted services cause client-side parsing errors.

4. SOAP is not REST

SOAP uses `POST` for every operation, including reads. Do not expect `GET` requests to trigger SOAP operations -- only the `?wsdl` query param uses `GET`.

Dependency Injection Container

1. The Problem With Hard Dependencies

Your route handler creates a `Tina4::Database` instance directly. Your mailer instantiates its SMTP connection inline. Your PDF generator calls `Tina4::Api.new(...)` inside the method body.

Hard dependencies make code hard to test (you cannot swap the real database for a fake one), hard to reconfigure (the connection string is buried in 12 files), and hard to share (every caller creates its own instance instead of sharing one).

A DI container is a registry. You register services by name once. Everything that needs a service asks the container for it by name. Tests register fake implementations. Production registers real ones.

---

2. Registering a Service

```
Tina4::Container.register(:database) do
  Tina4::Database.new(ENV["TINA4_DATABASE_URL"])
end
```

The block is a factory. It runs the first time the service is resolved (lazy initialization). The result is cached and reused for every subsequent resolve.

Registering Multiple Services

```
Tina4::Container.register(:database) do
  Tina4::Database.new(ENV["TINA4_DATABASE_URL"])
end

Tina4::Container.register(:cache) do
  Tina4::Cache.new(backend: :memory)
end

Tina4::Container.register(:stripe) do
  Tina4::Api.new("https://api.stripe.com", {
    "Authorization"  => "Bearer #{ENV['STRIPE_SECRET_KEY']}",
    "Content-Type"   => "application/json",
    "Stripe-Version" => "2024-11-20"
  })
end

Tina4::Container.register(:mailer) do
  Tina4::Mailer.new(
    host:     ENV["TINA4_MAIL_HOST"],
    port:     ENV["TINA4_MAIL_PORT"].to_i,
    username: ENV["SMTP_USER"],
    password: ENV["SMTP_PASS"]
  )
end
```

---

3. Resolving a Service

```
db = Tina4::Container.resolve(:database)
users = db.query("SELECT * FROM users")
```

The first `resolve` call runs the factory block and caches the result. Every subsequent call returns the same instance.

---

4. Checking Registration

```
if Tina4::Container.registered?(:cache)
  cache = Tina4::Container.resolve(:cache)
  cached = cache.get("products:list")
end
```

---

5. Using Services in Route Handlers

```
# @noauth
Tina4::Router.get("/api/users") do |request, response|
  db = Tina4::Container.resolve(:database)

  users = db.query("SELECT id, name, email FROM users")

  response.json({ users: users, count: users.length })
end

Tina4::Router.post("/api/orders") do |request, response|
  body   = request.body
  db     = Tina4::Container.resolve(:database)
  mailer = Tina4::Container.resolve(:mailer)
  stripe = Tina4::Container.resolve(:stripe)

  result = stripe.post("/v1/payment_intents", body: {
    amount:   (body["total"].to_f * 100).to_i,
    currency: "usd"
  })

  unless result.success?
    next response.json({ error: "Payment failed" }, 402)
  end

  order_id = db.execute(
    "INSERT INTO orders (email, total, payment_intent) VALUES (?, ?, ?)",
    body["email"], body["total"], result.body["id"]
  )

  mailer.send(
    to:      body["email"],
    subject: "Order Confirmation",
    body:    "Your order ##{order_id} has been placed."
  )

  response.json({ order_id: order_id }, 201)
end
```

---

6. clear!: Reset the Container

Remove all registered services and their cached instances.

```
Tina4::Container.clear!
```

Use `clear!` in tests to start fresh between test cases and register test doubles.

---

7. Testing With the Container

Swap real services for fakes in tests without changing production code.

```
# test/test_orders.rb
require "minitest/autorun"

class FakeMailer
  attr_reader :sent

  def initialize
    @sent = []
  end

  def send(opts)
    @sent << opts
  end
end

class FakeStripe
  def post(path, body:)
    OpenStruct.new(
      success?: true,
      body: { "id" => "pi_test_123" }
    )
  end
end

class FakeDatabase
  def execute(sql, *args)
    42  # Simulated insert returning an ID
  end
end

class OrderTest < Minitest::Test
  def setup
    Tina4::Container.clear!
    @mailer = FakeMailer.new

    Tina4::Container.register(:database) { FakeDatabase.new }
    Tina4::Container.register(:mailer)   { @mailer }
    Tina4::Container.register(:stripe)   { FakeStripe.new }
  end

  def teardown
    Tina4::Container.clear!
```

```
  end

  def test_order_sends_confirmation_email
    # Simulate a POST /api/orders request
    # ...route invocation logic...

    assert_equal 1, @mailer.sent.length
    assert_equal "Order Confirmation", @mailer.sent.first[:subject]
  end
end
```

No real database, no real Stripe, no real email -- the test runs in milliseconds and never touches external services.

---

8. Service Dependencies

A factory block can resolve other services from the container.

```
Tina4::Container.register(:order_service) do
  db     = Tina4::Container.resolve(:database)
  mailer = Tina4::Container.resolve(:mailer)
  stripe = Tina4::Container.resolve(:stripe)

  OrderService.new(db: db, mailer: mailer, stripe: stripe)
end
```

```
class OrderService
  def initialize(db:, mailer:, stripe:)
    @db     = db
    @mailer = mailer
    @stripe = stripe
  end

  def place(email:, total:)
    result = @stripe.post("/v1/payment_intents", body: {
      amount: (total * 100).to_i, currency: "usd"
    })

    raise "Payment failed" unless result.success?

    order_id = @db.execute(
      "INSERT INTO orders (email, total) VALUES (?, ?)", email, total
    )

    @mailer.send(to: email, subject: "Order ##{order_id} confirmed")

    order_id
  end
end
```

```
Tina4::Router.post("/api/orders") do |request, response|
  body    = request.body
  service = Tina4::Container.resolve(:order_service)

  order_id = service.place(email: body["email"], total: body["total"].to_f)
  response.json({ order_id: order_id }, 201)
rescue => e
```

```
  response.json({ error: e.message }, 500)
end
```

The route handler has one job: parse the request, delegate to the service, return the response.

---

9. Gotchas

1. clear! drops cached instances

`clear!` removes the factory registration and the cached instance. Any subsequent `resolve` after `clear!` raises unless you re-register.

2. Singletons by default

The container caches the first resolved instance. If your service is not thread-safe, wrapping it in a mutex or using a new instance per request is safer.

3. Circular dependencies cause infinite loops

If service A resolves B, and B resolves A during factory execution, the container loops forever. Restructure to break the cycle.

4. Registration order matters

A factory block that resolves `:database` runs when it is first resolved, not when it is registered. Registration order does not matter. Resolution order does.

Service Runner

1. Work That Runs Forever

Some work does not fit inside an HTTP request. A heartbeat that pings a health check endpoint every 30 seconds. A queue consumer that processes jobs continuously.

These are background services: long-running processes that start with the app and run until it stops.

Tina4's Service Runner manages background services. Register a service with a name and a block. The runner starts it in a background thread. The service runs for the lifetime of the app. If it crashes, the runner restarts it.

---

2. Defining a Service

```
Tina4::ServiceRunner.register("heartbeat") do
  loop do
    Tina4::Log.info("Heartbeat", status: "alive", time: Time.now.utc.iso8601)
    sleep 30
  end
end
```

The block runs in a background thread. The `loop` keeps it alive. `sleep 30` yields control between iterations.

---

3. Starting All Services

```
Tina4::ServiceRunner.start_all
```

Call this once, typically in your app startup file after route definitions. All registered services start in background threads.

```
# config/app.rb

require_relative "../src/routes/api"
require_relative "../src/services/background"

Tina4::ServiceRunner.start_all
```

---

4. Starting a Single Service

```
Tina4::ServiceRunner.start("heartbeat")
```

Useful for starting services conditionally based on environment or configuration.

---

5. Stopping Services

```
# Stop a specific service
Tina4::ServiceRunner.stop("heartbeat")

# Stop all running services
Tina4::ServiceRunner.stop_all
```

`stop` signals the thread to terminate. The runner waits for the thread to finish its current iteration before stopping.

---

6. Checking Service Status

```
Tina4::ServiceRunner.running?("heartbeat")
# => true or false

Tina4::ServiceRunner.status
# => { "heartbeat" => :running, "metrics" => :stopped, "queue_consumer" => :running }
```

---

7. Automatic Restart on Failure

If a service crashes, the runner restarts it with an exponential backoff delay.

```
Tina4::ServiceRunner.register("unreliable_service") do
  loop do
    begin
      do_something_that_might_fail
      sleep 10
    rescue => e
      Tina4::Log.error("Service error", service: "unreliable_service", error: e.message)
      # re-raise to trigger restart, or rescue and continue
      raise
    end
  end
end
```

If the block raises, the runner logs the error, waits (1s, 2s, 4s... up to 60s), then restarts the service.

Set `max_restarts` to cap automatic restarts:

```
Tina4::ServiceRunner.register("payment_sync", max_restarts: 5) do
  # ...
end
```

After 5 restarts, the service is marked as `:failed` and not restarted again.

---

8. Queue Consumer Service

The most common background service: processing a queue continuously.

```
Tina4::ServiceRunner.register("email_consumer") do
  queue = Tina4::Queue.new(topic: "emails")

  loop do
    job = queue.pop

    if job.nil?
      sleep 2  # No jobs -- wait before polling again
      next
    end

    begin
      payload = job.payload
      send_email(payload[:to], payload[:subject], payload[:body])
      job.complete

      Tina4::Log.info("Email sent", to: payload[:to], subject: payload[:subject])
    rescue => e
      job.fail(e.message)
      Tina4::Log.error("Email failed", to: payload[:to], error: e.message)
    end
  end
end
```

---

9. Scheduled Task Service

Run a task on a schedule using a service with sleep-based timing.

```
Tina4::ServiceRunner.register("daily_report") do
  loop do
    now = Time.now

    # Run at 08:00 every day
    if now.hour == 8 && now.min == 0
      Tina4::Log.info("Generating daily report")

      db = Tina4::Container.resolve(:database)
      rows = db.query("SELECT COUNT(*) as orders FROM orders WHERE DATE(created_at) = DATE('now'

      Tina4::Log.info("Daily report complete", orders_today: rows.first["orders"])

      sleep 61  # Skip past the current minute to avoid re-running
    else
      sleep 30  # Check every 30 seconds
    end
  end
end
```

---

10. Metrics Collector Service

```
Tina4::ServiceRunner.register("metrics_collector") do
  loop do
    mem_mb = `ps -o rss= -p #{Process.pid}`.strip.to_i / 1024

    Tina4::Log.info("Metrics",
      memory_mb:  mem_mb,
      pid:        Process.pid,
      threads:    Thread.list.count
    )

    sleep 60
  end
end
```

---

11. Listing Registered Services

```
Tina4::ServiceRunner.services.each do |name, info|
  puts "#{name}: #{info[:status]}"
end
```

Output:

```
heartbeat: running
email_consumer: running
daily_report: running
metrics_collector: running
```

---

12. Shutdown Hooks

Register cleanup logic that runs when a service is stopped.

```
Tina4::ServiceRunner.register("database_sync") do |service|
  db = Tina4::Container.resolve(:database)

  service.on_stop do
    Tina4::Log.info("Database sync shutting down -- flushing pending writes")
    db.flush
  end

  loop do
    sync_records(db)
    sleep 5
  end
end
```

---

13. Gotchas

1. Services share the main process

Background services run in threads, not separate processes. A runaway service that consumes 100% CPU affects the HTTP server running in the same process. Keep service work lightweight or move heavy computation to separate processes via the queue system.

2. sleep is required in loops

A `loop` with no `sleep` spins the CPU at 100%. Always `sleep` between iterations.

3. Thread safety

Services share memory with the main thread. Access shared mutable state (global variables, class variables, shared caches) through a mutex.

```
COUNTER_MUTEX = Mutex.new
COUNTER = { value: 0 }

Tina4::ServiceRunner.register("counter_service") do
  loop do
    COUNTER_MUTEX.synchronize { COUNTER[:value] += 1 }
    sleep 1
  end
end
```

4. stop_all on shutdown

In production, call `Tina4::ServiceRunner.stop_all` in a signal trap to cleanly shut down services before the process exits.

```
trap("SIGTERM") do
  Tina4::ServiceRunner.stop_all
  exit 0
end
```

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:7147/__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:7147/__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:7147/__dev/mcp/sse` -- returns the message endpoint URL
- **Message endpoint**: `POST http://localhost:7147/__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 the framework's `Tina4::Mcp::Tools` registration block -- that 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, Ruby 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 dependencies (from `Gemfile` / `*.gemspec`)
`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:7147/__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 `Tina4::ORM.bind(db)` in `app.rb`
- 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

Require `McpServer` and create an instance on any path:

```
require "tina4"

mcp = Tina4::McpServer.new("/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.rb` before `run`:

```
mcp.register_routes
```

---

3. Registering Tools with mcp_tool

The `Tina4.mcp_tool` method turns a block into an MCP tool. Parameter metadata becomes the input schema:

```
require "tina4"

mcp = Tina4::McpServer.new("/crm/mcp", name: "CRM Tools")

Tina4.mcp_tool("lookup_customer", description: "Find a customer by email", server: mcp,
  params: [{ name: "email", type: "string" }]) do |args|
  db.fetch_one("SELECT * FROM customers WHERE email = ?", [args["email"]])
end

Tina4.mcp_tool("recent_orders", description: "Get recent orders for a customer", server: mcp,
  params: [
    { name: "customer_id", type: "integer" },
    { name: "limit", type: "integer", default: 10 }
  ]) do |args|
  db.fetch(
    "SELECT * FROM orders WHERE customer_id = ? ORDER BY created_at DESC",
    [args["customer_id"]], limit: args["limit"] || 10
```

```
  ).to_a
end
```

The registration extracts:

- **Parameter names** from the params list
- **Types** from the type field (`"string"`, `"integer"`, `"boolean"`, etc.)
- **Required vs optional** -- parameters with defaults are optional
- **Description** from the `description` argument

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:

```
Tina4.mcp_resource("crm://product-catalog", description: "All active products", server: mcp) do
  db.fetch("SELECT id, name, price, category FROM products WHERE active = 1").to_a
end

Tina4.mcp_resource("crm://tax-rates", description: "Current tax rates by region", server: mcp) d
  db.fetch("SELECT region, rate FROM tax_rates").to_a
end
```

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. Register each method as a tool:

```
require "tina4"

mcp = Tina4::McpServer.new("/accounting/mcp", name: "Accounting Tools")

class AccountingService
  def initialize(db)
    @db = db
  end

  def lookup(invoice_no)
```

```
    @db.fetch_one("SELECT * FROM invoices WHERE invoice_no = ?", [invoice_no])
  end

  def balances(min_amount = 0.0)
    @db.fetch("SELECT * FROM invoices WHERE paid = 0 AND total >= ?", [min_amount]).to_a
  end

  def summary(year, month)
    @db.fetch_one(
      "SELECT SUM(total) as revenue, COUNT(*) as invoice_count " \
      "FROM invoices WHERE strftime('%Y', created_at) = ? " \
      "AND strftime('%m', created_at) = ?",
      [year.to_s, format("%02d", month)]
    )
  end
end

svc = AccountingService.new(db)

Tina4.mcp_tool("invoice_lookup", description: "Find an invoice by number", server: mcp,
  params: [{ name: "invoice_no", type: "string" }]) do |args|
  svc.lookup(args["invoice_no"])
end

Tina4.mcp_tool("outstanding_balances", description: "List all unpaid invoices", server: mcp,
  params: [{ name: "min_amount", type: "number", default: 0.0 }]) do |args|
  svc.balances(args["min_amount"] || 0.0)
end

Tina4.mcp_tool("monthly_summary", description: "Revenue summary for a month", server: mcp,
  params: [
    { name: "year", type: "integer" },
    { name: "month", type: "integer" }
  ]) do |args|
  svc.summary(args["year"], args["month"])
end
```

---

6. Securing MCP Endpoints

By default, developer MCP servers are public. Add authentication using Tina4 middleware:

```
# Secure the entire MCP path
Tina4.secured do
  mcp.register_routes
end
```

Or check the bearer token inside individual tools:

```
Tina4.mcp_tool("sensitive_data", description: "Access restricted data", server: mcp,
  params: [{ name: "token", type: "string" }]) do |args|
  payload = Tina4::Auth.valid_token(args["token"])
  unless payload
    next { error: "Unauthorized" }
  end
  db.fetch("SELECT * FROM sensitive_table").to_a
end
```

---

7. Testing MCP Tools

Test tool functions directly, or test via the MCP protocol:

```
# Test the tool block directly
def test_lookup_customer
  result = db.fetch_one("SELECT * FROM customers WHERE email = ?", ["alice@example.com"])
  assert result != nil
  assert_equal "alice@example.com", result["email"]
end

# Test via MCP protocol
def test_mcp_tool_call
  resp = JSON.parse(mcp.handle_message({
    "jsonrpc" => "2.0",
    "id" => 1,
    "method" => "tools/call",
    "params" => {
      "name" => "lookup_customer",
      "arguments" => { "email" => "alice@example.com" }
    }
  }))
  assert resp.key?("result")
  content = resp["result"]["content"][0]["text"]
  assert content.downcase.include?("alice")
end
```

---

8. Complete Example: CRM MCP Server

Here is a full working example -- a CRM system with customer, order, and product tools:

```
# app.rb
require "tina4"

db = Tina4::Database.new("sqlite:///crm.db")
Tina4::ORM.bind(db)

# Create MCP server
crm_mcp = Tina4::McpServer.new("/crm/mcp", name: "CRM Assistant", version: "1.0.0")

# Tools
Tina4.mcp_tool("find_customer", description: "Search customers by name or email", server: crm_mc
  params: [{ name: "query", type: "string" }]) do |args|
  q = args["query"]
  db.fetch("SELECT * FROM customers WHERE name LIKE ? OR email LIKE ?",
           ["%#{q}%", "%#{q}%"]).to_a
end

Tina4.mcp_tool("customer_orders", description: "Get all orders for a customer", server: crm_mcp,
  params: [{ name: "customer_id", type: "integer" }]) do |args|
  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",
```

```
    [args["customer_id"]]
  ).to_a
end

Tina4.mcp_tool("create_note", description: "Add a note to a customer record", server: crm_mcp,
  params: [
    { name: "customer_id", type: "integer" },
    { name: "note", type: "string" }
  ]) do |args|
  db.insert("customer_notes", { customer_id: args["customer_id"], note: args["note"] })
  { success: true }
end

# Resources
Tina4.mcp_resource("crm://products", description: "Product catalog", server: crm_mcp) do
  db.fetch("SELECT * FROM products WHERE active = 1").to_a
end

Tina4.mcp_resource("crm://stats", description: "CRM statistics", server: crm_mcp) do
  customers = db.fetch_one("SELECT COUNT(*) as count FROM customers")
  orders = db.fetch_one("SELECT COUNT(*) as count, SUM(total) as revenue FROM orders")
  {
    customers: customers["count"],
    orders: orders["count"],
    revenue: orders["revenue"]
  }
end

# Register routes
crm_mcp.register_routes

Tina4.run
```

Connect Claude Code to `http://localhost:7147/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 param metadata** -- types and defaults become the schema. An AI assistant cannot call a tool correctly without knowing the parameter types
- **Return structured data** -- hashes and arrays, not formatted strings. Let the AI format for the user
- **Secure production endpoints** -- use `Tina4.secured` or middleware for any MCP server that runs outside localhost

• **Test tools directly** -- call the Ruby method 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 the checkout endpoint. You open the dev dashboard. Find the failing request in the request inspector. Full stack trace with source code context. Line 47 of `src/routes/checkout.rb` -- a nil reference on the shipping address because the user skipped the form. Add a nil check. Push the fix. Go back to sleep. 30 seconds.

Tina4's dev tools are not an afterthought. They ship with the framework from day one. Set `TINA4_DEBUG=true` and you get a full development dashboard, an error overlay with source code, live reload, a request inspector with replay, a SQL query runner, a queue monitor, system info, and a gallery of ready-to-use code examples -- all without installing a single extra gem.

---

2. Enabling the Dev Dashboard

Set `TINA4_DEBUG=true` in your `.env`:

```
TINA4_DEBUG=true
```

Restart your server and navigate to:

```
http://localhost:7147/__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 Ruby version
- **Ruby version** -- The running Ruby version and loaded gems
- **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 Ruby 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	Ruby 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:

```
Tina4::Router.get("/api/data") do |request, response|
  response.json({ data: "no toolbar" }, 200, { "X-Debug-Toolbar" => "off" })
end
```

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 class and message** -- What went wrong, in plain language
- **Stack trace** -- Every method call that led to the error, from entry point to crash
- **Source code** -- The Ruby 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:

```
Tina4::Router.get("/api/users/{user_id}") do |request, response|
  user_id = request.params["user_id"]
  user = get_user_from_db(user_id)
  response.json({ name: user.name, email: user.email })
end
```

If `get_user_from_db` returns `nil` for a missing user, the error overlay shows:

```
NoMethodError: undefined method 'name' for nil:NilClass

  File: src/routes/users.rb, line 4

  2 |   user_id = request.params["user_id"]
  3 |   user = get_user_from_db(user_id)
> 4 |   response.json({ name: user.name, email: user.email })
  5 | end

  Request: GET /api/users/999
  Headers: {"Accept": "application/json"}
```

The highlighted line makes the cause obvious. `user` is `nil` and the code calls `.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. Create custom error pages:

- `src/templates/errors/404.html` -- Page not found
- `src/templates/errors/500.html` -- Internal server error

---

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 Ruby v3.11.12
  HTTP server running at http://0.0.0.0:7147
  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. There is no framework-side watcher (the `listen`-gem based `dev_reload.rb` was removed in 3.11.x).

- The CLI watches `src/`, `migrations/`, `.env` with the `notify` crate. Events are filtered to real source changes — metadata/access events, `__pycache__`, `.git`, `node_modules`, `vendor`, `logs`, `.log`/`.db*`/`.swp` files are ignored. A real mtime check defeats overlayfs / polling-mode spurious events (Podman, distrobox).
- On a real change, the CLI POSTs `/__dev/api/reload` to the running Ruby server. The server keeps running.
- `Tina4::DevAdmin` bumps its `@reload_mtime` 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 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.

---

8. 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 `puts` statements scattered through your code. The inspector shows what happened. Every request. Every detail. With one-click replay.

---

9. 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.

---

10. 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 method 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.

---

11. Log Viewer

View application logs in real time from the dashboard:

```
http://localhost:7147/__dev#logs
```

Filter by level (DEBUG, INFO, WARNING, ERROR), search by keyword, and view timestamps. In development, all log levels are shown. In production with `TINA4_LOG_LEVEL=WARNING`, only warnings and errors appear.

Logging from Code

```
Tina4::Logger.debug("Processing order #{order_id}")
Tina4::Logger.info("User #{user_id} logged in")
Tina4::Logger.warning("Rate limit approaching for IP #{ip}")
Tina4::Logger.error("Failed to send email: #{error.message}")
```

Logs are written to `logs/app.log` and displayed in the dev dashboard.

---

12. System Info

The System Info tab shows detailed information about your environment:

- **Ruby Configuration** -- Version, installed gems, GEM_HOME 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 Ruby version. A missing gem. A misconfigured environment variable. The answer is in the panel.

---

13. Health Check Endpoint

The `/health` endpoint reports application status:

```
curl http://localhost:7147/health
```

```
{
  "status": "ok",
  "database": "connected",
  "uptime_seconds": 3600,
  "version": "3.0.0",
  "framework": "tina4-ruby",
  "ruby_version": "3.3.0",
  "memory_mb": 42.5,
  "pid": 12345
}
```

In production with `TINA4_DEBUG=false`, create a `.broken` file in the project root to make the health check return a failure status. This is useful for graceful shutdown: set the file, wait for the load balancer to stop sending traffic, then restart the server.

---

14. Exercise: Debug a Broken Endpoint

Create a route with an intentional bug and use the dev tools to find and fix it.

Setup

Create `src/routes/buggy.rb`:

```
Tina4::Router.get("/api/buggy/users") do |request, response|
  users = Tina4.cache_get("all_users")
  if users
    return response.json({ users: users, source: "cache" })
  end

  # Bug 1: This query has an error
  db = Tina4.database
  users = db.fetch("SELCT * FROM users ORDER BY name")

  Tina4.cache_set("all_users", users, 300)
  response.json({ users: users, source: "database" })
end

Tina4::Router.post("/api/buggy/users") do |request, response|
  body = request.body

  # Bug 2: Missing validation
  user = User.new
  user.name = body["name"]
  user.email = body["email"]
  user.save

  # Bug 3: Wrong status code
  response.json({ message: "User created", user: user.to_hash })
end
```

Requirements

- Open the dev dashboard at `http://localhost:7147/__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

---

15. Solution

```
Tina4::Router.get("/api/buggy/users") do |request, response|
  users = Tina4.cache_get("all_users")
  if users
    return response.json({ users: users, source: "cache" })
  end

  db = Tina4.database
  users = db.fetch("SELECT * FROM users ORDER BY name")  # Fixed: SELCT -> SELECT

  Tina4.cache_set("all_users", users, 300)
  response.json({ users: users, source: "database" })
end

Tina4::Router.post("/api/buggy/users") do |request, response|
  body = request.body

  # Fixed: Added validation
  if body.nil? || body["name"].nil? || body["email"].nil?
    return response.json({ error: "Name and email are required" }, 400)
  end

  user = User.new
  user.name = body["name"]
  user.email = body["email"]
  user.save

  response.json({ message: "User created", user: user.to_hash }, 201)  # Fixed: 201
end
```

The dev tools made each bug visible. The error overlay showed the SQL syntax error with the exact line. The request inspector showed the nil body causing a `NoMethodError`. Request Replay let you re-send the fixed requests without leaving the dashboard.

---

16. 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: Save files less often during WebSocket development, or use a separate terminal to run the WebSocket test client that auto-reconnects. The frond.js WebSocket client has built-in auto-reconnect.

3. Error Overlay Shows in Production

Problem: Users see Ruby 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. 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.

7. Log Files Growing Without Bound

Problem: `logs/app.log` grows to gigabytes over time.

Cause: No log rotation is configured.

Fix: Set `TINA4_LOG_MAX_SIZE=10mb` and `TINA4_LOG_KEEP=5` in `.env` to rotate logs automatically.

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."

9. Memory Usage Grows During Development

Problem: The server's memory usage increases over time during development.

Cause: Each live reload may not fully release old objects from the Ruby garbage collector.

Fix: This is normal during active development. Restart the server periodically during long development sessions. Ruby's GC will clean up on restart.

Tina4 CLI

1. Getting a New Developer Up to Speed

Monday morning. New developer joins the 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 hunts. No boilerplate from old projects. The CLI handles the scaffolding.

The Tina4 CLI is a single Rust binary. It manages all four Tina4 frameworks -- PHP, Python, Ruby, Node.js. Commands are identical across languages. Learn the CLI for Ruby, and 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: Ruby (Gemfile)
  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
  bundle install
  tina4 serve
```

Language Detection

The CLI detects the language from existing files:

File Present	Language

`Gemfile`	Ruby
`composer.json`	PHP
`pyproject.toml` or `requirements.txt`	Python
`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
> 3

Creating Tina4 Ruby project in ./my-project ...
```

Explicit Language Selection

Skip the prompt by specifying the language:

```
tina4 init my-project --lang ruby
```

This creates a Ruby project with `Gemfile`, `app.rb`, 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 Ruby v3.0.0
  HTTP server running at http://0.0.0.0:7147
  WebSocket server running at ws://0.0.0.0:7147
  Live reload enabled
  Press Ctrl+C to stop
```

`tina4 serve` detects the language and starts the appropriate server. For Ruby, it spawns the Tina4 Ruby 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, CI runners, APM agents) set `TINA4_OVERRIDE_CLIENT=true` in `.env` and only then run:

```
TINA4_OVERRIDE_CLIENT=true bundle exec ruby app.rb
```

This bypasses SCSS compilation and live reload. For production deployments use `tina4 serve --production` (or set the override and run Puma directly):

```
TINA4_OVERRIDE_CLIENT=true bundle exec puma -C config/puma.rb
```

---

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.rb
Created migrations/20260322120000_create_products_table.sql
```

The generated model:

```
class Product < Tina4::ORM
  integer_field :id, primary_key: true
  created_at: true

  table_name "products"
end
```

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:

```
class Product < Tina4::ORM
  integer_field :id, primary_key: true
  string_field :name
  float_field :price
  string_field :category
```

```
  boolean_field :in_stock

  table_name "products"
end
```

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	Ruby Field	SQLite Column
`string`	`string_field`	`TEXT`
`int`	`integer_field`	`INTEGER`
`float`	`float_field`	`REAL`
`bool`	`boolean_field`	`INTEGER`
`text`	`text_field`	`TEXT`
`date`	`string_field`	`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 type casting and nil checks.

```
tina4 generate route products
```

```
Created src/routes/products.rb
```

The generated route file:

```
Tina4::Router.get("/api/products") do |request, response|
  page = (request.params["page"] || 1).to_i
  per_page = (request.params["per_page"] || 20).to_i
  offset = (page - 1) * per_page

  products, total = Product.where("1=1", [], limit: per_page, offset: offset)
  results = products.map(&:to_hash)

  response.json({
    data: results,
    page: page,
    per_page: per_page,
    count: results.length
  })
end

Tina4::Router.get("/api/products/{product_id:int}") do |request, response|
  product = Product.find(request.params["product_id"])

  if product.nil?
    return response.json({ error: "Product not found" }, 404)
  end

  response.json(product.to_hash)
end

Tina4::Router.post("/api/products") do |request, response|
  body = request.body

  product = Product.new
  product.name = body["name"] || ""
  product.price = (body["price"] || 0).to_f
  product.category = body["category"] || ""
  product.in_stock = body["in_stock"] ? true : false
  product.save

  response.json(product.to_hash, 201)
end

Tina4::Router.put("/api/products/{product_id:int}") do |request, response|
  product = Product.find(request.params["product_id"])

  if product.nil?
    return response.json({ error: "Product not found" }, 404)
  end

  body = request.body
```

```
  product.name = body["name"] if body.key?("name")
  product.price = body["price"].to_f if body.key?("price")
  product.category = body["category"] if body.key?("category")
  product.in_stock = body["in_stock"] if body.key?("in_stock")
  product.save

  response.json(product.to_hash)
end

Tina4::Router.delete("/api/products/{product_id:int}") do |request, response|
  product = Product.find(request.params["product_id"])

  if product.nil?
    return response.json({ error: "Product not found" }, 404)
  end

  product.delete
  response.json(nil, 204)
end
```

The generator reads the model's properties and creates routes with type casting and nil checks. Customize the generated code. It is regular Ruby, 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.

---

7. tina4 generate middleware -- Middleware Scaffolding

The `generate middleware` command creates a middleware file with the correct signature.

```
tina4 generate:middleware rate_limit
```

```
Created src/middleware/rate_limit.rb
```

The generated file:

```
# Rate limit middleware
Tina4.middleware do |request, response, next_handler|
  # Add your middleware logic here

  # Continue to the next middleware or route handler
  next_handler.call(request, response)

  # Or return early to block the request:
  # response.json({ error: "Rate limit exceeded" }, 429)
end
```

Reference the middleware in route definitions:

```
Tina4::Router.get("/api/data", middleware: ["rate_limit"]) do |request, response|
  response.json({ data: "protected" })
end
```

---

8. Generate All at Once

Combine flags to generate multiple files in a single command:

```
tina4 generate:crud Product
```

```
Created src/orm/product.rb
Created src/routes/products.rb
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] Ruby 3.3.0 detected
```

```
  [OK] Bundler found
  [OK] tina4 gem 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:

- Ruby version and package manager
- Tina4 gem 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...

  ProductSpec
    [PASS] creates a product
    [PASS] loads a 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 tests/product_spec.rb            # Specific file
tina4 test --verbose                         # Verbose output
tina4 test --coverage                        # Generate coverage report
```

---

11. tina4 routes -- Route Listing

See all registered routes in your project:

```
tina4 routes
```

```
Registered Routes:

  Method  Path                          Handler              Middleware        Auth
  ------  --------------------------    -------------------  ----------------  --------
  GET     /health                       health_check         -                 public
  GET     /api/products                 list_products        ResponseCache     public
  GET     /api/products/{id:int}        get_product          -                 public
  POST    /api/products                 create_product       auth_middleware   secured
  PUT     /api/products/{id:int}        update_product       auth_middleware   secured
  DELETE  /api/products/{id:int}        delete_product       auth_middleware   secured
  GET     /admin                        admin_dashboard      -                 public

  7 routes registered
```

This is useful for verifying that your routes are registered and for finding the handler 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 early v3 release, the `tina4_migration` tracking table may use the older two-column 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
```

Other Migration Commands

```
tina4 migrate:reset        # Roll back all migrations
tina4 migrate:fresh        # Roll back and re-run all migrations
```

---

13. tina4 queue -- Queue Management

```
tina4 queue work                    # Start processing all queues
tina4 queue work --queue emails     # Process specific queue
tina4 queue:dead                    # View dead letter queue
tina4 queue:retry 42                # Retry a dead letter job
tina4 queue:retry --all             # Retry all dead letter jobs
tina4 queue:clear --older-than 7d   # Clear old dead letter jobs
tina4 queue:stats                   # Show queue statistics
```

---

14. tina4 build -- Build Commands

```
tina4 scss                    # Compile SCSS to CSS
tina4 build                     # Bundle and minify JavaScript
tina4 build                        # Run all build steps
```

---

15. tina4 deploy -- Deployment

```
tina4 deploy docker                 # Generate Dockerfile and docker-compose.yml
tina4 deploy:systemd                # Generate systemd service file
tina4 deploy:nginx                  # Generate Nginx config
```

---

16. Environment-Specific Commands

```
tina4 serve --env production         # Use .env.production
tina4 migrate --env test             # Run migrations on test database
tina4 test --env test                # Run tests with test environment
```

---

17. Exercise: Scaffold a Feature in 5 Commands

Scaffold a complete "Customer" feature from scratch using only CLI commands.

Requirements

Starting from an existing Tina4 Ruby 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:crud Customer

# 2. Edit the model to add fields (manual step)
# Add fields to src/orm/customer.rb

# 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
```

---

18. Solution

Command 1: Generate everything:

```
tina4 generate:crud Customer
```

Command 2: Edit `src/orm/customer.rb`:

```
class Customer < Tina4::ORM
  integer_field :id, primary_key: true
  string_field :name
  string_field :email
  string_field :phone
  string_field :company
```

```
  table_name "customers"
end
```

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:7147/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.

---

19. 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 Ruby.

Cause: A `composer.json` file exists in the directory from a previous project.

Fix: Use explicit language selection: `tina4 init my-project --lang ruby`. 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.

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.rb src/routes/products_backup.rb`.

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. Generate migrations in order to ensure correct execution. 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 7147 but you need port 8080.

Cause: The default port for Ruby is 7147 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. Model Name Must Be PascalCase

Problem: `tina4 generate model order_item` creates a model class named `order_item` which is not valid Ruby 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`).

8. build:css Fails

Problem: `tina4 scss` fails with "sass not found".

Cause: The Sass compiler is not installed.

Fix: Install it: `gem install sass` or `npm install -g sass`.

9. deploy:docker Generates Wrong Dockerfile

Problem: The generated Dockerfile uses PHP instead of Ruby.

Cause: The CLI cannot detect the language without a `Gemfile`.

Fix: Make sure your project has a `Gemfile` so the CLI detects Ruby. Or use `tina4 deploy docker --lang ruby`.

---

20. 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.

---

21. Test Port (Dual-Port Development)

When `TINA4_DEBUG=true`, Tina4 automatically starts a second HTTP server on `port + 1000`:

- **Main port** (e.g. 7147) — hot-reload enabled, for AI dev tools
- **Test port** (e.g. 8147) — 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 know which packages to install, which config files to create, which naming conventions to follow, how to wire everything together. It hallucinated 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 attempt because there is one way to do things in Tina4.

This is not accidental. Tina4 was designed from the ground up for AI-assisted development. Here is why.

The Zero-Dependency Advantage

When an AI writes Rails code, it might suggest `ActiveRecord::Base` or `ApplicationRecord`. Both exist. Both behave differently. It might suggest `render json:` or `respond_to do |format|`. The AI guesses.

Tina4 gives it nothing to guess about. One ORM. One queue. One template engine. One cache. No ambiguity. No alternatives. No "it depends on which gem you installed." The AI knows.

```
# There is only one way to cache in Tina4
cached = Tina4.cache_get("products") || Tina4.cache_set("products", db.fetch("SELECT * FROM prod

# There is only one way to queue in Tina4
Tina4::Queue.produce("emails", { to: "user@test.com", subject: "Welcome" })

# There is only one way to send email in Tina4
mail = Tina4::Messenger.new
mail.to = "user@test.com"
mail.subject = "Welcome"
mail.send
```

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 use `response.json`, ORM uses `to_h`)

- 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.

Supported AI Tools

Tina4 auto-detects and installs context for eight tools. The exact list lives in the framework's `Tina4::AI` module:

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 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 and set `TINA4_AI_MODEL` accordingly.

The Convention Advantage

AI thrives on convention. When every Tina4 project follows the same structure, the AI never has to ask "where should I put this?"

```
src/
  routes/hello.rb     -> AI knows: this is a route file
  orm/product.rb      -> AI knows: this is an ORM model
  templates/page.html -> AI knows: this is a Frond template
  migrations/         -> AI knows: SQL migration files go here
```

The AI generates code that drops into the right directory with the right naming convention, every time.

Prompt Engineering for Tina4

Here are effective prompts for AI-assisted Tina4 Ruby development:

Creating a New Feature

```
Add a product catalog to my Tina4 Ruby project:
1. Create a Product model with fields: name, category, price, in_stock
2. Create a migration for the products table
3. Create CRUD routes at /api/products
4. Create a product listing template with category filters
5. Add response caching for the product list (5 minutes)
```

The AI generates all five files with correct Tina4 syntax because it read CLAUDE.md.

Adding Authentication

```
Add JWT authentication to my Tina4 Ruby project:
1. Create a users table migration with name, email, password_hash, role
2. Create register and login routes at /api/auth/register and /api/auth/login
3. Create auth middleware that validates Bearer tokens
4. Protect the /api/tasks group with auth middleware
5. Use Tina4::Auth.get_token and Tina4::Auth.valid_token
```

Building a Dashboard

```
Create an admin dashboard for my Tina4 Ruby project:
1. Create a GET /admin route that renders a dashboard template
2. The dashboard shows: total users, total products, recent orders
3. Use tina4css for styling (cards, tables, grid)
4. Use frond.js for AJAX data loading
5. Cache the dashboard stats for 60 seconds
```

Adding WebSocket

```
Add a real-time notification system to my Tina4 Ruby project:
1. Create a WebSocket endpoint at /ws/notifications/{user_id}
2. When a task is assigned, push a notification via WebSocket
3. Include a JavaScript snippet that connects and shows browser notifications
4. Auto-reconnect on disconnect
```

The Ruby Advantage for AI

Ruby's expressiveness makes AI-generated code particularly clean:

Block Syntax

```ruby
Tina4::Router.get("/api/products") do |request, response|
  products = Tina4.cache_get("products:all")
  if products.nil?
    db = Tina4.database
    products = db.fetch("SELECT * FROM products ORDER BY name")
    Tina4.cache_set("products:all", products, 300)
  end

  response.json({ products: products, count: products.length })
end
```

The block syntax is unambiguous. The AI knows exactly where the route handler starts and ends. No curly brace confusion, no indentation ambiguity.

Hash Syntax

```ruby
response.json({
  user: { id: user.id, name: user.name, email: user.email },
  tasks: tasks.map(&:to_h),
  stats: { total: total, completed: completed }
})
```

Ruby's symbol-key hash syntax (`name:` instead of `"name" =>`) is concise and consistent. The AI generates clean, idiomatic Ruby.

Enumerable Methods

```
# Filter products
electronics = products.select { |p| p[:category] == "Electronics" }

# Transform data
names = products.map { |p| p[:name] }

# Aggregate
total = items.sum { |item| item[:price] * item[:quantity] }

# Count with condition
featured_count = products.count { |p| p[:featured] }
```

These are the Ruby idioms the AI knows and uses. No external libraries, no complex method chains -- just Ruby.

Vibe Coding Workflow

The "vibe coding" workflow with AI and Tina4 looks like this:

1. Describe What You Want

Tell the AI what you need in plain English. Be specific about the data model, the endpoints, and the behavior.

2. Review the Generated Code

The AI generates the routes, models, migrations, and templates. Review them for correctness and completeness.

3. Run and Test

```
tina4 migrate
tina4 serve
```

Test the endpoints with curl or the Swagger UI. If something is wrong, tell the AI what needs to change.

4. Iterate

"The product list should be paginated. Add page and per_page query parameters." The AI updates the code.

5. Deploy

```
docker compose build
docker compose up -d
```

From idea to production in minutes, not days.

Real-World Example: Building an API in 5 Minutes

Here is a real conversation with an AI assistant:

You: "Add a blog to my Tina4 Ruby app. Posts with title, body, published flag, and user_id. CRUD API at /api/posts. Only published posts should be visible to anonymous users. Authors can see their own drafts."

AI generates:

- `migrations/20260322_create_posts_table.sql` -- the migration
- `src/orm/post.rb` -- the model with fields and relationships
- `src/routes/posts.rb` -- CRUD routes with authentication and filtering
- `tests/posts_spec.rb` -- RSpec tests for all scenarios

All using correct Tina4 conventions:

```
Tina4::Router.get("/api/posts") do |request, response|
  db = Tina4.database

  # Check if user is authenticated (optional)
  user_id = nil
  auth_header = request.headers["Authorization"] || ""
  if auth_header.start_with?("Bearer ")
    token = auth_header.sub("Bearer ", "")
    if Tina4::Auth.valid_token(token)
      payload = Tina4::Auth.get_payload(token)
      user_id = payload["user_id"]
    end
  end

  if user_id
    # Authenticated: show published + own drafts
    posts = db.fetch(
      "SELECT * FROM posts WHERE published = 1 OR user_id = :user_id ORDER BY created_at DESC",
      { user_id: user_id }
    )
  else
    # Anonymous: show published only
    posts = db.fetch("SELECT * FROM posts WHERE published = 1 ORDER BY created_at DESC")
  end

  response.json({ posts: posts, count: posts.length })
end
```

The AI knows:

- `Tina4::Router.get` for routes
- `request.headers["Authorization"]` for the token
- `Tina4::Auth.valid_token` and `Tina4::Auth.get_payload` for JWT
- `db.fetch` with named parameters for queries
- `response.json` for JSON responses

It did not hallucinate. It did not guess. It read CLAUDE.md and wrote correct code.

Why This Matters

Traditional development: write every line, look up every API, debug every typo. A simple CRUD feature takes an hour.

Vibe coding with Tina4: describe what you want. The AI generates correct code. You review and deploy. The same feature takes 5 minutes.

The insight: AI can only be as good as the framework it targets. A framework with 200 gems, 15 configuration files, and 3 ways to do everything gives the AI too many choices. A framework with zero dependencies, one way to do everything, and a complete reference in CLAUDE.md gives the AI exactly what it needs.

Every Tina4 design decision -- zero deps, convention over configuration, identical API across 4 languages, CLAUDE.md -- exists to make AI-assisted development work.

You bring the ideas. The AI brings the implementation. Tina4 is the bridge.

Exercise: Vibe-Code a Feature

Open your Tina4 Ruby project with an AI assistant (Claude Code, Cursor, or Copilot) and try this prompt:

```
Add a comment system to my blog:
1. Create a comments table with post_id, author_name, body, created_at
2. Create a Comment model with belongs_to :post relationship
3. Add POST /api/posts/{id}/comments to add a comment
4. Add GET /api/posts/{id}/comments to list comments
5. Modify GET /api/posts/{id} to include comments
6. Add RSpec tests for the comment endpoints
```

Verify the AI generates:

- A migration file in `migrations/`
- A model file in `src/orm/`
- Route handlers in `src/routes/`
- Test file in `tests/`

Run `tina4 migrate && tina4 test` to verify everything works.

Gotchas

1. AI Generates Rails Syntax Instead of Tina4

Problem: The AI writes `render json:` instead of `response.json`.

Cause: The AI defaulted to Rails conventions.

Fix: Make sure CLAUDE.md is in your project root. Mention "Tina4 Ruby" explicitly in your prompt.

2. AI Invents Non-Existent Methods

Problem: The AI calls `Tina4::Router.resource` which does not exist.

Cause: The AI hallucinated a method from another framework.

Fix: Review generated code against the CLAUDE.md reference. Ask the AI to only use methods documented in CLAUDE.md.

3. AI Generates Tests That Do Not Match the Implementation

Problem: Tests reference endpoints or response formats that differ from the actual routes.

Fix: Generate routes first, then ask the AI to generate tests that match the existing routes.

4. AI Forgets to Run Migrations

Problem: The app crashes because the table does not exist.

Fix: Always run `tina4 migrate` after generating migrations. Include it in your prompts: "...and tell me the commands to run."

5. AI Uses require Instead of Auto-Loading

Problem: The AI adds `require_relative "../orm/product"` at the top of route files.

Cause: Ruby convention is to require files explicitly, but Tina4 auto-loads everything in `src/routes/` and `src/orm/`.

Fix: Remove the require statements. Tina4 handles loading automatically.

6. AI Generates Over-Engineered Solutions

Problem: The AI creates service objects, repository patterns, and dependency injection for a simple CRUD feature.

Fix: Tell the AI: "Keep it simple. Use Tina4's built-in features only. No extra abstractions." Tina4's philosophy is simplicity -- the AI should follow it.

7. AI Does Not Know About to_h

Problem: The AI serializes model objects with `to_json` instead of `to_h`.

Fix: Tina4 ORM objects use `to_h` (Ruby idiom for hash conversion) followed by `response.json`. The CLAUDE.md file documents this pattern.

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 Ruby 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 Ruby 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`)*	Tina4-specific alias for `HOST`.
`PORT`	`7147`	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:7147`	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. Falls back to `RACK_ENV` then `RUBY_ENV`.
`RACK_ENV`	*(none)*	Rack-standard environment label. Used if `TINA4_ENV` is unset.
`RUBY_ENV`	*(none)*	Ruby-ecosystem environment label. Used if `TINA4_ENV` and `RACK_ENV` are unset.

`TINA4_NO_BROWSER`	`false`	Stops `tina4 serve` from opening your browser on every restart. Recommended during active development.
`TINA4_NO_RELOAD`	`false`	Disables the dev hot-reload signal from the Rust CLI. Use when you want a stable server for debugging.
`ENVIRONMENT`	*(none)*	PaaS-style environment label. Read by tooling alongside `RACK_ENV`/`RUBY_ENV`.
`TINA4_SHUTDOWN_TIMEOUT`	`10`	Seconds the supervisor waits for graceful shutdown before terminating workers.
`TINA4_SUPERVISOR_URL`	*(derived from `TINA4_PORT` + 2000)*	Override URL for the local Rust CLI supervisor used by the dev dashboard.
`TINA4_TEMPLATE_ROUTING`	`on`	Controls automatic template-to-route binding. Set to `off`, `false`, `0`, `no`, or `disabled` to require explicit route declarations.
`TINA4_ALLOW_LEGACY_ENV`	`false`	Bypass the v3.12 legacy-env startup guard. Intended for CI/migration scripts only.
`TINA4_SUPPRESS`	`false`	Suppresses the startup banner. Useful for headless processes and CI runs.
`TINA4_ENV_FILE`	`.env`	Path or filename of the dotenv file loaded at boot. Resolved against the project root when relative.
`TINA4_HEALTH_PATH`	`/__health`	Health-check route mounted by `Tina4::Health`. The legacy `/health` path stays registered as an alias unless it's the configured value.

Routing

Variable	Default	Description
`TINA4_TRAILING_SLASH_REDIRECT`	`false`	When truthy, the Rack app issues a `301` redirect that strips trailing slashes from request paths.

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_JWT_ALGORITHM`	`HS256`	JWT signing algorithm.
`TINA4_API_KEY`	*(empty)*	Static API key used by `Tina4::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`	*(required for non-SQLite)*	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_DATABASE_URL`	*(empty)*	Legacy alias for `TINA4_DATABASE_URL`.
`TINA4_AUTOCOMMIT`	`false`	Auto-commit after every write. Default is off — call `commit` explicitly.
`TINA4_DB_CACHE`	`false`	Enables in-memory query-result caching for read queries.
`TINA4_DB_CACHE_TTL`	`30`	Query cache TTL in seconds when `TINA4_DB_CACHE=true`.
`TINA4_ORM_PLURAL_TABLE_NAMES`	`true`	When `true`, the ORM pluralises class names into table names (`User` → `users`). Set `false` to keep them singular.
`TINA4_DB_POOL`	`0`	Connection pool size used by `Tina4::Database.new(url, pool:)` when the caller doesn't pass `pool:` explicitly. `0` disables pooling.

CORS

Variable	Default	Description
`TINA4_CORS_ORIGINS`	`*`	Comma-separated allowed origins. Lock down to real domains in production.
`TINA4_CORS_METHODS`	`GET, POST, PUT, PATCH, DELETE, OPTIONS`	Allowed request methods.
`TINA4_CORS_HEADERS`	`Content-Type,Authorization,X-Request-ID`	Allowed request headers.
`TINA4_CORS_CREDENTIALS`	`false`	Send `Access-Control-Allow-Credentials: true`.
`TINA4_CORS_MAX_AGE`	`86400`	Preflight cache lifetime in seconds.

Security Headers

Variable	Default	Description
`TINA4_CSP`	`default-src 'self'`	`Content-Security-Policy` header.
`TINA4_CSRF`	`false`	CSRF token validation on POST/PUT/PATCH/DELETE. Off by default in Ruby; enable with `true`.
`TINA4_HSTS`	*(empty/off)*	`Strict-Transport-Security` max-age in seconds. Set `31536000` in production with HTTPS.
`TINA4_FRAME_OPTIONS`	`SAMEORIGIN`	`X-Frame-Options` header.
`TINA4_REFERRER_POLICY`	`strict-origin-when-cross-origin`	`Referrer-Policy` header.
`TINA4_PERMISSIONS_POLICY`	`camera=(), microphone=(), geolocation=()`	`Permissions-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`	`3600`	Session expiry in seconds.
`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`	Session cookie name. Used unless the caller passes an explicit `cookie_name`.
`TINA4_SESSION_HTTPONLY`	`true`	Sets the `HttpOnly` attribute on the session cookie. Set `false` to allow JavaScript access.
`TINA4_SESSION_SECURE`	`false`	Sets the `Secure` attribute on the session cookie. Enable in production behind HTTPS.

Valkey/Redis session backend

Variable	Default	Description
`TINA4_SESSION_VALKEY_HOST`	`localhost`	Valkey/Redis host for the session handler.
`TINA4_SESSION_VALKEY_PORT`	`6379`	Valkey/Redis port.
`TINA4_SESSION_VALKEY_PASSWORD`	*(none)*	Valkey/Redis auth password.
`TINA4_SESSION_VALKEY_DB`	`0`	Valkey/Redis database number.
`TINA4_SESSION_VALKEY_PREFIX`	`tina4:session:`	Key prefix used for session entries.
`TINA4_SESSION_VALKEY_TTL`	`86400`	Per-key expiry in seconds.

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`	`0`	Default cache TTL in seconds (`0` disables caching for the global instance; the singleton helper uses `60` when unset).
`TINA4_CACHE_MAX_ENTRIES`	`1000`	Maximum cache entries before eviction.
`TINA4_CACHE_URL`	`redis://localhost:6379`	Connection URL for the Redis backend.

Templates

Variable	Default	Description
`TINA4_TEMPLATE_CACHE_TTL`	`0`	Frond template cache TTL in seconds. `0` keeps compiled templates cached permanently; positive values expire entries after `N` seconds.

GraphQL

Variable	Default	Description
`TINA4_GRAPHQL_AUTO_SCHEMA`	`true`	When `true`, the GraphQL module auto-generates a schema from registered ORM models. Set `false` to require an explicit schema.
`TINA4_GRAPHQL_ENDPOINT`	`/graphql`	Path the GraphQL handler is mounted at when the caller doesn't pass an explicit path.

Queues

Variable	Default	Description
`TINA4_QUEUE_BACKEND`	`file`	Queue backend. Options: `file`, `kafka`, `rabbitmq`, `mongo`, `database`.
`TINA4_QUEUE_URL`	*(none)*	Connection URL for remote backends. Parsed for `host`/`port` when set.

Kafka queue backend

Variable	Default	Description
`TINA4_KAFKA_BROKERS`	*(none)*	Comma-separated broker list (e.g. `localhost:9092`).
`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. Options: `redis`, `nats`. Required for multi-instance broadcasts.
`TINA4_WS_BACKPLANE_URL`	`redis://localhost:6379` (Redis) / `nats://localhost:4222` (NATS)	Connection URL for the chosen backplane.

Email

Variable	Default	Description
`TINA4_MAIL_HOST`	`localhost`	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`	`dev@localhost`	Default sender email address.
`TINA4_MAIL_FROM_NAME`	*(empty)*	Default sender display name.
`TINA4_MAIL_ENCRYPTION`	`tls`	Connection encryption. Options: `tls`, `ssl`, `none`.
`TINA4_MAIL_IMAP_HOST`	*(inherits mail host)*	IMAP server for inbound mail.
`TINA4_MAIL_IMAP_PORT`	`993`	IMAP server port.
`TINA4_MAIL_IMAP_USERNAME`	*(inherits SMTP username)*	IMAP authentication username. Mapped from legacy `IMAP_USER`.
`TINA4_MAIL_IMAP_PASSWORD`	*(inherits SMTP password)*	IMAP authentication password. Mapped from legacy `IMAP_PASS`.
`TINA4_MAIL_IMAP_ENCRYPTION`	`tls`	IMAP connection encryption. Accepts `tls`, `starttls`, or `none`.
`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

Variable	Default	Description
`TINA4_LOG_LEVEL`	`[TINA4_LOG_ALL]`	Console log level. Options: `[TINA4_LOG_ALL]`, `[TINA4_LOG_DEBUG]`, `[TINA4_LOG_INFO]`, `[TINA4_LOG_WARNING]`, `[TINA4_LOG_ERROR]`, `[TINA4_LOG_NONE]`. Also accepts plain `DEBUG`, `INFO`, `ERROR`, etc.
`TINA4_LOG_MAX_SIZE`	`10`	Per-file log size limit in megabytes. Rotated when exceeded.
`TINA4_LOG_KEEP`	`5`	Number of rotated log files to retain.

Logs default to stdout. Set `TINA4_LOG_OUTPUT=file` plus `TINA4_LOG_FILE=app.log` to write to disk; Ruby's stdlib `Logger` rotates at `TINA4_LOG_ROTATE_SIZE` bytes and keeps `TINA4_LOG_ROTATE_KEEP` backups natively.

Variable	Default	Description
`TINA4_LOG_FILE`	*(empty = stdout only)*	Explicit log file path. Absolute, or resolved relative to `TINA4_LOG_DIR`.
`TINA4_LOG_DIR`	`logs`	Directory used when `TINA4_LOG_FILE` is set without an absolute path.
`TINA4_LOG_FORMAT`	`text`	Log line format. Accepts `text` or `json`.
`TINA4_LOG_OUTPUT`	`stdout`	Output sink. Accepts `stdout`, `file`, or `both`.
`TINA4_LOG_CRITICAL`	`false`	When `true`, gates the new `critical` level and raises on log write failures instead of swallowing them.

`TINA4_LOG_ROTATE_SIZE`	`10485760`	Per-file rotation threshold in bytes (10 MB). `0` disables rotation. Handled natively by stdlib `Logger.new(path, shift_age, shift_size)`.
`TINA4_LOG_ROTATE_KEEP`	`5`	Number of rotated backups to retain.

Uploads

Variable	Default	Description
`TINA4_MAX_UPLOAD_SIZE`	`10485760`	Maximum multipart upload size in bytes (10 MB). Requests larger than this are rejected before parsing.

Services (background tasks)

Variable	Default	Description
`TINA4_SERVICE_DIR`	`src/services`	Directory scanned for service classes.
`TINA4_SERVICE_SLEEP`	`5`	Seconds the service runner sleeps between iterations.

Localisation

Variable	Default	Description
`TINA4_LOCALE`	`en`	Default locale for `Tina4::Localization`.
`TINA4_LOCALE_DIR`	`src/locale`	Directory containing locale JSON files.

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).
`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.
`TINA4_AI_MODEL`	`nomic-embed-text`	Embedding model used to index the framework and `src/`.
`TINA4_MCP_REMOTE`	`false`	Allow the MCP server to bind on non-localhost interfaces. **Never enable in production.**
`TINA4_NO_AI_PORT`	`false`	Disables the MCP port listener in dev mode.
`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.

MCP

Variable	Default	Description
`TINA4_MCP`	*(inherits `TINA4_DEBUG`)*	Enables the MCP server. Defaults to whatever `TINA4_DEBUG` is set to; pass `true`/`false` to override explicitly.
`TINA4_MCP_PORT`	*(derived from `TINA4_PORT` + 2000)*	Port the MCP server binds to. Defaults to the HTTP port plus `2000`.

Swagger / OpenAPI

Variable	Default	Description
`TINA4_SWAGGER_TITLE`	`Tina4 API`	OpenAPI spec title. Falls back to `PROJECT_NAME`.
`TINA4_SWAGGER_DESCRIPTION`	`Auto-generated API documentation`	OpenAPI spec description.
`TINA4_SWAGGER_VERSION`	*(Gem version)*	Overrides the spec `version`.
`TINA4_SWAGGER_ENABLED`	*(inherits `TINA4_DEBUG`)*	Enables the Swagger UI and `/swagger.json` route. Defaults to `TINA4_DEBUG`; set explicitly to override.
`TINA4_SWAGGER_CONTACT_EMAIL`	*(empty)*	Contact email rendered into the OpenAPI `info.contact` block.
`TINA4_SWAGGER_LICENSE`	*(empty)*	SPDX license name (e.g. `MIT`) rendered into the OpenAPI `info.license` block.
`PROJECT_NAME`	*(none)*	Alternative OpenAPI title source when `TINA4_SWAGGER_TITLE` is unset.

Minimal `.env` for Development

```
TINA4_DEBUG=true
TINA4_LOG_LEVEL=ALL
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:7147`. 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, Puma 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 gem dependencies and the second stage copies only the 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=7147

# 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. Puma Configuration

Puma is the production server for Tina4 Ruby. It runs multiple worker processes, handles concurrent requests, and supports graceful shutdown.

Create `config/puma.rb`:

```
# Port
port ENV.fetch("TINA4_PORT", 7147)

# Workers (processes) -- set to number of CPU cores
workers ENV.fetch("WEB_CONCURRENCY", 2)

# Threads per worker — Puma's own config var, not a Tina4 setting.
threads_count = ENV.fetch("PUMA_MAX_THREADS", 5)
threads threads_count, threads_count

# Environment
environment ENV.fetch("RACK_ENV", "production")

# Preload app for faster worker boot
preload_app!

# PID file
pidfile ENV.fetch("PIDFILE", "tmp/pids/server.pid")

# Logging
```

```
stdout_redirect "logs/puma.stdout.log", "logs/puma.stderr.log", true

# Worker timeout
worker_timeout 60

# Graceful shutdown
on_worker_boot do
  # Reconnect to database after fork
  Tina4::Database.reconnect
end
```

Start with Puma:

```
bundle exec puma -C config/puma.rb
```

How Many Workers?

Start with `(2 * CPU cores) + 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

The built-in WEBrick server is for development only. It handles one request at a time. Always use Puma in production.

---

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.

Dockerfile

```
FROM ruby:3.3-slim

WORKDIR /app

# Install build dependencies
RUN apt-get update && apt-get install -y --no-install-recommends \
    build-essential \
    libsqlite3-dev \
    curl \
    && rm -rf /var/lib/apt/lists/*

# Install gems (copy lock file for better layer caching)
COPY Gemfile Gemfile.lock ./
RUN bundle install --without development test

# Copy application code
COPY . .

# Create necessary directories
RUN mkdir -p data logs secrets tmp/pids

# Expose port
EXPOSE 7147

# Health check
HEALTHCHECK --interval=30s --timeout=5s --retries=3 \
  CMD curl -f http://localhost:7147/health || exit 1

# Start with Puma
CMD ["bundle", "exec", "puma", "-C", "config/puma.rb"]
```

.dockerignore

```
.git
.env
*.gem
.bundle
tmp
log
data/*.db
logs/*.log
.claude
node_modules
spec
```

Docker Compose

For a complete setup with supporting services:

```
services:
  app:
    build: .
    ports:
      - "7147:7147"
    environment:
      - TINA4_DEBUG=false
      - TINA4_LOG_LEVEL=WARNING
      - JWT_SECRET=${JWT_SECRET}
```

```
      - TINA4_DATABASE_URL=sqlite:///data/app.db
      - TINA4_CACHE_BACKEND=redis
      - TINA4_CACHE_URL=redis
    volumes:
      - app-data:/app/data
      - app-logs:/app/logs
    depends_on:
      - redis
    restart: unless-stopped
    healthcheck:
      test: ["CMD", "curl", "-f", "http://localhost:7147/health"]
      interval: 30s
      timeout: 5s
      retries: 3

  worker:
    build: .
    command: tina4 queue work
    environment:
      - TINA4_DATABASE_URL=sqlite:///data/app.db
    volumes:
      - app-data:/app/data
    depends_on:
      - app
    restart: unless-stopped

  redis:
    image: redis:7-alpine
    ports:
      - "6379:6379"
    volumes:
      - redis-data:/data
    restart: unless-stopped

volumes:
  app-data:
  app-logs:
  redis-data:
```

Building and Running

```
# Build the image
docker compose build

# 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:

```
Tina4::Router.get("/health") do |request, response|
  db = Tina4.database
  db_ok = false

  begin
    db.fetch_one("SELECT 1")
    db_ok = true
  rescue => e
    # Database is down
  end

  status = db_ok ? "ok" : "degraded"
  status_code = db_ok ? 200 : 503

  response.json({
    status: status,
    version: "1.0.0",
    framework: "tina4-ruby",
    database: db_ok ? "connected" : "disconnected",
    ruby_version: RUBY_VERSION,
    uptime_seconds: (Time.now - $start_time).to_i,
    memory_mb: ((`ps -o rss= -p #{Process.pid}`.to_i) / 1024.0).round(1),
    pid: Process.pid,
    timestamp: Time.now.utc.iso8601
  }, status_code)
end
```

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)

For graceful shutdown, create a `.broken` file in the project root to make the health check return a failure status. Wait for the load balancer to drain traffic, then restart the server.

---

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
```

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
```

Graceful Restart Pattern

```
# Signal the load balancer to stop routing traffic
touch .broken

# Wait for health checks to fail and traffic to drain
sleep 30

# Restart the application
sudo systemctl restart tina4-app

# Remove the signal file
rm .broken
```

---

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 deploy deploy
    postrotate
        kill -USR1 $(cat /app/tmp/pids/server.pid) 2>/dev/null || true
    endscript
}
```

This rotates logs daily, keeps 14 days of history, and compresses old logs. The `USR1` signal tells Puma to reopen its log files after rotation.

Docker Logging

Docker captures stdout/stderr automatically. Configure log rotation in the 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 Ruby)
- Request buffering
- Rate limiting
- WebSocket proxying

Create `/etc/nginx/sites-available/my-app`:

```
upstream tina4_app {
    server 127.0.0.1:7147;
}

server {
    listen 80;
    server_name yourdomain.com;
    return 301 https://$host$request_uri;
}

server {
    listen 443 ssl http2;
    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 Ruby)
    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://tina4_app;
        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_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header X-Forwarded-Proto $scheme;
        proxy_read_timeout 86400;
    }

    # Application
    location / {
        proxy_pass http://tina4_app;
        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. 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. Process Management with systemd

Create `/etc/systemd/system/tina4-app.service`:

```
[Unit]
```

```
Description=Tina4 Ruby Application
After=network.target

[Service]
Type=simple
User=deploy
WorkingDirectory=/app
ExecStart=/usr/local/bin/bundle exec puma -C config/puma.rb
Restart=always
RestartSec=5
Environment=RACK_ENV=production

[Install]
WantedBy=multi-user.target
```

Create `/etc/systemd/system/tina4-worker.service`:

```
[Unit]
Description=Tina4 Queue Worker
After=network.target tina4-app.service

[Service]
Type=simple
User=deploy
WorkingDirectory=/app
ExecStart=/usr/local/bin/tina4 queue work
Restart=always
RestartSec=5
Environment=RACK_ENV=production

[Install]
WantedBy=multi-user.target
```

```
sudo systemctl enable tina4-app tina4-worker
sudo systemctl start tina4-app tina4-worker
sudo systemctl status tina4-app
```

---

11. Scaling

A single server handles many applications. When traffic outgrows one server, you scale.

Multiple Workers

Puma runs multiple worker processes. Configure the count in `config/puma.rb`:

```
workers ENV.fetch("WEB_CONCURRENCY", 4)
threads 5, 5
```

Start with the number of CPU cores on your server. For I/O-heavy applications (database queries, external API calls), double 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 127.0.0.1:7147;
    server 127.0.0.1:7247;
```

```
    server 127.0.0.1:7347;
    server 127.0.0.1:7447;
}

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. Ruby's framework default is `7147`; use any free ports for the additional instances. Set `TINA4_OVERRIDE_CLIENT=true` so Puma can run without going through `tina4 serve`:

```
TINA4_OVERRIDE_CLIENT=true TINA4_PORT=7147 bundle exec puma -C config/puma.rb &
TINA4_OVERRIDE_CLIENT=true TINA4_PORT=7247 bundle exec puma -C config/puma.rb &
TINA4_OVERRIDE_CLIENT=true TINA4_PORT=7347 bundle exec puma -C config/puma.rb &
TINA4_OVERRIDE_CLIENT=true TINA4_PORT=7447 bundle exec puma -C config/puma.rb &
```

Docker Scaling

With Docker Compose, scale horizontally:

```
docker compose up -d --scale app=4
```

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.

---

12. 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

Prometheus Metrics

Enable metrics in `.env`:

Metrics are available at `/metrics` in Prometheus format:

```
curl http://localhost:7147/metrics
```

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.

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

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.

---

13. Exercise: Docker Deploy

Deploy a Tina4 Ruby application using Docker.

Requirements

- Create a `Dockerfile` that:
- Uses `ruby:3.3-slim` as the base image
- Installs dependencies with `bundle`
- Copies the application code
- Exposes port 7147
- Includes a health check
- Runs the app with Puma
- Create a `docker-compose.yml` that:
- Builds and runs the app
- Starts a Redis container for caching
- Starts a queue worker
- 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:7147/health

# Test the app
curl http://localhost:7147/api/products

# View logs
docker compose logs -f app

# Stop
docker compose down
```

---

14. Solution

The Dockerfile, docker-compose.yml, and Puma config are shown in sections 3 and 4. The health check route is shown in section 5. Combine them in your project, then:

```
docker compose up -d --build
```

```
[+] Building 18.4s
[+] Running 3/3
  Container redis     Started
  Container my-app    Started
  Container worker    Started
```

```
curl http://localhost:7147/health
```

```
{
  "status": "ok",
  "version": "1.0.0",
  "framework": "tina4-ruby",
  "database": "connected",
  "timestamp": "2026-03-22T14:30:00+00:00"
}
```

The app runs in production mode with Redis caching, persistent data volumes, a queue worker, automatic restarts, and health monitoring.

---

15. 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 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. Puma Workers Not Reconnecting to Database

Problem: Database errors after Puma forks worker processes.

Cause: Puma forks the master process to create workers. The forked processes inherit the parent's database connection, which is invalid after fork.

Fix: Use `on_worker_boot` in Puma config to reconnect:

```
on_worker_boot do
  Tina4::Database.reconnect
end
```

5. Built-in Server Used in Production

Problem: The server handles one request at a time and performance is terrible.

Cause: WEBrick (the built-in server) is single-threaded. It is for development only.

Fix: Use Puma in production: `bundle exec puma -C config/puma.rb`.

6. Static Files Slow

Problem: CSS, JS, and images load slowly in production.

Cause: Ruby serves static files. Every static file request goes through the Ruby process.

Fix: Serve static files from Nginx (see the Nginx config in section 8). Nginx serves static files from memory-mapped files, orders of magnitude faster than Ruby.

7. Memory Usage Grows Over Time

Problem: The container's memory usage climbs until it crashes with OOM (out of memory).

Cause: Memory leaks -- unclosed database connections, growing caches without TTL, accumulating data in global variables.

Fix: Set TTLs on all cache entries. Close database connections. Avoid storing request data in module-level variables. Use `docker stats` to monitor memory. Set memory limits in Docker Compose: `deploy: {resources: {limits: {memory: 512M}}}`. Set Puma's `worker_timeout` and consider periodic worker restarts.

8. SSL Certificate Not Renewing

Problem: Your HTTPS certificate expires and the site goes down.

Cause: The auto-renewal process 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:

```
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.

10. Queue Worker Stops Processing

Problem: The queue worker stops processing jobs without error messages.

Cause: The worker process crashed or was killed by OOM.

Fix: Use `Restart=always` in systemd. Monitor the worker with health checks. Set up alerting on dead letter queue growth.

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 via WebSocket
- 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 taskflow --lang ruby
cd taskflow
bundle install
```

Update `.env`:

```
TINA4_DEBUG=true
JWT_SECRET=taskflow-dev-secret-change-in-production
JWT_EXPIRY=86400
```

Create Migrations

Create `migrations/20260322150000_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/20260322150100_create_tasks_table.sql`:

```
-- UP
CREATE TABLE tasks (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    title TEXT NOT NULL,
    description TEXT DEFAULT '',
    status TEXT NOT NULL DEFAULT 'todo',
    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] 20260322150000_create_users_table.sql
  [UP] 20260322150100_create_tasks_table.sql

  2 migrations applied
```

Verify the database:

```
curl http://localhost:7147/health
```

```
{"status": "ok", "database": "connected"}
```

---

4. Step 2: ORM Models

Create `src/orm/user.rb`:

```
class User < Tina4::ORM
  integer_field :id, primary_key: true
```

```
  string_field :name
  string_field :email
  string_field :password_hash
  string_field :role, default: "user"
  string_field :created_at

  table_name "users"

  has_many :created_tasks, class_name: "Task", foreign_key: "created_by"
  has_many :assigned_tasks, class_name: "Task", foreign_key: "assigned_to"
end
```

Create `src/orm/task.rb`:

```
class Task < Tina4::ORM
  integer_field :id, primary_key: true
  string_field :title
  string_field :description, default: ""
  string_field :status, default: "todo"
  string_field :priority, default: "medium"
  integer_field :created_by
  integer_field :assigned_to
  string_field :due_date
  string_field :completed_at
  string_field :created_at
  string_field :updated_at

  table_name "tasks"

  belongs_to :creator, class_name: "User", foreign_key: "created_by"
  belongs_to :assignee, class_name: "User", foreign_key: "assigned_to"
end
```

---

5. Step 3: Authentication

Create `src/routes/middleware.rb`:

```
def auth_middleware(request, response, next_handler)
  auth_header = request.headers["Authorization"] || ""

  if auth_header.empty? || !auth_header.start_with?("Bearer ")
    return response.json({ error: "Authorization required" }, 401)
  end

  token = auth_header.sub("Bearer ", "")

  unless Tina4::Auth.valid_token(token)
    return response.json({ error: "Invalid or expired token" }, 401)
  end

  request.user = Tina4::Auth.get_payload(token)
  next_handler.call(request, response)
end
```

Create `src/routes/auth.rb`:

```
# @noauth
```

```
Tina4::Router.post("/api/auth/register") do |request, response|
  body = request.body

  errors = []
  errors << "Name is required" if body["name"].nil? || body["name"].empty?
  errors << "Email is required" if body["email"].nil? || body["email"].empty?
  errors << "Password must be at least 8 characters" if body["password"].nil? || body["password"

  return response.json({ errors: errors }, 400) unless errors.empty?

  db = Tina4.database
  existing = db.fetch_one("SELECT id FROM users WHERE email = ?", [body["email"]])
  return response.json({ error: "Email already registered" }, 409) unless existing.nil?

  hash = Tina4::Auth.hash_password(body["password"])
  db.execute("INSERT INTO users (name, email, password_hash) VALUES (?, ?, ?)",
    [body["name"], body["email"], hash])

  user = db.fetch_one("SELECT id, name, email, role FROM users WHERE id = last_insert_rowid()")
  response.json({ message: "Registration successful", user: user }, 201)
end

# @noauth
Tina4::Router.post("/api/auth/login") do |request, response|
  body = request.body

  db = Tina4.database
  user = db.fetch_one("SELECT * FROM users WHERE email = ?", [body["email"]])

  if user.nil? || !Tina4::Auth.check_password(body["password"] || "", user["password_hash"])
    return response.json({ error: "Invalid email or password" }, 401)
  end

  token = Tina4::Auth.get_token({
    user_id: user["id"], email: user["email"], name: user["name"], role: user["role"]
  })

  response.json({
    message: "Login successful",
    token: token,
    user: { id: user["id"], name: user["name"], email: user["email"], role: user["role"] }
  })
end

Tina4::Router.get("/api/profile", middleware: "auth_middleware") do |request, response|
  db = Tina4.database
  user = db.fetch_one("SELECT id, name, email, role, created_at FROM users WHERE id = ?",
    [request.user["user_id"]])
  response.json(user)
end
```

Test Registration and Login

```
# Start the server
tina4 serve

# Register a user
curl -X POST http://localhost:7147/api/auth/register \
  -H "Content-Type: application/json" \
  -d '{"name": "Alice Johnson", "email": "alice@example.com", "password": "securepass123"}'
```

```
{
  "message": "Registration successful",
  "user": {
    "id": 1,
    "name": "Alice Johnson",
    "email": "alice@example.com",
    "role": "user"
  }
}
```

```
# Login
curl -X POST http://localhost:7147/api/auth/login \
  -H "Content-Type: application/json" \
  -d '{"email": "alice@example.com", "password": "securepass123"}'
```

```
{
  "message": "Login successful",
  "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: Task CRUD

Create `src/routes/tasks.rb`:

```
Tina4::Router.group("/api/tasks", middleware: "auth_middleware") do

  # List tasks with filters
  Tina4::Router.get("") do |request, response|
    db = Tina4.database
    user_id = request.user["user_id"]

    status = request.params["status"]
    priority = request.params["priority"]
    assigned = request.params["assigned_to_me"]

    sql = "SELECT t.*, u1.name AS creator_name, u2.name AS assignee_name FROM tasks t LEFT JOIN
    conditions = []
    params = {}

    if assigned == "true"
      conditions << "t.assigned_to = :user_id"
      params[:user_id] = user_id
    end

    if status && !status.empty?
      conditions << "t.status = :status"
      params[:status] = status
    end

    if priority && !priority.empty?
```

```
    conditions << "t.priority = :priority"
    params[:priority] = priority
  end

  sql += " WHERE #{conditions.join(' AND ')}" unless conditions.empty?
  sql += " ORDER BY t.created_at DESC"

  tasks = db.fetch(sql, params)
  response.json({ tasks: tasks, count: tasks.length })
end

# Get single task
Tina4::Router.get("/{id:int}") do |request, response|
  db = Tina4.database
  id = request.params["id"]

  task = db.fetch_one("SELECT t.*, u1.name AS creator_name, u2.name AS assignee_name FROM task

  if task.nil?
    return response.json({ error: "Task not found" }, 404)
  end

  response.json(task)
end

# Create task
Tina4::Router.post("") do |request, response|
  body = request.body
  user_id = request.user["user_id"]

  if body["title"].nil? || body["title"].empty?
    return response.json({ error: "Title is required" }, 400)
  end

  db = Tina4.database
  db.execute(
    "INSERT INTO tasks (title, description, status, priority, created_by, assigned_to, due_dat
    [body["title"], body["description"] || "", body["status"] || "todo", body["priority"] || "
  )

  task = db.fetch_one("SELECT * FROM tasks WHERE id = last_insert_rowid()")

  # Queue notification if assigned to someone else
  if body["assigned_to"] && body["assigned_to"].to_i != user_id
    assignee = db.fetch_one("SELECT name, email FROM users WHERE id = ?", [body["assigned_to"]
    if assignee
      Tina4::Queue.produce("send-email", {
        to: assignee["email"],
        subject: "New task assigned: #{body['title']}",
        template: "emails/task-assigned.html",
        data: { task_title: body["title"], assignee_name: assignee["name"], assigner_name: req
      })
    end
  end

  # Push WebSocket update
  Tina4::WebSocket.broadcast("/ws/tasks", {
    type: "task_update",
    action: "created",
```

```
    task: task
  }.to_json)

  # Invalidate cache
  Tina4.cache_delete("dashboard:stats:#{user_id}")

  response.json(task, 201)
end

# Update task
Tina4::Router.put("/{id:int}") do |request, response|
  db = Tina4.database
  id = request.params["id"]
  body = request.body

  existing = db.fetch_one("SELECT * FROM tasks WHERE id = ?", [id])
  return response.json({ error: "Task not found" }, 404) if existing.nil?

  completed_at = existing["completed_at"]
  if body["status"] == "done" && existing["status"] != "done"
    completed_at = Time.now.strftime("%Y-%m-%d %H:%M:%S")
  elsif body["status"] && body["status"] != "done"
    completed_at = nil
  end

  db.execute(
    "UPDATE tasks SET title = ?, description = ?, status = ?, priority = ?, assigned_to = ?, d
    [
      body["title"] || existing["title"],
      body["description"] || existing["description"],
      body["status"] || existing["status"],
      body["priority"] || existing["priority"],
      body.key?("assigned_to") ? body["assigned_to"] : existing["assigned_to"],
      body["due_date"] || existing["due_date"],
      completed_at,
      id
    ]
  )

  # Push WebSocket update
  task = db.fetch_one("SELECT * FROM tasks WHERE id = ?", [id])
  Tina4::WebSocket.broadcast("/ws/tasks", {
    type: "task_update",
    action: "updated",
    task: task
  }.to_json)

  # Invalidate cache
  Tina4.cache_delete("dashboard:stats:#{request.user['user_id']}")

  response.json(task)
end

# Delete task
Tina4::Router.delete("/{id:int}") do |request, response|
  db = Tina4.database
  id = request.params["id"]

  existing = db.fetch_one("SELECT * FROM tasks WHERE id = ?", [id])
```

```
    return response.json({ error: "Task not found" }, 404) if existing.nil?

    db.execute("DELETE FROM tasks WHERE id = ?", [id])

    Tina4::WebSocket.broadcast("/ws/tasks", {
      type: "task_update",
      action: "deleted",
      task: { id: id }
    }.to_json)

    Tina4.cache_delete("dashboard:stats:#{request.user['user_id']}")

    response.json(nil, 204)
  end

end
```

Test the Task API

```
# Set your token from the login step
TOKEN="eyJhbGciOiJIUzI1NiIs..."

# Create a task
curl -X POST http://localhost:7147/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": "todo",
  "created_by": 1,
  "created_at": "2026-03-22 10:00:00"
}
```

```
# Create more tasks
curl -X POST http://localhost:7147/api/tasks \
  -H "Content-Type: application/json" \
  -H "Authorization: Bearer $TOKEN" \
  -d '{"title": "Build API endpoints", "priority": "high"}'

curl -X POST http://localhost:7147/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:7147/api/tasks \
  -H "Authorization: Bearer $TOKEN"

# Update a task status
curl -X PUT http://localhost:7147/api/tasks/1 \
  -H "Content-Type: application/json" \
  -H "Authorization: Bearer $TOKEN" \
  -d '{"status": "done"}'

# Delete a task
curl -X DELETE http://localhost:7147/api/tasks/3 \
```

```
  -H "Authorization: Bearer $TOKEN"
```

Every endpoint responds. Create. Read. Update. Delete. The CRUD cycle works.

---

7. Step 5: Dashboard Stats with Caching

Create `src/routes/dashboard.rb`:

```
Tina4::Router.get("/api/dashboard/stats", middleware: "auth_middleware") do |request, response|
  user_id = request.user["user_id"]

  cache_key = "dashboard:stats:#{user_id}"
  stats = Tina4.cache_get(cache_key)

  if stats.nil?
    db = Tina4.database

    total = db.fetch_one("SELECT COUNT(*) AS count FROM tasks WHERE assigned_to = ? OR created_b
    todo = db.fetch_one("SELECT COUNT(*) AS count FROM tasks WHERE (assigned_to = ? OR created_b
    in_progress = db.fetch_one("SELECT COUNT(*) AS count FROM tasks WHERE (assigned_to = ? OR cr
    done = db.fetch_one("SELECT COUNT(*) AS count FROM tasks WHERE (assigned_to = ? OR created_b
    overdue = db.fetch_one("SELECT COUNT(*) AS count FROM tasks WHERE (assigned_to = ? OR create

    recent = db.fetch(
      "SELECT t.*, u.name AS assignee_name FROM tasks t LEFT JOIN users u ON t.assigned_to = u.i
    )

    stats = {
      total_tasks: total["count"],
      todo: todo["count"],
      in_progress: in_progress["count"],
      done: done["count"],
      overdue: overdue["count"],
      recent_tasks: recent
    }

    Tina4.cache_set(cache_key, stats, 30)
  end

  response.json(stats)
end

Tina4::Router.get("/admin") do |request, response|
  response.render("dashboard.html", {
    title: "TaskFlow Dashboard"
  })
end
```

Verify the stats endpoint:

```
curl http://localhost:7147/api/dashboard/stats \
  -H "Authorization: Bearer $TOKEN"

{
  "total_tasks": 2,
  "todo": 1,
  "in_progress": 0,
```

```
  "done": 1,
  "overdue": 0,
  "recent_tasks": [...]
}
```

Hit it again within 30 seconds. The second response comes from cache -- no database queries.

---

8. Step 6: WebSocket Real-Time Updates

Create `src/routes/task_ws.rb`:

```
Tina4::WebSocket.on("/ws/tasks") do |connection, event, data|
  case event
  when "open"
    connection.send({ type: "connected", message: "Listening for task updates" }.to_json)

  when "message"
    message = JSON.parse(data)
    if message["type"] == "ping"
      connection.send({ type: "pong" }.to_json)
    end

  when "close"
    # Connection closed -- cleanup if needed
  end
end
```

Any user creates, updates, or deletes a task. All connected dashboard users see the change. The `Tina4::WebSocket.broadcast` calls in the task routes (Step 4) push updates to every client connected to `/ws/tasks`.

---

9. Step 7: Email Notifications

Create the queue consumer for task assignment notifications:

```
Tina4::Queue.consume("send-email") do |job|
  mail = Tina4::Messenger.new
  mail.to = job.payload["to"]
  mail.subject = job.payload["subject"]

  if job.payload["template"]
    mail.html_template = job.payload["template"]
    mail.template_data = job.payload["data"] || {}
  else
    mail.body = job.payload["body"] || ""
  end

  mail.send
  true
end
```

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>{{ assigner_name }}</strong> assigned you a new task:</p>

        <div class="card">
            <div class="card-body">
                <h3>{{ task_title }}</h3>
            </div>
        </div>

        <p><a href="http://localhost:7147/admin">View Dashboard</a></p>
    </div>
</body>
</html>
```

Start the queue worker in a separate terminal:

```
tina4 queue work
```

---

10. Step 8: 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">Todo</h6><h2 id="stat-todo">--</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">Done</h6><h2 id="stat-done">--</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>

<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" onclick="showNewTaskForm()">
                    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-todo").textContent = data.todo;
        document.getElementById("stat-progress").textContent = data.in_progress;
        document.getElementById("stat-done").textContent = data.done;
        document.getElementById("stat-overdue").textContent = data.overdue;

        var tbody = document.getElementById("task-list");
        tbody.innerHTML = "";
        (data.recent_tasks || []).forEach(function (task) {
            var tr = document.createElement("tr");
            var badgeColor = {todo: "secondary", in_progress: "warning",
                              done: "success"};
            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 %}
```

Open `http://localhost:7147/admin` in your browser to see the full dashboard with stats, task list, and live update panel.

---

11. Step 9: Tests

Create `tests/taskflow_spec.rb`:

```
require "tina4"

RSpec.describe "TaskFlow" do
  let(:client) { Tina4::TestClient.new }
  let(:token) do
    email = "test#{rand(100000)}@example.com"
    client.post("/api/auth/register", {
      name: "Test User", email: email, password: "password123"
    })
    result = client.post("/api/auth/login", {
      email: email, password: "password123"
    })
    result.json["token"]
  end

  it "registers a user" do
    result = client.post("/api/auth/register", {
      name: "Alice", email: "alice#{rand(100000)}@example.com", password: "securePass123"
    })
    expect(result.status).to eq(201)
  end

  it "rejects duplicate email" do
    email = "dup#{rand(100000)}@example.com"
    client.post("/api/auth/register", {
      name: "First", email: email, password: "password123"
    })
    result = client.post("/api/auth/register", {
      name: "Second", email: email, password: "password123"
    })
    expect(result.status).to eq(409)
  end

  it "creates a task" do
    result = client.post("/api/tasks", { title: "Test Task" },
```

```
      headers: { "Authorization" => "Bearer #{token}" })
    expect(result.status).to eq(201)
    expect(result.json["title"]).to eq("Test Task")
    expect(result.json["status"]).to eq("todo")
  end

  it "lists tasks" do
    client.post("/api/tasks", { title: "Task 1" }, headers: { "Authorization" => "Bearer #{token
    client.post("/api/tasks", { title: "Task 2" }, headers: { "Authorization" => "Bearer #{token

    result = client.get("/api/tasks", headers: { "Authorization" => "Bearer #{token}" })
    expect(result.status).to eq(200)
    expect(result.json["tasks"]).to be_an(Array)
  end

  it "updates a task status" do
    created = client.post("/api/tasks", { title: "Update Me" },
      headers: { "Authorization" => "Bearer #{token}" })
    id = created.json["id"]

    result = client.put("/api/tasks/#{id}", { status: "done" },
      headers: { "Authorization" => "Bearer #{token}" })
    expect(result.status).to eq(200)
    expect(result.json["status"]).to eq("done")
    expect(result.json["completed_at"]).not_to be_nil
  end

  it "deletes a task" do
    created = client.post("/api/tasks", { title: "Delete Me" },
      headers: { "Authorization" => "Bearer #{token}" })
    id = created.json["id"]

    result = client.delete("/api/tasks/#{id}",
      headers: { "Authorization" => "Bearer #{token}" })
    expect(result.status).to eq(204)
  end

  it "returns dashboard stats" do
    client.post("/api/tasks", { title: "Stats Task" },
      headers: { "Authorization" => "Bearer #{token}" })

    result = client.get("/api/dashboard/stats",
      headers: { "Authorization" => "Bearer #{token}" })
    expect(result.status).to eq(200)
    expect(result.json["total_tasks"]).to be >= 1
  end

  it "requires authentication for tasks" do
    result = client.get("/api/tasks")
    expect(result.status).to eq(401)
  end
end
```

Run the tests:

```
tina4 test

Running tests...

  TaskFlow
```

```
    registers a user
    rejects duplicate email
    creates a task
    lists tasks
    updates a task status
    deletes a task
    returns dashboard stats
    requires authentication for tasks

  8 examples, 0 failures (0.84s)
```

All green. The application works.

---

12. Step 10: Docker Deployment

Use the Dockerfile and docker-compose.yml from Chapter 33. Create `.env.production`:

```
TINA4_DEBUG=false
TINA4_LOG_LEVEL=WARNING
JWT_SECRET=your-production-secret-at-least-32-characters
TINA4_DATABASE_URL=sqlite:///data/app.db
TINA4_CACHE_BACKEND=redis
TINA4_CACHE_URL=redis
```

Deploy:

```
docker compose up -d --build
```

Verify:

```
curl http://localhost:7147/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.

---

13. The Complete Project Structure

```
taskflow/
■■■ .env
■■■ .env.example
■■■ .gitignore
■■■ Gemfile
■■■ Gemfile.lock
■■■ app.rb
■■■ Dockerfile
■■■ docker-compose.yml
■■■ config/
■   ■■■ puma.rb
■■■ src/
■   ■■■ routes/
■   ■   ■■■ auth.rb               # Registration, login
■   ■   ■■■ tasks.rb              # Task CRUD
■   ■   ■■■ dashboard.rb          # Dashboard stats + page
■   ■   ■■■ middleware.rb         # Auth middleware
■   ■   ■■■ task_ws.rb            # WebSocket event handlers
■   ■■■ orm/
■   ■   ■■■ user.rb               # User model with relationships
■   ■   ■■■ task.rb               # Task model with relationships
■   ■■■ migrations/
■   ■   ■■■ 20260322150000_create_users_table.sql
■   ■   ■■■ 20260322150100_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/
    ■■■ taskflow_spec.rb
```

Every file has a purpose. Every directory follows the convention. A new developer looks at this structure and knows where to find things.

---

14. What You Built

This chapter used every major concept from the book:

Feature	Chapter
Route definitions	Chapter 2
Request/response handling	Chapter 3
Frond templates	Chapter 4
Database queries	Chapter 5
ORM models (User, Task)	Chapter 6
JWT authentication	Chapter 8
Auth middleware	Chapter 10
Queue-based email	Chapter 12
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

---

15. 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.

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.

---

16. Gotchas

1. Circular Dependencies Between Models

Problem: User loads tasks, tasks load users, infinite loop.

Fix: Use lazy loading. Do not eager-load relationships that create cycles.

2. Cache Not Invalidated on Related Model Changes

Problem: Dashboard stats are stale after task updates.

Fix: Invalidate cache in every write operation: `Tina4.cache_delete("dashboard:stats:#{user_id}")`.

3. Queue Worker Not Processing Emails

Problem: Task assignment emails are queued but never sent.

Fix: Run `tina4 queue work` in a separate terminal or as a systemd service.

4. Token Expired During Long Sessions

Problem: Users get logged out while actively using the dashboard.

Fix: Set a reasonable JWT expiry (`86400` for 24 hours) and implement token refresh.

5. WebSocket Notifications Not Received

Problem: Real-time dashboard updates do not work.

Fix: Ensure the WebSocket endpoint is configured, the server is running, and the client JavaScript connects to the correct URL. Check the browser console for connection errors.

6. Database Locked Under Load

Problem: SQLite returns "database is locked" errors with concurrent requests.

Fix: For production with concurrent users, switch to PostgreSQL. SQLite handles one writer at a time.

7. Missing Migration Before Deployment

Problem: The app crashes in production because a table does not exist.

Fix: Always run `tina4 migrate` as part of your deployment pipeline. Add it to your Dockerfile or CI/CD script.

---

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`.

No separate ORM gem. No template engine gem. 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, Python, 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 `self.auto_map = false` on your model class to restore the old behaviour.

• **feat:** `to_h(case:)` parameter — pass `case: 'camel'` to get camelCase keys (for JSON APIs) or `case: 'snake'` (default) for snake_case keys matching DB columns. All aliases (`to_dict`, `to_hash`, `to_assoc`, `to_object`) support the parameter.

• **feat:** Frond `replace` filter now accepts Hash args — `{{ v|replace({"T": " ", "-": "/"}) }}` for multiple substitutions in one call.

• **tests:** 6 new parity tests covering `to_h(case:)`, `auto_map` default, `replace` filter (Hash + positional), and `ServiceRunner` registration. 2,519 tests passing.

• **parity:** All features shipped identically across Python, PHP, Ruby, Node.js.

v3.10.97 (2026-04-11)

• **fix:** frond.form.submit redirect handling — XHR follows 3xx redirects transparently; fixed by detecting `xhr.responseURL` mismatch and navigating instead.

• **dep:** Updated frond.min.js to v2.1.2.

• **parity:** All 4 frameworks bumped to 3.10.97.

v3.10.93 (2026-04-11)

• **fix:** Frond bracket depth tracking in `find_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 simple variable lookup, enabling `arr[loop.index0 % 2]`.

• **fix:** Frond slice with variable bounds — `items[start:end]` evaluates bounds through `eval_expr()`.

• **docs:** Developer skills updated — Metrics Dashboard guidance, Frond Template Parity rules, `@noauth` security warnings.

• **parity:** All Frond fixes applied identically across Python, PHP, Ruby, Node.js. 2,513 tests passing.

v3.10.92 (2026-04-10)

• **breaking:** Rename `ErrorOverlay` methods — `render` → `render_error_overlay`, `render_production` → `render_production_error`, `debug_mode?` → `is_debug_mode`.

• **feat:** Add `Server.handle(env)` for cross-framework parity.

• **breaking:** Rename `WebSocketBackplane.create` → `WebSocketBackplane.create_backplane`.

• **feat:** Add `ScssCompiler.compile`, `add_import_path`, `set_variable` methods.

• **feat:** Add `DevAdmin.register` method.

• **parity:** 44/44 cross-framework features green. 2,487 tests passing.

v3.10.91 (2026-04-10)

• **feat:** Add parity methods — `Response.send` params, `Middleware.check`/`is_preflight`, AI/Log aliases, MCP optional router.

• **breaking:** Rename `from()` → `from_table()`, `error_envelope` → `error_response`, remove aliases.

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>value.inspect</pre>` 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 `Tina4::Frond.render_dump` helper and produce identical output (`<pre>value.inspect</pre>` HTML-escaped).

• **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 `SafeString`. 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.

• **test:** 3 new `spec/frond_spec.rb` examples covering debug-mode output, production silencing, function/filter parity, and function-form production silencing.

v3.10.86 (2026-04-09)

• **feat:** `foreign_key_field` DSL auto-wires both sides of a foreign key relationship. Declaring `foreign_key_field :user_id, references: User` registers the integer column, calls `belongs_to :user` on the declaring class, and calls `has_many :posts` on the referenced class. Supports `related_name:` for custom has-many names and deferred wiring via a module-level registry so the referenced class can be defined either before or after the declaring one.

• **feat:** Cross-framework parity — same FK auto-wiring semantics now available in Python (`ForeignKeyField`), PHP (`$foreignKeys`), and Node.js (`type: "foreignKey"`)

• **docs:** Chapter 6 (ORM) updated with a new "foreign*key*field — Auto-Wired Relationships" section

v3.10.85 (2026-04-09)

• Version bump for parity with Python and PHP releases

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 after `valid_token?` 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)

• **add:** Headless routing auth payload integration tests to prevent regression

v3.10.83 (2026-04-08)

• **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

• **feat:** Auth cleanup — canonical `getToken`/`validToken` methods

• Full parity across Python, PHP, Ruby, Node.js

---

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

Tina4 Ruby follows semantic versioning. The major version (3) marks the initial Ruby launch — Tina4 Ruby is new in the v3 line, alongside Tina4 for Node.js. Minor versions (3.1, 3.2, etc.) introduce features and non-breaking API additions. Patch versions carry bug fixes and small improvements.

This chapter covers every v3 release from the initial launch through the current stable line. Each section groups releases by minor version, highlights the changes that affect your code, and shows migration steps for anything that breaks.

---

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, arrays not tuples, toDict/toAssoc, scope registers method, where()/all() on Node, count() on PHP

• **Auth:** expires*in minutes, PBKDF2 260k, env TINA4*SECRET fallback, API key fallback

- **Session:** dual-mode flash(), get_flash, cookieHeader, getSessionId
- **Database:** execute() bool/DatabaseResult, get/*last*id/get_error, getColumns, cacheStats
- **Request/Response:** files dict, query, cookies, contentType, xml(), callable
- **Queue:** consume() poll_interval
- **WebSocket:** event naming, connection properties
- **GraphQL:** schema_sdl() + introspect() on all 4
- **Events:** emitAsync() on all 4
- **i18n:** zero-dep YAML support

v3.10.67 (2026-04-03)

- **load() is now an instance method** — `model.load(sql, params)` calls select_one internally, populates the instance, returns `true`/`false`. Use `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
- **File upload docs** — `request.files` format documented in CLAUDE.md

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** — already existed, now matches Python/PHP/Node API
- **tina4 env** — interactive environment configuration
- **Brand update** — "TINA4 — The Intelligent Native Application 4ramework"
- **Imperative relationships** — query*has*one/many/belongs_to for ad-hoc queries
- **Port kill-and-take-over** — default port always reclaimed
- **MongoDB adapter** (mongo gem), **ODBC adapter** (ruby-odbc gem)
- **Pagination standardized** — limit/offset primary, merged dual-key response
- **CORS fix** — returns empty string when origin not allowed

---

v3.10.57 (2026-04-02)

- **MongoDB adapter** — `Database.new("mongodb://host:port/db")`, requires `gem install mongo`
- **ODBC adapter** — `Database.new("odbc:///DSN=MyDSN")`, requires `gem install ruby-odbc`
- **Imperative relationships** — `query_has_one`/`query_has_many`/`query_belongs_to`
- **Pagination standardized** — limit/offset primary, merged dual-key to_paginate response
- **Test port at +1000** — user testing port (e.g. 8147) stable, no hot-reload
- **CORS fix** — returns empty string when origin not allowed
- **ORM TINA4*DATABASE*URL discovery** — auto-connect from env
- **108 features at 100% parity**, 2,333 tests

---

v3.10.54 (2026-04-02)

- **Auto AI dev port** — second WEBrick on port+1 with no-reload when TINA4_DEBUG=true
- **TINA4*NO*RELOAD** env var + --no-reload CLI flag
- **CORS fix** — returns empty string when origin not allowed (not *)
- **ORM TINA4*DATABASE*URL discovery** — auto-connect from env
- **QueryBuilder docs** — added to ORM chapter

---

v3.10.48 — April 2, 2026

Bug Fixes

Puma requires `--production` flag — Puma no longer auto-selected when `TINA4_DEBUG=false`. Use `tina4ruby serve --production` to enable Puma. Added FakeData (46), Gallery (16), and DevReload (37) tests.

---

v3.10.46 — April 1, 2026

Test Coverage

344 new tests added across cache (56), ORM (19), Frond (28), database drivers (85), auth (21), SCSS (10), dotenv (30), queue backends (10), migration (10), session handlers (11), router (14), log (13), CSRF middleware (17). Fixed session handler DB key bug (symbol vs string). Ruby now at 2,274 tests with full parity across all 49 core areas.

---

v3.10.45 — April 1, 2026

Notes

Version bump for parity with PHP CLI serve fix. No Ruby-specific changes.

---

v3.10.44 — April 1, 2026

New Features

Database tab redesign — Split-screen layout with tables navigation on the left and query editor + results on the right. Click-to-select table highlighting.

Copy CSV / Copy JSON — Copy query results to clipboard in CSV or JSON format.

Paste data — Modal for pasting JSON arrays or CSV/tab-separated data. Auto-generates INSERT statements targeting the selected table, or prompts for a new table name with CREATE TABLE generation. SQL input passes through unchanged.

Multi-statement execution — Query runner handles batched SQL statements in a transaction.

Database badge on load — Table count shows immediately without clicking the Database tab.

Star wiggle animation — Empty star (■) on the landing page with delayed wiggle animation at random intervals.

Bug Fixes

Default port — Ruby default port set to 7147 (PHP=7145, Python=7146, Ruby=7147, Node=7148).

SQLite LIMIT fix — Prevents double-LIMIT errors in the database browser.

browseTable quote escaping — Fixed table name click handlers.

ORM table name pluralization — Fixed default table name resolution. Table names are now pluralized by default (adding "s" suffix), only skipping when `TINA4_ORM_PLURAL_TABLE_NAMES` is explicitly set to false.

QueryBuilder closed-connection detection — `ensure_db!` now checks if the resolved database connection is still open, raising a proper error instead of crashing with `ArgumentError: prepare called on a closed database`.

Metrics directory validation — `quick_metrics` and `full_analysis` now check directory existence before `_resolve_root` fallback, so missing-directory errors are raised correctly.

Test Coverage

88 new tests added (DevMailbox 40, Static files 18, CLI scaffolding 30), plus 13 v3.10.44 feature specs and 60 pre-existing ORM/metrics bug fixes. 1,913 tests passing, 0 failures.

---

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 RubyGems. The overlay now shows a purple "ahead of RubyGems" message. Also added a breaking changes warning (red banner with changelog link) when a major or minor version update is available.

---

v3.10.39 — April 1, 2026

New Features

`Container.singleton(name, &block)` — Register a memoized factory. The block is called once on first `resolve()` and the same instance is returned on all subsequent calls. `register()` with a block is now always transient (new instance per call), matching Python's behavior.

```
Tina4::Container.singleton(:db) { Tina4::Database.new(ENV["TINA4_DATABASE_URL"]) }
db1 = Tina4::Container.resolve(:db)  # creates instance
db2 = Tina4::Container.resolve(:db)  # same instance
```

`Router.match(method, path)` — primary route lookup (replaces `find_route`; consistent with Python, PHP, Node.js). `Router.add(method, path, handler)` — primary imperative registration (replaces `add_route`; all convenience methods delegate to this).

`Router.get_routes` **and** `Router.list_routes` — explicit listing methods (remove ambiguous `routes` alias).

AI installer — `ai_spec.rb` and smoke tests updated to reflect the menu-based API (`installed?`, `install_selected`, `install_all`, `generate_context`).

---

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 appear as animated bubbles sized by LOC and colored by maintainability index. Click any bubble to drill down into per-function cyclomatic complexity.

The metrics engine uses `Ripper` (Ruby stdlib) for zero-dependency static analysis covering cyclomatic complexity, Halstead volume, maintainability index, coupling, and violation detection. File analysis is sorted worst-first. Results are cached for 60 seconds.

AI Context Installer

`tina4ruby ai` now presents a simple numbered menu instead of auto-detection. Select tools by number, comma-separated or `all`. Already-installed tools show green. Generated context includes the full skills table.

Dashboard Improvements

Full-width layout, sticky header/tabs, full-screen overlay.

Cleanup

Removed `demo/` directory. Removed old `plan/` spec documents, replaced with `PARITY.md` and `TESTS.md`. Central parity matrix added to tina4-book.

---

v3.10.x -- Previous Releases

Released: March 28 -- 30, 2026

The v3.10 line is the most active release series. It delivered Auto-CRUD, ORM transaction safety, Frond template engine hardening, and full cross-language parity with the Python, PHP, and Node.js implementations.

v3.10.29 -- Version Parity (March 30)

Version parity release. All four Tina4 frameworks now share the same version number and feature set.

v3.10.27 -- Frond Macro HTML Escaping Fix (March 30)

Bug fix: Macro output was HTML-escaped when used inside `{{ }}` expressions. Characters like `<`, `>`, and `"` rendered as `<`, `>`, `"` instead of raw HTML. Nested macro calls double-escaped.

```
# BEFORE (broken): macro output escaped
```

```
## 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

# Template: {{ my_macro() }}

## 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

# Rendered: &lt;div class=&quot;card&quot;&gt;...&lt;/div&gt;

## 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


# AFTER (fixed): macro output treated as safe HTML

## 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

# Template: {{ my_macro() }}

## 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

# Rendered: <div class="card">...</div>

## 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
```

v3.10.25 -- ORM Transaction Fix (March 30)

Bug fix: ORM `save` and `delete` called `commit` without an active transaction on SQLite. This raised `cannot commit -- no transaction is active` errors.

```
# BEFORE (broken):
```

```
## 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.new(name: "Alice")
user.save  # => RuntimeError: cannot commit

# AFTER (fixed): save/delete wrap operations in a transaction block

## 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.new(name: "Alice")
user.save  # => works on all database engines
```

v3.10.22 -- Unique Form Tokens (March 30)

Form tokens now include a nonce in the JWT payload. Each token is unique per form render, which prevents replay attacks.

```
# In your Frond template:

## 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

<input type="hidden" name="formToken" value="{{ formTokenValue() }}">
```

v3.10.18 -- Frond Ternary Parser Fix (March 29)

Bug fix: The Frond template ternary/inline-if parser failed on quoted strings containing special characters.

```
# BEFORE (broken):

## 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

# {{ status == "active" ? "Yes" : "No" }}  =>  parse error

## 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


# AFTER (fixed):

## 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

# {{ status == "active" ? "Yes" : "No" }}  =>  "Yes"

## 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
```

v3.10.16 -- Template Filters: to*json*, *js*escape (March 28)

Three new Frond template filters for working with data in JavaScript contexts.

```
# Convert a Ruby hash to JSON inside a template:

## 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

<script>
  const data = {{ user|to_json }};
  const name = "{{ user.name|js_escape }}";
</script>
```

v3.10.15 -- Replace Filter Backslash Fix (March 28)

Bug fix: The `|replace` filter mishandled backslash characters in replacement strings.

```
{# Before (broken) — backslash produced corrupted output #}
{{ "hello\\world"|replace("\\\\", "/") }}
{# rendered: helo/world (ate a character) #}

{# After (fixed) — backslash escaping works correctly #}
{{ "hello\\world"|replace("\\\\", "/") }}
{# renders: hello/world #}
```

v3.10.14 -- get*next*id() (March 28)

Pre-generate the next primary key before inserting a record. The method detects your database engine and uses the correct sequence or auto-increment mechanism.

```
next_id = User.get_next_id
user = User.new(id: next_id, name: "Alice")
user.save
```

v3.10.13 -- ORM Auto-Commit on Write (March 28)

Write operations (`save`, `delete`) now auto-commit by default. No more forgotten `commit` calls leaving data uncommitted.

v3.10.12 -- Session GC and NATS Backplane (March 28)

- Session garbage collection runs on a configurable interval
- NATS added as a WebSocket backplane option alongside Redis

v3.10.11 -- Frond Variable Key Access (March 28)

Bug fix: Accessing a hash value with a variable key (`dict[variable_key]`) failed in Frond templates.

```
# BEFORE (broken):

## 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

# {% set key = "name" %}

## 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[key] }}  =>  empty

## 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


# AFTER (fixed):

## 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

# {% set key = "name" %}

## 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[key] }}  =>  "Alice"

## 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
```

v3.10.10 -- Firebird Migration Runner Fixes (March 28)

Firebird migrations now use generators and `VARCHAR` instead of `AUTOINCREMENT` and `TEXT`. The migration tracking table uses a proper Firebird sequence (`GEN_TINA4_MIGRATION_ID`).

v3.10.6 -- WSDL/SOAP Rewrite (March 28)

Complete rewrite of the WSDL/SOAP module. Frond templates now support dotted function names in expressions.

v3.10.5 -- Frond Quote-Aware Operator Matching (March 28)

Bug fix: Operators inside quoted strings were incorrectly parsed as expression operators. The Frond engine now respects quote boundaries.

v3.10.4 -- Auto-CRUD REST Endpoint Generator (March 28)

Generate a complete CRUD interface from a single method call. The generator creates searchable, sortable, paginated HTML tables with create/edit/delete modals, plus REST API routes for POST, PUT, and DELETE.

```
Tina4::Router.get("/admin/users") do |request, response|
  Tina4::CRUD.to_crud(request, model: User, fields: [:name, :email, :role])
end
```

v3.10.2 -- Frond Hash Method Calls (March 28)

Frond templates can now call methods on Hash and object values inside expressions.

v3.10.1 -- autoMap and Case Conversion (March 28)

- `auto_map` class attribute added to ORM for cross-language API parity (no-op in Ruby since `snake_case` is native)
- `Tina4.snake_to_camel("my_field")` returns `"myField"`
- `Tina4.camel_to_snake("myField")` returns `"my_field"`

v3.10.0 -- Optimized For-Loops (March 28)

The Frond template engine rewrote its for-loop renderer. Templates with large iteration counts render faster.

---

v3.9.x

Released: March 26 -- 27, 2026

v3.9.0 -- QueryBuilder, Sessions, Path Injection (March 26)

Three features arrived together.

QueryBuilder. A fluent SQL builder that integrates with the ORM.

```
# 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:

## 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

rows = Tina4::QueryBuilder.from("users")
  .where("active = ?", [true])
  .select("name", "email")
  .get
```

The builder supports `where`, `or_where`, `join`, `left_join`, `group_by`, `having`, `order_by`, `limit`, `first`, `count`, `exists`, and `to_sql`.

Path parameter injection. Route handlers receive path parameters as named arguments.

```
Tina4::Router.get("/users/{id:int}") do |request, response, id|
  user = User.find(id)
  response.json(user.to_hash)
end
```

Auto-start sessions. Every route handler has access to `request.session` with zero configuration. The session API includes `get`, `set`, `delete`, `has`, `clear`, `destroy`, `save`, `regenerate`, `flash`, `get_flash`, and `all`.

v3.9.1 -- Security Defaults (March 27)

Breaking change: POST, PUT, PATCH, and DELETE routes now require authentication by default.

```
# BEFORE (v3.8.x): all routes open
```

```
## 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::Router.post("/api/users") do |request, response|
  # anyone could call this
end

# AFTER (v3.9.1): unauthenticated requests get 401

## 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

# To allow public access, add .public:

## 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::Router.post("/api/users").public do |request, response|
  # open to all
end
```

This release also added:

- CSRF middleware with session-bound form tokens
- Standardized environment variables for CORS headers, session TTL, token limits
- Queue parity: `push` with priority/delay, `size(status)`, `message.retry`

v3.9.2 -- NoSQL QueryBuilder, WebSocket Backplane (March 27)

- QueryBuilder works with MongoDB
- WebSocket backplane support for multi-process deployments
- `SameSite=Lax` set as the default cookie policy

---

v3.8.x

Released: March 25 -- 26, 2026

v3.8.0 -- Base64 Filters, Template Cache (March 25)

- `base64encode` and `base64decode` filters in Frond templates
- Production template cache: single filesystem scan at startup, O(1) lookups after

v3.8.1 -- Security Headers Middleware (March 25)

A built-in middleware that sets `X-Frame-Options`, `Strict-Transport-Security`, `Content-Security-Policy`, and `X-Content-Type-Options` on every response.

```
# In your .env:

## 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_MAX_UPLOAD_SIZE=10485760  # 10 MB (default)
```

Upload size limits and input validation also landed in this release.

v3.8.2 -- Connection Pooling (March 26)

Database connections now pool. Pass `pool: N` to the constructor for round-robin, mutex-protected pooling.

```
db = Tina4::Database.new("sqlite://data.db", pool: 5)
```

v3.8.3 -- Claude Code Commands (March 26)

Seventeen `.claude/commands/` slash commands shipped for AI-assisted development.

v3.8.7 -- Benchmark and Stability (March 26)

- Keyword argument fix for `run!()`: `port:`, `host:`, and `debug:` no longer crash the environment loader
- Updated benchmarks against Roda, Sinatra, and Rails

---

v3.7.x

Released: March 25, 2026

v3.7.0 -- Template Auto-Serve, Firebird Migrations (March 25)

The framework serves `index.html` or `index.twig` from `src/templates/` at `/` without a route definition. User-registered `GET /` routes take priority.

Firebird migrations now check `RDB$RELATION_FIELDS` before executing `ALTER TABLE ADD`. Columns that exist are skipped.

v3.7.1 -- Full Template Auto-Serve (March 25)

Any `.twig` or `.html` file in `src/templates/` is now browsable by URL path. `/hello` serves `src/templates/hello.twig`. Production mode caches the lookup table at startup.

---

v3.6.x

Released: March 25, 2026

v3.6.0 -- Architectural Parity (March 25)

Breaking change: `fetch(skip:)` is replaced by `fetch(offset:)`. No alias.

```
# BEFORE (v3.5.x):

## 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.fetch(limit: 10, skip: 20)

# AFTER (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

users = User.fetch(limit: 10, offset: 20)
```

Other changes:

- Source directories follow the `src/` prefix convention across all languages
- `TINA4_LOCALE` is the only supported locale environment variable (other names removed)
- Migration file paths standardized to `src/migrations/`

---

v3.5.x

Released: March 24, 2026

v3.5.0 -- Bundled Frontend, Swagger CRUD, Middleware (March 24)

- `tina4js.min.js` (13.6 KB) ships inside the gem. The reactive frontend library loads without a CDN or npm install
- Auto-CRUD routes now include Swagger metadata
- Middleware standardized to `before_*` and `after_*` naming with three built-in middlewares

---

v3.4.x

Released: March 24, 2026

v3.4.0 -- Auth, WebSocket, DatabaseResult (March 24)

Breaking change: Auth method names changed. The old names remain as aliases.

```
# BEFORE (v3.3.x):

## 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)
valid = auth.validate_token(token)

# AFTER (v3.4.0 -- preferred):

## 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)
valid = auth.valid_token(token)

# Old names still work but are deprecated.

## 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
```

HS256 authentication. Set `TINA4_SECRET` in your `.env` and auth uses HS256. Provide RSA key files and it uses RS256. The framework picks the right algorithm.

```
# .env for HS256:

## 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_SECRET=my-secret-key

# .env for RS256:

## 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
```

Bug fix: Base64url padding in HS256 tokens caused validation failures. Fixed.

WebSocket improvements:

- `Router.websocket("/ws/chat")` for route-based WebSocket handlers
- Path-scoped broadcast: messages sent to `/ws/chat` reach only clients connected to that path
- `send_text` renamed to `send` on `WebSocketConnection` (`send_text` kept as alias)

DatabaseResult enhancements:

- `columns` returns column names
- `column_info` provides schema metadata (type, nullable, default) on demand
- `to_paginate` formats results for paginated responses

Frond template additions:

- Ternary-with-filter: `{{ value ? value|upper : "default" }}`
- `data_uri` filter for inline file display in templates

---

v3.3.x

Released: March 24, 2026

v3.3.0 -- Queue API, Route Chaining (March 24)

Breaking change: `Producer` and `Consumer` classes removed. Use `queue.produce()` and `queue.consume()` directly.

```
# BEFORE (v3.2.x):

## 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 = Tina4::Producer.new(queue)
producer.send(message)
consumer = Tina4::Consumer.new(queue)
consumer.listen { |msg| handle(msg) }

# AFTER (v3.3.0):

## 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

queue.produce("channel", { data: "payload" })
```

```
queue.consume("channel") do |job|
  handle(job)
  job.complete
end
```

Route chaining. Mark routes as authenticated or cached with chainable modifiers.

```
Tina4::Router.get("/dashboard").secure do |request, response|
  response.html("<h1>Dashboard</h1>")
end

Tina4::Router.get("/static-page").cache(ttl: 3600) do |request, response|
  response.html("<h1>Cached for one hour</h1>")
end
```

Other additions:

- MongoDB queue backend
- Database session handler for full backend parity
- Valkey added to session handler options
- Migration parity: advanced SQL splitting, status tracking, rollback via CLI
- Auto-increment port if the default is in use; browser opens on startup

---

v3.2.x

Released: March 23, 2026

v3.2.0 -- Flexible Route Handlers (March 23)

Route handlers now accept zero, one, or two parameters. The framework detects what your block expects and provides the right objects.

```
# Zero params -- just return a response:

## 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::Router.get("/health") { "OK" }

# One param -- 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

Tina4::Router.get("/hello") { |response| response.html("Hello") }
```

```
# Two params -- request and response:

## 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::Router.get("/echo") do |request, response|
  response.json({ body: request.body })
end

# Named :request or :req -- single param receives the request:

## 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::Router.post("/submit") { |request| process(request.body) }
```

Bug fix: The 500 error overlay crashed because it did not receive the Rack environment. Fixed.

---

v3.1.x

Released: March 21 -- 22, 2026

v3.1.0 -- ORM Relationships, Caching, Queues (March 22)

The largest feature release after the initial launch. Fourteen capabilities landed in one version.

ORM relationships. Define `has_many`, `has_one`, and `belongs_to` with eager loading.

```
class User < Tina4::ORM
  has_many :posts
  has_one :profile
end

class Post < Tina4::ORM
  belongs_to :user
end

user = User.find(1)
user.posts  # => eager-loaded array of Post objects
```

Caching. Switch between memory, Redis, and file cache by setting one environment variable.

```
# .env:

## 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_CACHE_BACKEND=redis
TINA4_CACHE_URL=redis://localhost:6379

# Code stays the same:

## 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::Cache.set("key", "value", ttl: 300)
Tina4::Cache.get("key")
```

Database query caching. Set `TINA4_DB_CACHE=true` for transparent query result caching.

Queue system. Switch between SQLite, RabbitMQ, and Kafka via `.env` without changing code.

Messenger. Unified messaging driven by environment configuration.

Scaffolding. `tina4 generate model User`, `tina4 generate route api/users`, `tina4 generate migration create_users`, `tina4 generate middleware auth`.

Frond template engine. `raw`/`endraw` blocks and `from` imports.

Performance. Frond pre-compilation caches parsed tokens. File rendering runs faster.

Other additions:

- Production server auto-detection (Puma, cluster mode)
- GitHub Actions CI/CD
- Error pages: clean 404/500/403 without branding
- `numeric_field` type in ORM
- `truthy?()` helper method
- Log rotation

v3.1.1 -- DevMailbox Fix (March 22)

Bug fix: DevMailbox timestamp precision was insufficient for reliable sort ordering.

v3.1.2 -- Documentation Fixes (March 22)

README code examples updated to match the actual v3 API. Quick start guide added.

---

v3.0.x

Released: March 21, 2026

v3.0.0 -- Initial Release (March 21)

The initial Ruby release. Zero gem dependencies. Everything the framework needs -- HTTP server, template engine, ORM, migrations, auth, queue, GraphQL, WebSocket, WSDL -- ships inside a single gem.

Core features:

- Rack-based HTTP server (compatible with Puma, Thin, WEBrick)
- Frond template engine (Twig-compatible syntax)
- ORM with support for SQLite, PostgreSQL, MySQL, MSSQL, and Firebird
- JWT authentication (RS256)
- Queue system
- GraphQL endpoint
- WebSocket server
- WSDL/SOAP service generation
- DevAdmin dashboard with developer tooling
- AI coding tool integration (auto-detect and install context for seven tools)
- Full test suite passing

Quick start:

```
require "tina4"

Tina4::Router.get("/") { |request, response| response.html("<h1>Hello Tina4!</h1>") }

Tina4::App.new.run
```

```
gem install tina4-ruby
```

The server starts on port 7147 by default. Set `host: "0.0.0.0"` for Docker deployments.

Pre-Release (v0.x)

Released: March 18, 2026

Versions v0.4.0 through v0.5.2 were development previews. They established the gem structure and basic routing but lacked the ORM, template engine, and queue system. If you used a v0.x release, upgrade directly to v3.0.0 -- there is no migration path from v0.x.

Complete Feature List

A reference of all 45 features in Tina4 Ruby, grouped by category.

---

1. Core Server

#	Feature	Class / Method	Description
1	HTTP Server	`Tina4::Server`	Zero-dependency HTTP server. Starts with `tina4 serve` (or `TINA4_OVERRIDE_CLIENT=true bundle exec ruby app.rb` to bypass the Rust CLI).
2	Routing	`Tina4::Router.get/post/put/patch/delete`	Declarative route definitions with path params, wildcards, and middleware.
3	Request Object	`request.body`, `request.params`, `request.headers`, `request.cookies`	Parsed inbound request. Handles JSON, form data, and multipart bodies.
4	Response Object	`response.json`, `response.html`, `response.render`, `response.redirect`	Structured response builder with status codes and headers.
5	Middleware	`middleware: ["Name:arg"]`	Per-route or global middleware pipeline. Built-in: `ResponseCache`, `RequestLogger`, `CORS`, `RateLimit`.

---

2. Database

#	Feature	Class / Method	Description
6	Database Connection	`Tina4::Database.new(url)`	Single connection via `TINA4_DATABASE_URL`. Supports SQLite, PostgreSQL, MySQL, MSSQL, Oracle.
7	Raw SQL	`db.query(sql)`, `db.execute(sql)`	Execute raw SQL. `query` returns rows. `execute` returns affected row count or last insert ID.
8	Query Builder	`Tina4::QueryBuilder`	Chainable `.where`, `.select`, `.order`, `.limit`, `.offset`, `.join`.
9	ORM	`Tina4::ORM`	ActiveRecord-style base class. `find`, `save`, `delete`, `all`, `where`.
10	Migrations	`Tina4::Migration`	Version-controlled schema changes. `tina4 migrate` applies pending migrations.
11	get*next*id	`Database.get_next_id(table)`	Race-safe sequence generator using a dedicated sequence table.

---

3. Authentication and Sessions

#	Feature	Class / Method	Description
12	JWT Authentication	`Tina4::Auth`	Token-based auth. Protects routes by default. Bypass with `# @noauth` comment.
13	Session Management	`Tina4::Session`	Server-side sessions stored in file, memory, or database backends.
14	Cookie Handling	`request.cookies`, `response.set_cookie`	Read and write cookies with expiry, domain, secure, and HttpOnly flags.

---

4. Templates and Frontend

#	Feature	Class / Method	Description
15	Frond Templates	`response.render("page.html", data)`	Tina4's zero-dependency template engine. `{variable}`, `{if}`, `{each}`, `{include}`.
16	Static Files	`public/` directory	Files in `public/` are served directly without a route. CSS, JS, images.
17	Frontend Integration	`tina4-js`	The companion 13.6KB reactive frontend library. Signals, components, routing.

---

5. Caching

#	Feature	Class / Method	Description
18	Response Cache	`middleware: ["ResponseCache:ttl"]`	Caches complete HTTP responses. Memory or file backend.
19	Cache API	`Tina4::Cache.get/set/delete/flush`	Direct cache read/write with TTL. Backends: memory, file, Redis, Memcached.
20	Query Caching	`db.cache(ttl) { query }`	Caches database query results for a given TTL.

---

6. Background Processing

#	Feature	Class / Method	Description
21	Queue System	`Tina4::Queue`	File-based queue by default. `push`, `pop`, `consume`, `retry`. Backends: file, RabbitMQ, Kafka, MongoDB.
22	Job Lifecycle	`job.complete`, `job.fail`, `job.retry`	Explicit job state transitions: PENDING → RESERVED → COMPLETED / FAILED / DEAD LETTER.
23	Service Runner	`Tina4::ServiceRunner`	Manages long-running background threads. Auto-restart on crash with exponential backoff.

---

7. Events

#	Feature	Class / Method	Description
24	Event Bus	`Tina4::Events`	In-process synchronous event system.
25	Listener Registration	`Events.on(event, priority:)`	Register a listener block. Priority controls execution order.
26	One-Shot Listeners	`Events.once(event)`	Registers a listener that fires exactly once then removes itself.
27	Listener Removal	`Events.off(event, listener)`, `Events.clear`	Remove specific or all listeners.

---

8. Localization

#	Feature	Class / Method	Description
28	I18n	`Tina4::I18n.new(locale:, path:)`	Loads JSON locale files. Dot-notation key lookup.
29	Translation	`i18n.t("key", field: value)`	Translates a key with `{placeholder}` interpolation.
30	Fallback Locale	`fallback: "en"`	Falls back to a default locale when a key is missing.
31	Locale Switching	`i18n.locale = "fr"`	Switch the active locale at runtime.

---

9. Logging

#	Feature	Class / Method	Description
32	Structured Logger	`Tina4::Log.info/debug/warning/error/fatal`	Structured log entries with timestamp, level, message, and keyword fields.
33	Log Level Control	`TINA4_LOG_LEVEL` env var	Filter output by minimum severity. Values: debug, info, warning, error, fatal.
34	JSON Output	`TINA4_LOG_FORMAT=json`	Emit NDJSON for log aggregators (Datadog, Elastic, CloudWatch).
35	File Logging	`TINA4_LOG_FILE=path`	Write logs to a file in addition to stdout.

---

10. API and Integrations

#	Feature	Class / Method	Description
36	API Client	`Tina4::Api.new(base_url, headers)`	HTTP client for calling external REST APIs. `get`, `post`, `put`, `delete`.
37	WSDL / SOAP Server	`Tina4::WSDL`, `wsdl_operation`	Define SOAP operations with auto-generated WSDL.
38	SOAP Client	`Tina4::WSDL::Client.new(wsdl_url)`	Call remote SOAP services by parsing their WSDL.
39	GraphQL	`Tina4::GraphQL`	Schema-first GraphQL endpoint. Types, queries, mutations, subscriptions.
40	WebSocket	`Tina4::WebSocket`	Real-time bidirectional communication. `on_message`, `broadcast`, `rooms`.
41	SSE / Streaming	`response.stream`	Server-Sent Events for real-time data push via `response.stream(generator)`.

---

11. Developer Tools

#	Feature	Class / Method	Description
42	Swagger / OpenAPI	`Tina4::Swagger`	Auto-generates OpenAPI 3.0 docs from route comments. `/swagger` serves the UI.
43	DI Container	`Tina4::Container`	Register, resolve, and replace services. `registered?`, `clear!`.
44	CLI	`tina4` command	`new`, `server`, `migrate`, `scaffold`, `routes`, `version`.
45	MCP Dev Tools	`Tina4::MCP`	Model Context Protocol server for AI assistant integration.

Feature Summary by Version

Version	Features Added
3.0.0	Core server, routing, request/response, templates, database, ORM, auth, sessions
3.1.0	Queue system, caching, middleware pipeline
3.5.0	GraphQL, WebSocket, Swagger
3.8.0	Service Runner, Events, MCP dev tools
3.10.0	API client, WSDL/SOAP, DI container
3.10.20	get*next*id, race-safe sequences
3.11.0	Localization, structured logging, log level control

Quick Reference: Environment Variables

Variable	Default	Description
`TINA4_DATABASE_URL`	—	Database connection string
`TINA4_ENV`	`development`	Environment: development, production, test
`TINA4_PORT`	`7147`	HTTP server port
`TINA4_LOG_LEVEL`	`info`	Minimum log level
`TINA4_LOG_FORMAT`	`text`	Log format: text or json
`TINA4_LOG_FILE`	—	Log file path (optional)
`TINA4_QUEUE_BACKEND`	`file`	Queue backend: file, rabbitmq, kafka, mongodb
`TINA4_QUEUE_PATH`	`./queue`	File queue storage directory
`TINA4_QUEUE_URL`	—	Queue broker URL (RabbitMQ, Kafka, MongoDB)
`TINA4_CACHE_BACKEND`	`memory`	Cache backend: memory, file, redis, memcached
`TINA4_SECRET`	—	JWT signing secret
`TINA4_TOKEN_LIMIT`	`3600`	JWT expiry in seconds