Tina4 for Node.js 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 Node.js 58

1. What Is Tina4 Node.js . . . 58

2. Prerequisites and Installation . . . 58

What You Need . . . 58

Installing the Tina4 CLI . . . 59

Creating a New Project . . . 59

Starting the Dev Server . . . 60

3. Project Structure Walkthrough . . . 60

4. Your First Route . . . 61

Explicit Route Registration . . . 61

File-Based Routing Alternative . . . 62

Test It . . . 62

Understanding What Happened . . . 62

Adding More HTTP Methods . . . 63

5. Your First Template . . . 63

Create a Base Layout . . . 63

Create a Product Listing Page . . . 64

Create the Route That Renders the Template . . . 65

See It in the Browser . . . 65

How Template Rendering Works . . . 66

About tina4css . . . 66

6. Understanding .env . . . 66

7. The Dev Dashboard . . . 67

8. The app.ts Entry Point . . . 68

9. Manual Setup (No CLI) . . . 69
Step 1: Initialize and Install . . . 69
Step 2: Create app.ts . . . 69
Step 3: Create tsconfig.json . . . 69
Step 4: Create the Folder Structure . . . 69
Step 5: Create .env . . . 70
Step 6: Run It . . . 70
10. Request & Response Fundamentals . . . 70
Reading Query Parameters . . . 70
Reading URL Path Parameters . . . 70
Reading the Request Body . . . 70
Reading Headers . . . 71
Sending JSON Responses . . . 71
Sending HTML / Template Responses . . . 71
Status Codes . . . 71
Worked Example: A Complete Route File . . . 71
11. Exercise: Greeting API + Product List Template . . . 73
Exercise Part A: Greeting API . . . 73
Exercise Part B: Product List Page . . . 74
12. Solutions . . . 74
Solution A: Greeting API . . . 74
Solution B: Product List Page . . . 75
13. Gotchas . . . 77
1. File not auto-discovered . . . 77
2. "Module not found" errors . . . 77
3. JSON response shows HTML . . . 77
4. Template not found . . . 77

5. Port already in use . . . 77
6. Changes not reflected . . . 77
7. .env not loaded . . . 78

Routing **79**

1. How Routing Works in Tina4 . . . 79
2. HTTP Methods . . . 79
3. Path Parameters . . . 80
Typed Parameters . . . 81
Typed Parameters in Action . . . 82
4. Query Parameters . . . 83
5. Route Groups . . . 83
6. Middleware on Routes . . . 84
Middleware on a Single Route . . . 84
Blocking Middleware . . . 85
Middleware on a Group . . . 85
Multiple Middleware . . . 85
7. Route Decorators: @noauth and @secured . . . 86
@noauth -- Public Routes . . . 86
@secured -- Protected GET Routes . . . 86
8. Route Chaining: .secure() and .cache() . . . 86
.secure() . . . 87
.cache() . . . 87
Chaining Both . . . 87
9. Wildcard and Catch-All Routes . . . 87
Wildcard Routes . . . 87
Catch-All Route (Custom 404) . . . 88
10. Route Listing via CLI . . . 88

11. Organizing Route Files . . . 89
Pattern 1: One File Per Resource . . . 89
Pattern 2: File-Based Routing by Feature . . . 89
12. Exercise: Build a Full CRUD API for Products . . . 89
Requirements . . . 89
13. Solution . . . 91
14. Gotchas . . . 93
1. Trailing Slashes Matter . . . 93
2. Parameter Names Must Be Unique in a Path . . . 93
3. Method Conflicts . . . 93
4. Route Handler Must Return a Response . . . 93
5. Async Handlers . . . 93
6. Middleware Must Be Passed as Function References in an Array . . . 93
7. Group Prefix Must Start with a Slash . . . 94
Request & Response **95**
1. The Two Objects You Always Get . . . 95
2. The Request Object . . . 95
method . . . 95
path . . . 95
params . . . 95
query . . . 95
body . . . 96
headers . . . 96
ip . . . 96
cookies . . . 96
files . . . 96
Inspecting the Full Request . . . 96

3. The Response Object . . . 97
json() -- JSON Response . . . 97
html() -- Template or Raw HTML Response . . . 97
text() -- Plain Text Response . . . 98
redirect() -- Redirect Response . . . 98
file() -- File Download Response . . . 98
4. Status Codes . . . 98
5. Custom Headers . . . 99
CORS Headers . . . 99
6. Cookies . . . 99
7. File Uploads . . . 100
Multiple File Uploads . . . 100
Handling a Single File Upload . . . 101
Saving the Uploaded File . . . 102
8. File Downloads . . . 102
9. XML Responses . . . 103
10. Content Negotiation . . . 103
11. Input Validation . . . 104
The Validator Class . . . 104
The Error Response Envelope . . . 104
Upload Size Limits . . . 105
12. Exercise: Build an Image Upload API . . . 105
Requirements . . . 105
13. Solution . . . 105
14. Gotchas . . . 107
1. Forgetting return . . . 107
2. Body Is Undefined for JSON Requests . . . 107

3. File Uploads Return Empty . . . 107
4. Redirect Loops . . . 107
5. Cookie Not Set . . . 108
6. Large Request Body Rejected . . . 108
7. Headers Are Lowercase in Node.js . . . 108

Templates **109**

1. Why Templates . . . 109
2. Variables and Expressions . . . 109
Accessing Nested Data . . . 109
Method Calls on Objects . . . 110
Expressions . . . 110
3. Template Inheritance . . . 110
Base Layout . . . 110
Child Template . . . 111
Using {{ parent() }} . . . 111
4. Includes . . . 112
Passing Variables to Includes . . . 112
5. For Loops . . . 112
The loop Variable . . . 113
Empty Lists . . . 113
Looping Over Key-Value Pairs . . . 113
6. Conditionals . . . 114
if / elseif / else . . . 114
Ternary Operator . . . 114
Testing for Existence . . . 114
{% set %} -- Local Variables . . . 114
7. Filters . . . 114

Complete Filter Reference . . . 115
String Filters . . . 115
Array Filters . . . 116
Encoding Filters . . . 117
Numeric Filters . . . 117
JSON Filters . . . 117
Dict Filters . . . 118
Other Filters . . . 118
Chaining Filters . . . 118
Dumping Values for Debugging . . . 118
8. Macros . . . 119
Defining a Macro . . . 119
Using Macros . . . 120
9. Special Tags . . . 120
{% raw %} -- Literal Output . . . 120
{% spaceless %} -- Remove Whitespace Between Tags . . . 120
{% autoescape %} -- Control HTML Escaping . . . 120
Whitespace Control . . . 121
Comments . . . 121
10. Template Route Export Pattern . . . 121
11. Exercise: Build a Product Catalog Page . . . 122
Requirements . . . 122
Data . . . 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 . . . 125
7. Filters Are Not JavaScript Functions . . . 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 . . . 128
Separate Credentials . . . 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 columnInfo() . . . 130
fetchOne() -- 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
getTables() . . . 133
getColumns() . . . 133
tableExists() . . . 134
getDatabaseType() . . . 134
Schema Info Endpoint . . . 134
8. FakeData Seeder . . . 134
Seeding a Table . . . 135
9. Batch Operations with executeMany() . . . 135
10. Helper Methods: insert(), update(), delete() . . . 136
insert() . . . 136
update() . . . 136
delete() . . . 136
11. Migrations . . . 136
File Naming . . . 136
Generating a Migration . . . 136
Running Migrations . . . 137
Checking Status . . . 137
Rolling Back . . . 137
The Tracking Table . . . 137
Advanced SQL Splitting . . . 138
12. Query Caching . . . 138
13. Exercise: Build a Notes App . . . 138
Requirements . . . 138

14. Solution . . . 139
Migration . . . 139
Routes . . . 139
15. Gotchas . . . 141
1. Forgetting await . . . 141
2. Connection String Formats . . . 141
3. SQLite File Paths . . . 142
4. Parameterised Queries with LIKE . . . 142
5. Boolean Values in SQLite . . . 142
6. Migration Already Applied . . . 142
8. Down Migration Missing on Rollback . . . 142
7. fetch() Returns Empty Array, Not Null . . . 142
ORM 143
1. From SQL to Objects . . . 143
ORM at a Glance: Four Languages, One Shape . . . 143
Defining a Model . . . 143
Common Query Operations . . . 144
2. Defining a Model . . . 145
Field Types . . . 146
Field Options . . . 146
Field Mapping . . . 146
autoMap and Case Conversion . . . 147
getDbColumn and getDbData . . . 148
find() vs where() -- naming convention . . . 148
3. createTable -- Schema from Models . . . 148
4. CRUD Operations . . . 149
save -- Create or Update . . . 149

create -- Build and Save in One Step . . . 149
findById -- Fetch One Record by Primary Key . . . 149
find -- Query by Filter Dict . . . 150
where -- Query with SQL Conditions . . . 150
delete -- Remove a Record . . . 150
Listing Records . . . 151
selectOne -- Fetch a Single Record by SQL . . . 151
load -- Populate an Existing Instance . . . 151
count -- Count Records . . . 152
5. toDict, toJson, and Other Serialisation . . . 152
toDict . . . 152
toJson . . . 152
Other Serialisation Methods . . . 152
6. Relationships . . . 153
foreignKey Field Type — Auto-Wired Relationships . . . 153
hasMany . . . 153
hasOne . . . 155
belongsTo . . . 155
7. Eager Loading . . . 155
Declarative Relationships . . . 156
Nested Eager Loading . . . 156
toDict with Nested Includes . . . 157
8. Soft Delete . . . 157
Deleting and Restoring . . . 158
Including Soft-Deleted Records . . . 158
Counting with Soft Delete . . . 158
When to Use Soft Delete . . . 158

9. Auto-CRUD . . . 158
The autoCrud Flag . . . 158
Manual Registration . . . 159
What the Generated Routes Do . . . 159
Custom Routes Alongside Auto-CRUD . . . 160
Table Filter . . . 160
10. Scopes . . . 160
11. Input Validation . . . 161
12. Exercise: Build a Blog with Relationships . . . 162
Requirements . . . 162
13. Solution . . . 163
14. Gotchas . . . 166
1. Forgetting to call save() . . . 166
2. findById() returns null . . . 166
3. find() vs findById() . . . 167
4. all() not findAll() . . . 167
5. toDict() includes everything . . . 167
6. Validation only runs on validate() . . . 167
7. Foreign key not enforced . . . 167
8. N+1 query problem . . . 168
9. Auto-CRUD endpoint conflicts . . . 168
10. Soft-deleted records appearing in queries . . . 168
15. QueryBuilder Integration . . . 168
Multiple Database Connections . . . 169
QueryBuilder **170**
1. Why a Query Builder? . . . 170
2. Creating a QueryBuilder . . . 170

Standalone: QueryBuilder.fromTable() . . . 170
From an ORM Model: Model.query() . . . 170
3. Selecting Columns . . . 171
4. WHERE Conditions . . . 171
where() -- AND . . . 171
orWhere() -- OR . . . 171
Combining AND and OR . . . 172
5. Joins . . . 172
Inner Join . . . 172
Left Join . . . 172
Multiple Joins . . . 172
6. Grouping and Aggregation . . . 173
groupBy() . . . 173
having() . . . 173
7. Ordering . . . 173
8. Limit and Offset . . . 174
Limit Only . . . 174
Limit with Offset . . . 174
9. Executing Queries . . . 174
get\<T\>() -- All Rows . . . 174
first\<T\>() -- Single Row . . . 175
count() -- Row Count . . . 175
exists() -- Boolean Check . . . 175
10. Inspecting SQL with toSql() . . . 175
11. Using QueryBuilder in Route Handlers . . . 176
List Endpoint with Filters . . . 176
Single Record Lookup . . . 176

Dashboard with Aggregation . . . 177
12. QueryBuilder vs ORM select() . . . 177
13. Exercise: Sales Report API . . . 178
Setup . . . 178
Requirements . . . 178
14. Solution . . . 178
Revenue by Category . . . 178
Top Customers . . . 179
Summary . . . 180
Testing It . . . 180
15. NoSQL: MongoDB Queries . . . 180
Operator Mapping . . . 181
Example . . . 181
16. Gotchas . . . 182
1. Forgetting the Database Adapter . . . 182
2. Parameter Order Matters . . . 182
3. toSql() Does Not Include LIMIT . . . 182
4. Reusing a QueryBuilder . . . 182
5. Mixing QueryBuilder with ORM select() . . . 182
6. OR Without AND . . . 183
Authentication **184**
1. Locking the Door . . . 184
2. JWT Tokens . . . 184
Generating a Token . . . 184
Token Expiry . . . 184
Validating a Token . . . 185
Reading the Payload . . . 185

The Secret Key and Algorithm . . . 185
3. Password Hashing . . . 186
Hashing a Password . . . 186
Checking a Password . . . 186
Registration Example . . . 187
4. The Login Flow . . . 187
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 . . . 192
Generating a Token . . . 192
Validating the Token . . . 192
CSRF Middleware . . . 193
When to Use CSRF Tokens . . . 193
9. Sessions . . . 193
Session Configuration . . . 193
Using Sessions . . . 194
Session Options . . . 194
10. Token Refresh . . . 194
11. Exercise: Build Login, Register, and Profile . . . 195
Requirements . . . 195

Test with: . . . 196

12. Solution . . . 196

Migration . . . 196

Routes . . . 196

13. Gotchas . . . 200

1. Token Expiry Confusion . . . 200

2. Secret Key Management . . . 200

3. CORS with Authentication . . . 200

4. Storing Tokens in localStorage . . . 201

5. Forgetting @noauth on Login . . . 201

6. Password Hash Column Too Short . . . 201

7. Token in URL Query Parameters . . . 201

Sessions & Cookies **202**

1. State in a Stateless World . . . 202

2. Session Configuration . . . 202

Available Backends . . . 202

Redis Configuration . . . 202

MongoDB Configuration . . . 203

Valkey Configuration . . . 203

Database Sessions . . . 203

Session Lifetime . . . 203

Session Cookie . . . 203

3. The Session API . . . 203

Full API Reference . . . 204

4. Reading and Writing Session Data . . . 204

Writing to the Session . . . 204

Reading from the Session . . . 205

Deleting Session Data . . . 205
Checking if a Key Exists . . . 205
Getting All Session Data . . . 205
Preferences Example . . . 206
Visit Counter Example . . . 206
5. Flash Messages . . . 206
Setting a Flash Message . . . 207
Reading a Flash Message . . . 207
Flash Messages in Templates . . . 207
6. Cookies . . . 207
Setting a Cookie . . . 208
Reading a Cookie . . . 208
Deleting a Cookie . . . 208
Cookie Options . . . 209
When to Use Cookies vs Sessions . . . 209
7. Remember Me . . . 210
8. Session Security . . . 211
Regenerate Session ID After Login . . . 211
Destroy a Session . . . 212
Secure Cookie Settings . . . 212
Session Configuration Options . . . 213
9. Exercise: Build a Shopping Cart . . . 213
Requirements . . . 213
Test with: . . . 214
10. Solution . . . 214
11. Gotchas . . . 217
1. Session lost between requests . . . 217

2. Session data is not JSON-serializable . . . 218
3. Cookie not sent on cross-origin requests . . . 218
4. File-based sessions on multi-server deployments . . . 218
5. Session cookie overwritten by another app . . . 218
6. Secure cookies on localhost . . . 218
7. Flash messages shown twice . . . 218
8. Session data disappears after server restart . . . 219
9. Large session data causes slow requests . . . 219

Middleware **220**

1. The Pipeline Pattern . . . 220
2. What Middleware Is . . . 220
3. Built-in CorsMiddleware . . . 221
4. Built-in RateLimiterMiddleware . . . 222
5. Built-in RequestLogger . . . 223
Built-in SecurityHeadersMiddleware . . . 223
Combining All Four Built-In Middleware . . . 224
6. Writing Custom Middleware . . . 224
Function-Based Middleware . . . 224
Class-Based Middleware . . . 224
Example: Request Timer (Class-Based) . . . 225
Example: Request Logger (Function-Based) . . . 225
Example: Security Headers (Class-Based) . . . 225
Example: JSON Content-Type Enforcer (Function-Based) . . . 226
Example: Request ID (Class-Based) . . . 226
Example: Input Sanitization (Class-Based) . . . 226
Example: IP Whitelist (Function-Based) . . . 227
7. Applying Middleware to Routes . . . 227

8. Middleware on Route Groups . . . 227
9. Execution Order . . . 228
10. Short-Circuiting . . . 229
Conditional Short-Circuit . . . 230
11. Modifying Request and Response . . . 230
Adding Data to the Request . . . 230
Modifying the Response . . . 231
Adding a Request ID . . . 231
12. Real-World Example: JWT Authentication Middleware . . . 231
13. Real-World Example: API Key Middleware with Database Lookup . . . 233
14. Real-World Middleware Stack . . . 234
15. Exercise: Build an API Key Middleware System . . . 234
Requirements . . . 234
Test with: . . . 235
16. Solution . . . 235
Migration . . . 235
Routes . . . 236
17. Gotchas . . . 238
1. Forgetting to Call next() . . . 238
2. Middleware Modifies Response After It Is Sent . . . 238
3. Middleware Applied to Wrong Routes . . . 238
4. Middleware Execution Order Surprises . . . 238
5. Error in Middleware Breaks the Chain . . . 239
6. Database Connections in Middleware . . . 239
7. Modifying Request in Middleware Does Not Persist . . . 239
8. CORS Preflight Returns 404 . . . 239
9. Rate Limiter Counts Preflight Requests . . . 240

10. Middleware File Not Auto-Loaded . . . 240
11. Short-Circuiting Skips Cleanup . . . 240
12. Middleware Must Be Passed as Function References . . . 240

Security **240**

1. Secure-by-Default Routing . . . 241
Making a Write Route Public . . . 241
Protecting a GET Route . . . 241
The Rule . . . 242
2. CSRF Protection . . . 242
How It Works . . . 242
The Template . . . 242
The Middleware . . . 242
AJAX Requests . . . 243
Tokens in Query Strings — Blocked . . . 243
Skipping CSRF Validation . . . 243
Disabling CSRF Globally . . . 243
3. Session-Bound Tokens . . . 243
How Stolen Tokens Fail . . . 244
4. Security Headers . . . 244
HSTS — Enforcing HTTPS . . . 244
Customising Headers . . . 245
5. SameSite Cookies . . . 245
6. Login Flow — Complete Example . . . 245
The Login Route . . . 246
The Login Form . . . 247
Protected Pages — Checking the Session . . . 247
Logout — Destroying the Session . . . 248

7. Handling Expired Sessions . . . 248
The Pattern: Redirect to Login, Then Back . . . 248
Token Refresh . . . 249
8. Rate Limiting . . . 249
9. CORS and Credentials . . . 250
10. Security Checklist . . . 250
Gotchas . . . 251
1. "My POST route returns 401 but I didn't add auth" . . . 251
2. "CSRF validation fails on AJAX requests" . . . 251
3. "I disabled CSRF but forms still fail" . . . 251
4. "My Content-Security-Policy blocks inline scripts" . . . 251
5. "User stays logged in after session expires" . . . 252
Exercise: Secure Contact Form . . . 252
Solution . . . 252
Caching **254**
1. From 800ms to 3ms . . . 254
2. Response Caching with ResponseCache Middleware . . . 254
Cache Headers . . . 255
Caching with Query Parameters . . . 255
What Not to Cache . . . 255
3. Memory Cache (Default) . . . 256
4. Redis Cache . . . 256
Why Redis? . . . 256
5. File Cache . . . 257
When to Use File Cache . . . 257
6. Direct Cache API . . . 257

cacheSet . . . 257
cacheGet . . . 257
cacheDelete . . . 258
Real-World Pattern: Cache-Aside . . . 258
7. Database Query Caching . . . 259
When to Use DB Cache . . . 259
When Not to Use DB Cache . . . 259
Skipping Cache for Specific Queries . . . 260
8. Cache Invalidation Strategies . . . 260
Strategy 1: Time-Based Expiry (TTL) . . . 260
Strategy 2: Event-Based Invalidation . . . 260
Strategy 3: Write-Through Cache . . . 260
Choosing a Strategy . . . 261
9. TTL Management . . . 261
Dynamic TTL . . . 262
10. Cache Statistics . . . 262
11. Combining Cache Layers . . . 263
12. Exercise: Cache an Expensive Product Listing Endpoint . . . 264
Requirements . . . 264
Test with: . . . 264
13. Solution . . . 265
14. Gotchas . . . 267
1. Caching Authenticated Responses . . . 267
2. Cache Stampede . . . 267
3. Memory Cache Lost on Restart . . . 268
4. Stale Data After Database Update . . . 268
5. Cache Key Collisions . . . 268

6. Serialization Overhead . . . 268
7. Forgetting to Set TTL . . . 268
Queue System **270**
1. Do Not Make the User Wait . . . 270
2. Why Queues Matter . . . 270
3. File Queue (Default) . . . 270
Creating a Queue and Pushing a Job . . . 270
Convenience Method: produce . . . 270
Push with Priority . . . 270
Queue Size . . . 271
4. Pushing from Route Handlers . . . 271
5. Consuming Jobs . . . 271
Retry with Delay . . . 272
Manual Pop . . . 272
6. Job Lifecycle . . . 272
Job Methods . . . 272
Job Properties . . . 272
7. Retry and Dead Letters . . . 273
Max Retries . . . 273
Retrying Failed Jobs . . . 273
Dead Letters . . . 273
Purging Jobs . . . 273
8. Switching Backends . . . 273
Default: File . . . 273
RabbitMQ . . . 273
Kafka . . . 274
MongoDB . . . 274

9. Multiple Jobs from One Action . . . 274
10. Exercise: Build an Email Queue . . . 274
Requirements . . . 274
Test with: . . . 275
11. Solution . . . 275
12. Gotchas . . . 276
1. Always call complete or fail . . . 276
2. Worker not picking up messages . . . 277
3. Payload must be JSON-serializable . . . 277
4. Dead letters pile up . . . 277
5. File backend for production . . . 277
6. Consumer returns nothing . . . 277
Events System **278**
1. Decouple Without Wiring Everything Together . . . 278
2. Listening for Events . . . 278
3. Emitting Events . . . 278
4. One-Shot Listeners with once() . . . 279
5. Priority . . . 279
6. Removing Listeners with off() . . . 280
7. Clearing All Listeners with clear() . . . 280
8. Listing Active Listeners . . . 280
9. Organising Events in a Real Application . . . 280
10. Exercise: Order Processing Pipeline . . . 281
Requirements . . . 281
Test with: . . . 281
11. Solution . . . 282

12. Gotchas . . . 283
1. Async listeners are not awaited . . . 283
2. Same function reference required for off() . . . 283
3. Listeners persist across requests . . . 283
4. Forgetting to import the events file . . . 283
Localization **284**
1. Your App Speaks One Language. Your Users Speak Many. . . . 284
2. Locale Files . . . 284
3. Creating an I18n Instance . . . 285
4. Translating Keys with t() . . . 285
5. Interpolation . . . 286
6. Locale Switching Per Request . . . 286
7. Fallback Locale . . . 287
8. Listing Available Locales . . . 287
9. Using I18n in Templates . . . 287
10. Exercise: Multilingual API Errors . . . 288
Requirements . . . 288
Test with: . . . 288
11. Solution . . . 288
12. Gotchas . . . 290
1. Locale files must be valid JSON . . . 290
2. Missing keys return the key string . . . 290
3. setLocale is not thread-safe for concurrent requests . . . 290
4. Placeholder names are case-sensitive . . . 290
Structured Logging **291**
1. console.log Is Not a Logging Strategy . . . 291

2. The Four Log Levels . . . 291
3. Controlling Verbosity with TINA4LOGLEVEL . . . 291
4. Adding Context Fields . . . 292
5. Logging Errors . . . 292
6. Request-Scoped Logging . . . 293
7. Performance Logging . . . 294
8. Exercise: Add Logging to an Existing API . . . 294
Requirements . . . 294
Expected log output (debug level): . . . 294
9. Solution . . . 295
10. Gotchas . . . 296
1. Logging sensitive data . . . 296
2. Logging in hot paths adds latency . . . 296
3. Circular references in context objects . . . 296
4. TINA4LOGLEVEL defaults to info . . . 296
Email with Messenger **297**
1. Every App Sends Email . . . 297
2. Messenger Configuration via .env . . . 297
Common Provider Configurations . . . 297
3. Constructor Override Pattern . . . 298
4. Sending Plain Text Email . . . 298
The send() Method Signature . . . 299
5. Sending HTML Email with Text Fallback . . . 299
6. Adding Attachments . . . 300
Attachments with Custom Names and Binary Content . . . 301
7. CC, BCC, and Reply-To . . . 301

8. Custom Headers . . . 302
Default Headers on an Instance . . . 302
Per-Email Headers . . . 302
9. Reading Inbox via IMAP . . . 302
Listing Inbox Messages . . . 303
Reading a Specific Message . . . 304
Searching Messages . . . 304
Other IMAP Operations . . . 305
Read from a Specific Folder . . . 305
Testing IMAP Connectivity . . . 305
10. Dev Mode: Email Interception . . . 305
Disabling Interception . . . 306
11. Using Templates for Email Content . . . 306
Rendering and Sending . . . 307
12. Sending Email via Queues . . . 308
13. Exercise: Build a Contact Form with Email Notification . . . 309
Requirements . . . 309
Test with: . . . 309
14. Solution . . . 309
15. Gotchas . . . 312
1. Gmail Blocks "Less Secure" Apps . . . 312
2. Emails Go to Spam . . . 313
3. HTML Email Looks Broken . . . 313
4. Attachment File Not Found . . . 313
5. Dev Mode Silently Intercepts Emails . . . 313
6. Email Template Variables Not Substituted . . . 313
7. Connection Timeout on Send . . . 314

8. IMAP Host Not Configured . . . 314

Frontend Integration 315

1. Beyond JSON APIs . . . 315
2. What Ships with Tina4 . . . 315
3. The Grid System . . . 315
4. Components . . . 316
   - Navbar . . . 316
   - Cards . . . 316
   - Buttons . . . 317
   - Tables . . . 317
   - Alerts . . . 317
   - Modals . . . 318
   - Progress Bars . . . 318
   - Badges . . . 318
   - Forms . . . 319
   - Utility Classes . . . 319
5. SCSS Customization . . . 319
   - Live SCSS Compilation . . . 320
6. frond.js -- The JavaScript Helper . . . 320
   - AJAX Requests . . . 320
   - Form Submission via AJAX . . . 321
   - Token Management . . . 321
   - Loading Indicators . . . 321
   - WebSocket (Covered in Chapter 23) . . . 322
7. JavaScript API Reference . . . 322
   - Including the Scripts . . . 322
   - 7.1 tina4.min.js -- Core Utilities . . . 322

loadPage(url, targetId) . . . 322
saveForm(formId, url, method) . . . 322
sendRequest(url, method, data, callback) . . . 323
7.2 frond.min.js -- Template Engine Client-Side Helpers . . . 323
AJAX Form Handling . . . 323
WebSocket Auto-Reconnect . . . 323
Token Refresh . . . 323
Dynamic Template Loading . . . 323
7.3 tina4js.min.js -- Reactive Frontend Framework . . . 324
Reactive State: signal(), computed(), effect() . . . 324
DOM Rendering: html Tagged Template . . . 324
Web Components: Tina4Element . . . 324
Fetch Wrapper: api() . . . 325
WebSocket Client: ws() . . . 325
Client-Side Routing: route(), navigate() . . . 325
8. Building an Admin Dashboard . . . 325
Base Template . . . 325
Dashboard Page . . . 326
Dashboard Route . . . 328
9. Dark Mode . . . 329
Respecting System Preference . . . 329
10. Responsive Design . . . 329
Responsive Sidebar . . . 329
Responsive Tables . . . 330
Hiding Elements by Screen Size . . . 330
11. Building a Users Page with AJAX . . . 330
The Template . . . 330

The Route . . . 332
12. Building a CRUD Interface . . . 332
13. Static File Serving . . . 333
14. Integrating with React, Vue, or Svelte . . . 334
15. Exercise: Build an Admin Dashboard . . . 334
Requirements . . . 334
Test by: . . . 334
16. Solution . . . 335
17. Gotchas . . . 335
1. CSS Not Loading . . . 335
2. frond.js Functions Not Found . . . 336
3. Dark Mode Flickers on Page Load . . . 336
4. SCSS Not Compiling . . . 336
5. Modal Does Not Open . . . 336
6. Grid Columns Do Not Stack on Mobile . . . 336
7. Static Files Return 404 . . . 336
8. Form Data Not Reaching the Server . . . 337
9. Loading Indicator Never Disappears . . . 337
10. CORS Errors with Separate Frontend . . . 337
11. Large Bundle Sizes . . . 337
12. React Router Conflicts . . . 337
18. HtmlElement — Programmatic HTML Builder . . . 337
Testing **339**
1. Why Tests Matter More Than You Think . . . 339
2. Your First Test . . . 339
How It Works . . . 340
3. Assertion Functions . . . 340

assertEqual . . . 340
assertThrows . . . 340
assertTrue / assertFalse . . . 341
4. Testing Business Logic . . . 341
5. Testing ORM Models . . . 341
Test Database . . . 342
6. Testing Routes . . . 342
7. Testing Authentication . . . 343
8. Resetting Tests Between Files . . . 343
9. Runner Options . . . 344
CI Integration . . . 344
10. Running Tests . . . 344
Run All Tests . . . 344
Run a Specific Test File . . . 344
With npm Scripts . . . 344
11. Testing Best Practices . . . 345
Test One Thing Per Function . . . 345
Use Named Functions for Readable Output . . . 345
Isolate Tests . . . 345
Use Unique Values . . . 346
12. Failed Test Output . . . 346
13. Exercise: Write Tests for Utility Functions . . . 346
14. Solution . . . 347
15. Gotchas . . . 347
1. The tests() Decorator Returns the Original Function . . . 347
2. Arguments Are Passed as an Array . . . 347
3. Named Functions Are Required for Readable Output . . . 347

4. Call resetTests() Between Separate Test Files . . . 348
5. runAllTests() Returns Results . . . 348
6. Database State Carries Between Tests . . . 348
7. Async Functions Need Special Handling . . . 348

Scaffolding 349

The CRUD Generator . . . 349
What Each File Contains . . . 349
Run It . . . 351
Individual Generators . . . 352
Model . . . 352
Route . . . 352
Migration . . . 352
Middleware . . . 352
Test . . . 352
Form . . . 352
View . . . 352
CRUD . . . 353
Auth . . . 353
AutoCRUD . . . 353
Usage . . . 353
The Auth Generator . . . 353
Field Types . . . 354
Table Naming Convention . . . 354
Combining Generators . . . 355
Exercise: Scaffold a Blog . . . 355
Gotchas . . . 356
Generators Do Not Overwrite . . . 356

Run Migrate After Generate . . . 356
File Naming Matters . . . 356
Field Changes Need New Migrations . . . 356
Singular Table Names . . . 356
npx Alternative . . . 356

API Documentation with Swagger 357

1. The 47-Endpoint Problem . . . 357
2. What Swagger/OpenAPI Is . . . 357
3. Enabling Swagger . . . 357
Swagger in Production . . . 357
4. Adding Descriptions to Routes . . . 358
Path Parameters . . . 358
Query Parameters . . . 358
5. Documenting Request Bodies . . . 358
Supported Types in @body and @response . . . 359
6. Documenting Responses . . . 359
7. Tags for Grouping . . . 360
8. Example Values . . . 360
9. Authentication in Swagger . . . 361
Public and Secured Routes . . . 361
Authorize Button . . . 361
Workflow . . . 361
10. Try-It-Out from the Swagger UI . . . 362
11. A Complete Documented API . . . 362
12. Hiding Routes . . . 363
13. Customizing the Swagger Info Block . . . 364

14. Generating Client SDKs . . . 364
Available Generators . . . 364
15. Auto-CRUD Routes in Swagger . . . 365
16. Exercise: Document a Complete User API . . . 365
Requirements . . . 365
17. Solution . . . 366
18. Gotchas . . . 367
1. Annotations Must Be Directly Above the Route . . . 367
2. Missing @tags Makes Endpoints Hard to Find . . . 367
3. @body Must Be Valid JSON . . . 367
4. Swagger Shows Routes You Did Not Annotate . . . 367
5. Response Examples Do Not Match Actual Responses . . . 367
6. Swagger UI Not Available in Production . . . 367
7. SDK Generation Produces Incorrect Types . . . 367
8. @secured Routes Not Sending Auth Token . . . 367
API Client **369**
1. Calling External APIs Without the Boilerplate . . . 369
2. The api Singleton . . . 369
3. Configuring the Client . . . 369
4. GET Requests . . . 370
5. POST Requests . . . 370
6. PUT and PATCH Requests . . . 371
7. DELETE Requests . . . 371
8. Response Format . . . 371
9. Per-Request Headers . . . 372
10. Using api Inside Route Handlers . . . 372

11. Exercise: GitHub User Profile Proxy . . . 373
Requirements . . . 373
Test with: . . . 373
12. Solution . . . 373
13. Gotchas . . . 374
1. Configure before first use . . . 374
2. Swallowing upstream errors . . . 375
3. Timeout not set . . . 375
4. Sending secrets in URLs . . . 375
GraphQL and SOAP **376**
1. The Problem GraphQL Solves . . . 376
2. GraphQL vs REST -- A Quick Comparison . . . 376
3. Enabling GraphQL . . . 377
4. Defining a Schema . . . 377
5. Writing Resolvers . . . 378
Testing the Queries . . . 378
6. Mutations . . . 379
Mutation Resolvers . . . 380
Testing Mutations . . . 381
7. Nested Types and Relationships . . . 382
8. Auto-Generating Schema from ORM Models . . . 384
9. The GraphiQL Playground . . . 384
10. Query Variables . . . 385
11. Exercise: Build a GraphQL API for a Blog . . . 385
Requirements . . . 385
Test with these queries: . . . 386
12. Solution . . . 386

Schema . . . 386
Resolvers . . . 387
13. Input Validation . . . 389
Example: Missing Required Argument . . . 389
14. Field-Level Auth Directives . . . 390
Passing Auth Context . . . 390
Schema Example . . . 390
Behavior on Failed Auth . . . 390
15. Gotchas . . . 391
1. Schema File Not Found . . . 391
2. Resolver Not Called . . . 391
3. Nested Resolver Returns Wrong Data . . . 391
4. Mutation Input Not Parsed . . . 391
5. N+1 Query Problem . . . 392
6. GraphQL Playground Returns 404 . . . 392
7. Type Mismatch Between Schema and Resolver . . . 392
14. SOAP / WSDL Services . . . 393
What is SOAP? . . . 393
Defining a SOAP Service . . . 393
Registering the Service . . . 394
Auto-Generated WSDL . . . 394
Type Mappings . . . 394
Lifecycle Hooks . . . 395
Testing with curl . . . 395
A More Complete Example . . . 396
SOAP Gotchas . . . 396
When to Use SOAP vs REST vs GraphQL . . . 397

Real-time with WebSocket 398

1. The Refresh Button Problem . . . 398

2. What WebSocket Is . . . 398

3. Router.websocket() -- WebSocket as a Route . . . 398

Starting the Server . . . 399

4. Connection Events . . . 399

Open . . . 399

Message . . . 400

Close . . . 400

A Complete Handler . . . 400

5. Sending to a Single Client . . . 401

6. Broadcasting to All Clients . . . 402

Broadcast Excluding Sender . . . 402

Sending JSON . . . 403

Closing a Connection . . . 403

Connection Methods Summary . . . 404

7. Path-Scoped Isolation . . . 404

8. Building a Live Chat . . . 405

WebSocket Handler . . . 405

9. Live Notifications . . . 406

Use Cases . . . 407

Scoped Notifications . . . 407

10. Connecting from JavaScript . . . 407

Basic Connection . . . 408

Auto-Reconnect . . . 408

Using Native WebSocket . . . 409

Manual Auto-Reconnect with Native WebSocket . . . 409
11. A Complete Chat Page . . . 409
12. Exercise: Build a Real-Time Chat Room . . . 411
Requirements . . . 411
Test by: . . . 412
13. Solution . . . 412
14. Scaling with a Backplane . . . 414
Configuration . . . 415
Requirements . . . 415
15. Gotchas . . . 415
1. WebSocket Needs a Persistent Server . . . 415
2. Messages Are Strings, Not Objects . . . 415
3. Connection Count Is Per-Path . . . 415
4. Broadcasting Does Not Scale Across Servers . . . 416
5. Large Messages Cause Disconnects . . . 416
6. Memory Leak from Tracking Connected Users . . . 416
7. No Authentication on WebSocket Connect . . . 416
8. Route Params Use {id} Not :id . . . 416
Server-Sent Events (SSE) 418
1. The Polling Problem . . . 418
2. What SSE Is . . . 418
3. Your First Stream . . . 419
4. The SSE Protocol . . . 420
The data: Field . . . 420
The event: Field . . . 420
The id: Field . . . 420
The retry: Field . . . 420

The Delimiter . . . 421
5. Frontend: EventSource . . . 421
Basic Usage . . . 421
Named Events . . . 421
Closing the Connection . . . 422
With Authentication . . . 422
6. Real Examples . . . 422
Live Dashboard . . . 422
Build Progress . . . 423
LLM Chat Streaming . . . 423
File Processing Progress . . . 424
7. Custom Content Types . . . 425
NDJSON Streaming . . . 425
Consuming NDJSON on the Client . . . 425
8. SSE vs WebSocket . . . 426
9. Production Considerations . . . 427
Nginx Proxy Buffering . . . 427
Connection Limits . . . 427
Heartbeat Messages . . . 427
10. Exercise: Build a Live Metrics Dashboard . . . 428
Requirements . . . 428
Test by: . . . 428
11. Solution . . . 428
12. What You Learned . . . 430
WSDL / SOAP **431**
1. Legacy Services Are Still Services . . . 431

2. Consuming an External SOAP Service . . . 431
Loading the WSDL . . . 431
Calling an Operation . . . 431
Handling Errors . . . 431
3. wsdl_operation Decorator . . . 432
4. Publishing a SOAP Service . . . 432
Auto WSDL . . . 433
5. Calling Your Published Service . . . 433
6. Exercise: Expose a Product Catalogue as a SOAP Service . . . 434
Requirements . . . 434
Test with: . . . 434
7. Solution . . . 435
8. Gotchas . . . 435
1. SOAP is XML -- whitespace matters in some parsers . . . 435
2. Namespace mismatches break everything . . . 436
3. WSDL.load() is slow on first call . . . 436
4. Decimal types lose precision in JavaScript . . . 436
DI Container **437**
1. Stop Passing Dependencies Through Every Function Call . . . 437
2. The Container Class . . . 437
3. Registering a Service . . . 437
4. Singleton Registration . . . 438
5. Resolving Services with get() . . . 438
6. Checking Registration with has() . . . 438
7. Resetting the Container . . . 438
8. Services That Depend on Other Services . . . 439

9. Application-Wide Container Pattern . . . 439
10. Testing with the DI Container . . . 440
11. Exercise: Build a Service Container for a Store API . . . 440
Requirements . . . 440
Test with: . . . 441
12. Solution . . . 441
13. Gotchas . . . 442
1. Registering after first get() . . . 442
2. Circular dependencies . . . 442
3. Singletons holding stale state . . . 442
4. Over-registering transient services . . . 443
Service Runner **444**
1. Work That Never Stops . . . 444
2. Defining a Service . . . 444
3. Registering and Starting Services . . . 444
4. Service Lifecycle . . . 445
Stopping a Service . . . 445
Restart on Crash . . . 445
5. Service Options . . . 446
6. A Real-World Example: Queue Drainer . . . 446
7. Health Check Endpoint . . . 447
8. Integrating with the Application . . . 447
9. Exercise: Daily Report Service . . . 448
Requirements . . . 448
Test by setting the interval to 5 seconds so you can see it run: . . . 448
10. Solution . . . 448

11. Gotchas . . . 450
1. Services run immediately by default . . . 450
2. Uncaught exceptions outside run() crash the process . . . 450
3. interval is wall-clock delay between runs, not between start times . . . 450
4. Services do not share state safely . . . 450
MCP Dev Tools **451**
1. What is MCP? . . . 451
2. How It Works . . . 451
Localhost-Only Security . . . 451
3. Connecting AI Tools . . . 451
Claude Code . . . 451
Cursor . . . 452
Manual Connection . . . 452
4. Built-in Tools . . . 452
Database . . . 452
Routes . . . 452
Templates . . . 453
Files . . . 453
Migrations . . . 453
Data and ORM . . . 453
Infrastructure . . . 454
Debugging . . . 454
Project introspection . . . 454
Persistent code index . . . 455
Framework documentation (prose) . . . 455
Live API RAG (reflection) . . . 455
Plan management . . . 455

5. Protocol Details . . . 456
SSE Endpoint (GET) . . . 456
Message Endpoint (POST) . . . 456
Lifecycle . . . 457
6. Troubleshooting . . . 457
Building Custom MCP Servers **458**
1. Beyond Dev Tools . . . 458
2. Creating an MCP Server . . . 458
3. Registering Tools with mcpTool . . . 458
4. Registering Resources with mcpResource . . . 459
5. Class-Based MCP Services . . . 459
6. Securing MCP Endpoints . . . 460
7. Testing MCP Tools . . . 461
8. Complete Example: CRM MCP Server . . . 461
9. Best Practices . . . 462
Dev Tools **464**
1. Debugging at 2am . . . 464
2. Enabling the Dev Dashboard . . . 464
3. Dashboard Overview . . . 464
System Overview . . . 464
Request Inspector . . . 465
Error Log . . . 465
Queue Manager . . . 465
WebSocket Monitor . . . 465
Routes . . . 465
Mail . . . 465

4. The Dev Toolbar . . . 465
Disabling the Toolbar . . . 466
5. Error Overlay . . . 466
Example Error . . . 466
Error Overlay in Production . . . 467
6. The Gallery . . . 467
7. Live Reload . . . 467
How It Works . . . 468
8. Request Inspector . . . 468
Request Details Panel . . . 468
Filtering Requests . . . 469
Request Replay . . . 469
9. SQL Query Runner . . . 469
Safety . . . 470
10. Queue Monitor . . . 470
11. System Info . . . 470
12. Logging . . . 471
13. Health Check . . . 471
14. Exercise: Debug a Failing Route . . . 471
Setup . . . 472
Requirements . . . 472
Solution . . . 473
15. Gotchas . . . 473
1. Dev Dashboard Accessible on Network . . . 473
2. Live Reload Causes Connection Drops . . . 473
3. Error Overlay Shows in Production . . . 474
4. Request Inspector Slows Down the Server . . . 474

5. SQL Runner Executes Destructive Queries . . . 474
6. Template Changes Not Reflected . . . 474
7. Queue Monitor Shows No Jobs . . . 474
8. Debug Mode in Version Control . . . 474
9. Logs Fill Up Disk . . . 475
10. Performance Overhead . . . 475

Tina4 CLI **476**

1. Getting a New Developer Up to Speed . . . 476
2. tina4 init -- Project Scaffolding . . . 476
   - Language Detection . . . 476
   - Explicit Language Selection . . . 477
   - Init into an Existing Directory . . . 477
3. tina4 serve -- Dev Server . . . 477
   - Options . . . 477
   - Bypassing the CLI . . . 478
4. tina4 generate model -- ORM Scaffolding . . . 478
   - Adding Fields . . . 478
   - Field Types . . . 479
   - Options . . . 480
5. tina4 generate route -- CRUD Route Scaffolding . . . 480
   - Options . . . 481
6. tina4 generate migration -- Migration Scaffolding . . . 482
   - When to Generate a Migration . . . 482
7. tina4 generate middleware -- Middleware Scaffolding . . . 482
8. Generate All at Once . . . 483
9. tina4 doctor -- Health Check . . . 483
10. tina4 test -- Running Tests . . . 484

Test Options . . . 484
11. tina4 routes -- Route Listing . . . 484
Filtering . . . 485
12. tina4 migrate -- Database Migrations . . . 485
Rollback . . . 485
Migration Table Auto-Upgrade . . . 485
Status . . . 485
13. tina4 queue -- Queue Management . . . 486
14. tina4 build -- Production Build . . . 486
15. Custom CLI Commands . . . 486
16. Exercise: Scaffold a Feature in 5 Commands . . . 487
Requirements . . . 487
Expected Commands . . . 487
Solution . . . 487
17. Gotchas . . . 488
1. tina4 Command Not Found . . . 488
2. Wrong Language Detected . . . 489
3. Generated Files Overwrite Existing Code . . . 489
4. Migration Order Issues . . . 489
5. tina4 serve Uses Wrong Port . . . 489
6. Doctor Shows False Warnings . . . 489
7. Model Name Must Be PascalCase . . . 489
8. Build Fails . . . 490
9. Port Conflict . . . 490
18. Documentation . . . 490
19. Test Port (Dual-Port Development) . . . 490

Vibe Coding with AI 491

Why Tina4 is Built for AI-Assisted Development . . . 491

The Zero-Dependency Advantage . . . 491

CLAUDE.md -- The AI's Instruction Manual . . . 491

Skills -- Deep Framework Knowledge . . . 492

tina4-maintainer . . . 492

tina4-developer . . . 492

tina4-js . . . 492

The Convention Advantage . . . 492

Real-World Vibe Coding Session . . . 492

TypeScript-Specific Advantages . . . 493

Supported AI Tools . . . 494

The AI Chat in Dev Dashboard . . . 494

Tips for Effective Vibe Coding with Tina4 . . . 495

Exercise . . . 495

Gotchas . . . 495

The Philosophy . . . 496

Environment Variables 497

Core Server . . . 498

Secrets and Authentication . . . 499

Database . . . 500

CORS . . . 501

Security Headers . . . 501

Rate Limiting . . . 502

Sessions . . . 502

Redis/Valkey session backend . . . 503

MongoDB session backend . . . 504
Cache . . . 504
Queues . . . 505
Kafka queue backend . . . 505
RabbitMQ queue backend . . . 505
MongoDB queue backend . . . 506
WebSocket . . . 506
Email . . . 507
Logging . . . 507
File logging . . . 508
Localisation . . . 509
Uploads . . . 509
Services (background tasks) . . . 509
Routing . . . 510
Templates . . . 510
GraphQL . . . 510
AI and MCP Tooling . . . 511
MCP . . . 512
Swagger / OpenAPI . . . 512
Minimal .env for Development . . . 512
Minimal .env for Production . . . 513
Deployment **514**
1. From Development to Production . . . 514
2. Production .env Configuration . . . 514
Key Differences from Development . . . 514
Sensitive Values . . . 515

3. Building for Production . . . 515
4. Docker Deployment . . . 515
Dockerfile . . . 515
.dockerignore . . . 516
Building and Running . . . 516
Docker Compose . . . 516
5. Node.js Cluster for Production . . . 517
How Many Workers? . . . 517
6. Health Checks . . . 518
Broken File Check . . . 518
7. Graceful Shutdown . . . 519
Configuring Shutdown Timeout . . . 519
Docker Stop Grace Period . . . 519
8. Log Rotation . . . 519
Using logrotate (Linux) . . . 519
Docker Logging . . . 520
9. Nginx Reverse Proxy . . . 520
10. SSL/TLS with Let's Encrypt . . . 521
With Nginx and Certbot . . . 521
With Docker (Using Traefik as Reverse Proxy) . . . 522
Certificate Monitoring . . . 522
11. Process Management with systemd . . . 522
12. Queue Workers in Production . . . 523
13. Zero-Downtime Deployment . . . 524
14. Scaling . . . 524
Vertical Scaling . . . 524
Load Balancing with Nginx . . . 524

Docker Scaling . . . 525
Horizontal Scaling . . . 525
Scaling Considerations . . . 525
15. Monitoring . . . 526
Log Aggregation . . . 526
Uptime Monitoring . . . 526
Application Performance Monitoring (APM) . . . 527
16. Exercise: Docker Deploy . . . 527
Requirements . . . 527
Solution . . . 528
17. Gotchas . . . 528
1. .env Not Loaded in Docker . . . 528
2. SQLite Database Lost on Container Restart . . . 529
3. WebSocket Connections Drop Behind Nginx . . . 529
4. Static Files Slow . . . 529
5. Memory Usage Grows Over Time . . . 529
6. Container Starts Before Database Is Ready . . . 529
7. SSL Certificate Not Renewing . . . 530
8. Scaled Instances Have Different State . . . 530
9. Debug Mode in Production . . . 530
10. .env in Version Control . . . 530
11. Missing Migrations . . . 530
12. Port Conflicts . . . 530
Building a Complete App **531**
1. Putting It All Together . . . 531
2. Planning the App . . . 531
Models . . . 531

Relationships . . . 531
Routes . . . 532
Templates . . . 532
3. Step 1: Init Project and Set Up Database . . . 532
Create Migrations . . . 532
4. Step 2: Define Models . . . 533
5. Step 3: Auth Middleware . . . 535
6. Step 4: Auth Routes . . . 535
Test Registration and Login . . . 537
7. Step 5: Task Routes . . . 538
Test the Task API . . . 542
8. Step 6: Dashboard Stats with Caching . . . 542
9. Step 7: WebSocket for Real-Time Updates . . . 544
10. Step 8: Email Notifications via Queue . . . 545
11. Step 9: Dashboard Template . . . 546
12. Step 10: Tests . . . 548
13. Step 11: Docker Deployment . . . 551
14. The Complete Project Structure . . . 553
15. What You Built . . . 553
16. What to Build Next . . . 554
17. Closing Thoughts -- The Tina4 Philosophy . . . 555
Release Notes **556**
v3.12.4 (2026-05-06) . . . 556
25 new env vars across all 4 frameworks . . . 556
Logging — env-driven file output + rotation . . . 556
Documentation-truth CI gate now strict on both axes . . . 557

Tests added . . . 557
Upgrade path . . . 557
v3.12.3 (2026-05-05) . . . 557
Breaking changes (Ruby + PHP only) . . . 557
Doc parity — CLAUDE.md and book chapter 33 . . . 558
Other fixes . . . 558
Genuine gaps surfaced by the parity audit (follow-up, not blocking 3.12.3) . . . 558
Upgrade path . . . 559
v3.12.2 (2026-05-05) . . . 559
Firebird URL auto-detect . . . 559
Bug fix specific to PHP — mysqli localhost+port quirk . . . 560
Other fixes . . . 560
v3.12.1 (2026-05-04) . . . 560
v3.12.0 (2026-05-04) . . . 560
Why this release . . . 561
What changed . . . 561
Bug fixes shipped alongside the rename . . . 561
Migration — every renamed var . . . 562
Names that stay un-prefixed (not framework config) . . . 563
How to upgrade . . . 563
Parity . . . 563
v3.11.32 (2026-04-25) . . . 563
v3.11.13 (2026-04-16) . . . 564
v3.11.12 (2026-04-16) . . . 565
v3.11.11 (2026-04-16) . . . 566
v3.11.10 (2026-04-15) . . . 566
v3.11.9 (2026-04-15) . . . 567

v3.10.99 (2026-04-12) . . . 567
v3.10.97 (2026-04-11) . . . 568
v3.10.93 (2026-04-11) . . . 568
v3.10.92 (2026-04-10) . . . 568
v3.10.91 (2026-04-10) . . . 568
v3.10.90 (2026-04-09) . . . 569
v3.10.89 (2026-04-09) . . . 569
v3.10.88 (2026-04-09) . . . 569
v3.10.87 (2026-04-09) . . . 570
v3.10.86 (2026-04-09) . . . 570
v3.10.85 (2026-04-09) . . . 570
v3.10.84 (2026-04-09) . . . 570
v3.10.83 (2026-04-08) . . . 570
v3.10.70 (2026-04-06) . . . 571
v3.10.68 (2026-04-03) — Full Parity Release . . . 571
v3.10.67 (2026-04-03) . . . 571
v3.10.66 (2026-04-03) . . . 572
v3.10.65 (2026-04-03) . . . 572
v3.10.60 (2026-04-03) . . . 572
v3.10.57 (2026-04-02) . . . 572
v3.10.54 (2026-04-02) . . . 573
v3.10.48 — April 2, 2026 . . . 573
Bug Fixes . . . 573
v3.10.46 — April 1, 2026 . . . 573
Test Coverage . . . 573
v3.10.45 — April 1, 2026 . . . 574

Notes . . . 574
v3.10.44 — April 1, 2026 . . . 574
New Features . . . 574
Bug Fixes . . . 574
Test Coverage . . . 574
v3.10.40 — April 1, 2026 . . . 575
Bug Fixes . . . 575
v3.10.39 — April 1, 2026 . . . 575
New Features . . . 575
v3.10.38 — April 1, 2026 . . . 575
Code Metrics & Bubble Chart . . . 575
AI Context Installer . . . 575
Dashboard Improvements . . . 576
Cleanup . . . 576
v3.10.x — Previous Releases (March 28–31, 2026) . . . 576
Features . . . 576
Bug Fixes . . . 577
Firebird-Specific . . . 578
v3.9.x — QueryBuilder, Sessions, Path Injection (March 26–27, 2026) . . . 579
Features . . . 579
Breaking Changes . . . 580
Bug Fixes . . . 581
v3.8.x — Template Engine, Typed Params, Security (March 25–26, 2026) . . . 581
Features . . . 581
Breaking Changes . . . 582
v3.7.x — Template Auto-Serve, Firebird Migrations (March 25, 2026) . . . 583
Features . . . 583

v3.6.x — Architectural Parity (March 25, 2026) . . . 583
Features . . . 583
Bug Fixes . . . 583
v3.5.x — Bundled Frontend, Middleware (March 25, 2026) . . . 583
Features . . . 583
v3.4.x — Database, Auth, WebSocket, Uploads (March 24, 2026) . . . 584
Features . . . 584
Breaking Changes . . . 585
v3.3.x — Queue API, Field Mapping, Route Chaining (March 24, 2026) . . . 586
Features . . . 586
v3.2.x — Flexible Route Handlers (March 22, 2026) . . . 587
Features . . . 587
v3.1.x — Response Parity, Routing API (March 21–22, 2026) . . . 587
Features . . . 587
v3.0.0 — Initial Release (March 21, 2026) . . . 588
Features . . . 588
Pre-Release (rc.2–rc.5) . . . 589
Version Timeline . . . 590
Complete Feature List **591**
1. Routing and HTTP . . . 591
2. Templating . . . 592
3. Database and ORM . . . 593
4. Authentication and Sessions . . . 594
5. Background Processing . . . 594
6. Caching . . . 595
7. Communication . . . 595

8. Utilities and Services 596

9. Developer Experience 596

2. Feature Details 596

Routing and HTTP 596

Templating 597

Database and ORM 597

Authentication and Sessions 598

Background Processing 598

Caching 598

Communication 599

Utilities and Services 599

Developer Experience 600

3. Environment Variables Reference 600

4. Package Imports Reference 601

Getting Started with Tina4 Node.js

1. What Is Tina4 Node.js

Tina4 Node.js is a zero-dependency web framework. One npm package. It hands you routing, an ORM, a template engine, authentication, queues, WebSocket, and 70 other features. Node.js 22+ and TypeScript.

Truly zero runtime dependencies means no native C++ addons, no `node-gyp`, no platform-specific binaries. SQLite support uses Node's built-in `node:sqlite` module (available in Node 22+), so even database access requires nothing beyond what ships with Node.js itself.

It belongs to the Tina4 family -- four identical frameworks in Python, PHP, Ruby, and Node.js. Learn one, know all four. Same project structure. Same template syntax. Same CLI commands. Same `.env` variables.

Tina4 Node.js uses `camelCase` for method names (`fetchOne()`, `softDelete()`, `hasMany()`). JavaScript convention. Class names are `PascalCase`. Constants are `UPPER_SNAKE_CASE`.

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

---

2. Prerequisites and Installation

What You Need

- **Node.js 22 or later** -- check with:

```
node -v
```

You should see output like:

```
v22.0.0
```

Anything below 22 means you need to upgrade first. Node 22+ is required because Tina4 uses the built-in `node:sqlite` module for SQLite support, which is not available in earlier versions.

- **npm** -- Node's package manager. Check with:

```
npm -v
```

You should see:

```
10.2.4
```

npm ships with Node.js. If Node.js is installed, npm is too.

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

macOS (Homebrew):

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

Linux / macOS (install script):

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

Windows (PowerShell):

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

Verify the CLI:

```
tina4 --version
```

```
tina4 0.1.0
```

- **TypeScript and tsx** -- for running TypeScript directly:

```
npm install -g tsx
```

Verify:

```
tsx --version
```

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

The Tina4 CLI scaffolds a new project in one command:

```
tina4 init nodejs my-store
```

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

You should see:

```
Creating Tina4 project in ./my-store ...
  Detected language: Node.js (package.json)
  Created .env
  Created .env.example
  Created .gitignore
  Created src/routes/
  Created src/orm/
  Created migrations/
  Created src/seeds/
  Created src/templates/
  Created src/templates/errors/
  Created src/public/
  Created src/public/js/
  Created src/public/css/
  Created src/public/scss/
  Created src/public/images/
  Created src/public/icons/
  Created src/locales/
  Created data/
```

```
  Created logs/
  Created secrets/
  Created tests/

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

Install the Node.js dependencies:

```
cd my-store
npm install
```

```
added 1 package in 2s

1 package installed
```

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

Starting the Dev Server

```
tina4 serve
```

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

  Tina4 Node.js v3.10.3
  Server running at http://0.0.0.0:7148
  Debug mode: ON
  Database: sqlite:///data/app.db
  Press Ctrl+C to stop
```

Open your browser to `http://localhost:7148`. The Tina4 welcome page greets you.

Hit the health endpoint:

```
curl http://localhost:7148/health
```

```
{
  "status": "ok",
  "database": "connected",
  "uptime_seconds": 12,
  "version": "3.10.3",
  "framework": "tina4-nodejs"
}
```

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
■■■ package.json            # npm package definition
■■■ package-lock.json       # Locked dependency versions
■■■ tsconfig.json           # TypeScript configuration
■■■ app.ts                  # Application entry point
■■■ node_modules/           # npm packages (gitignored)
■■■ migrations/             # SQL migration files (project root)
■■■ src/
■   ■■■ routes/             # Your route handlers go here
■   ■■■ orm/                # Your ORM model classes go here
■   ■■■ seeds/              # Database seed files
■   ■■■ templates/          # Frond/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)
■■■ secrets/                # JWT keys (gitignored)
■■■ tests/                  # Your test files
    ■■■ run-all.ts          # Test runner
```

Key directories:

- `src/routes/` -- Every `.ts` file here is auto-loaded at startup. Drop your route definitions here. Organize into subdirectories if you want. Tina4 also supports file-based routing: a file at `src/routes/api/users/get.ts` maps to `GET /api/users` with zero configuration.
- `src/orm/` -- Every `.ts` 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 `res.html()` with a template.
- `src/public/` -- Files served directly. `src/public/images/logo.png` becomes `/images/logo.png`.
- `data/` -- The default SQLite database (`app.db`) lives here. Gitignored because databases do not belong in version control.

---

4. Your First Route

Time to build an API endpoint that returns a JSON greeting.

Explicit Route Registration

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

```
import { Router } from "tina4-nodejs";

Router.get("/api/greeting/{name}", async (req, res) => {
```

```
    const name = req.params.name;
    return res.json({
        message: `Hello, ${name}!`,
        timestamp: new Date().toISOString()
    });
});
```

Save the file. The dev server picks up the change. If live reload is off, restart with `tina4 serve`.

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

File-Based Routing Alternative

Create the file `src/routes/api/greeting/[name]/get.ts`:

```
export default async (req, res) => {
    const name = req.params.name;
    return res.json({
        message: `Hello, ${name}!`,
        timestamp: new Date().toISOString()
    });
};
```

Both approaches produce identical results. File-based routing maps the file path to the URL. Dynamic segments go in brackets (`[name]`).

Test It

Open your browser to:

```
http://localhost:7148/api/greeting/Alice
```

You should see:

```
{
  "message": "Hello, Alice!",
  "timestamp": "2026-03-22T14:30:00.000Z"
}
```

Or use curl:

```
curl http://localhost:7148/api/greeting/Alice
```

```
{"message":"Hello, Alice!","timestamp":"2026-03-22T14:30:00.000Z"}
```

Understanding What Happened

- You created a file in `src/routes/`. Tina4 discovered it at startup.
- `Router.get("/api/greeting/{name}", ...)` registered a GET route with a path parameter `{name}`.
- A request to `/api/greeting/Alice` hit the router. Pattern matched. Handler fired.
- `req.params.name` delivered the value `"Alice"` from the URL.
- `res.json(...)` serialized the object, set `Content-Type: application/json`, and returned `200 OK`.

Adding More HTTP Methods

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

```
import { Router } from "tina4-nodejs";

Router.get("/api/greeting/{name}", async (req, res) => {
    const name = req.params.name;
    return res.json({
        message: `Hello, ${name}!`,
        timestamp: new Date().toISOString()
    });
});

Router.post("/api/greeting", async (req, res) => {
    const name = req.body.name ?? "World";
    const language = req.body.language ?? "en";

    const greetings: Record<string, string> = {
        en: "Hello",
        es: "Hola",
        fr: "Bonjour",
        de: "Hallo",
        ja: "Konnichiwa"
    };

    const greeting = greetings[language] ?? greetings["en"];

    return res.status(201).json({
        message: `${greeting}, ${name}!`,
        language
    });
});
```

Test the POST endpoint:

```
curl -X POST http://localhost:7148/api/greeting \
  -H "Content-Type: application/json" \
  -d '{"name": "Carlos", "language": "es"}'
```

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

The HTTP status code is `201 Created` (set by `res.status(201)`).

---

5. Your First Template

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

Create a Base Layout

Create `src/templates/base.html`:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
```

```
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>{% block title %}My Store{% endblock %}</title>
    <link rel="stylesheet" href="/css/tina4.css">
    <style>
        body { font-family: -apple-system, BlinkMacSystemFont, "Segoe UI", Roboto, sans-serif; m
        .container { max-width: 960px; margin: 0 auto; padding: 20px; }
        .product-card { border: 1px solid #ddd; border-radius: 8px; padding: 16px; margin: 8px 0
        .product-card h3 { margin-top: 0; }
        .price { color: #2d8f2d; font-weight: bold; font-size: 1.2em; }
        .badge { display: inline-block; padding: 2px 8px; border-radius: 4px; font-size: 0.8em;
        .badge-success { background: #d4edda; color: #155724; }
        .badge-danger { background: #f8d7da; color: #721c24; }
        nav { background: #333; color: white; padding: 12px 20px; }
        nav a { color: white; text-decoration: none; margin-right: 16px; }
    </style>
</head>
<body>
    <nav>
        <a href="/">Home</a>
        <a href="/products">Products</a>
    </nav>
    <div class="container">
        {% block content %}{% endblock %}
    </div>
    <script src="/js/frond.js"></script>
</body>
</html>
```

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

Create a Product Listing Page

Create `src/templates/products.html`:

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

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

{% block content %}
    <h1>Our Products</h1>
    <p>Showing {{ products | length }} product{{ 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.inStock %}
                    <span class="badge badge-success">In Stock</span>
                {% else %}
                    <span class="badge badge-danger">Out of Stock</span>
                {% endif %}
            </div>
        {% endfor %}
    {% else %}
        <p>No products available at the moment.</p>
    {% endif %}
```

```
{% endblock %}
```

Create the Route That Renders the Template

Create `src/routes/pages.ts`:

```
import { Router } from "tina4-nodejs";

Router.get("/products", async (req, res) => {
    const products = [
        {
            name: "Wireless Keyboard",
            description: "Ergonomic wireless keyboard with backlit keys.",
            price: 79.99,
            inStock: true
        },
        {
            name: "USB-C Hub",
            description: "7-port USB-C hub with HDMI, SD card reader, and Ethernet.",
            price: 49.99,
            inStock: true
        },
        {
            name: "Monitor Stand",
            description: "Adjustable aluminum monitor stand with cable management.",
            price: 129.99,
            inStock: false
        },
        {
            name: "Mechanical Mouse",
            description: "High-precision wireless mouse with 16,000 DPI sensor.",
            price: 59.99,
            inStock: true
        }
    ];

    return res.html("products.html", { products });
});
```

File-based routing works here too. Create `src/routes/products/get.ts`:

```
export const template = "products.html";

export default async (req, res) => {
    return {
        products: [
            { name: "Wireless Keyboard", description: "Ergonomic wireless keyboard.", price: 79.
            // ... more products
        ]
    };
};
```

Export a `template` constant and Tina4 renders it with the data returned from the handler.

See It in the Browser

Open `http://localhost:7148/products`. You should see:

- A dark navigation bar at the top with "Home" and "Products" links

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

How Template Rendering Works

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

About tina4css

The `tina4.css` file in the base template is Tina4's built-in CSS utility framework. Layout utilities, typography, and common UI patterns -- without Bootstrap or Tailwind. Auto-provided at scaffolding time. No separate download needed.

---

6. Understanding .env

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

```
TINA4_DEBUG=true
```

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

The important defaults for development:

Variable	Default Value	What It Means
`TINA4_PORT`	`7148`	Server runs on port 7148
`TINA4_DATABASE_URL`	`sqlite:///data/app.db`	SQLite database in the `data/` directory
`TINA4_LOG_LEVEL`	`ALL`	All log messages are output

`CORS_ORIGINS`	`*`	All origins allowed (fine for development)
`TINA4_RATE_LIMIT`	`60`	60 requests per minute per IP

Log levels control how much output Tina4 produces:

Level	Behaviour
`ALL` / `DEBUG`	Full verbose output. DevReload active (live-reload, error overlay).
`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 (`Ctrl+C`, then `tina4 serve`). It now runs on port 8080.

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

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

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

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

---

7. The Dev Dashboard

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

Restart the server and navigate to:

```
http://localhost:7148/__dev
```

The dashboard reveals:

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

The dev dashboard shows you what your application is doing without littering your code with `console.log` statements.

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

- Request details (method, URL, duration)
- Database queries executed (with timing)
- Template renders (with timing)
- Session data
- Recent log entries

This overlay lives only in debug mode. Production never sees it.

---

8. The app.ts Entry Point

The `app.ts` file is the entry point:

```
import { startServer } from "tina4-nodejs";

startServer();
```

That is the entire file. Tina4 discovers routes, models, and templates from the `src/` directory. You register nothing manually.

The standard way to start the server is with the CLI:

```
tina4 serve

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

```

```
Tina4 Node.js v3.10.3
Server running at http://0.0.0.0:7148
```

The CLI adds live reload and other development features. For direct Node.js execution (advanced usage), see Chapter 31: CLI.

---

9. Manual Setup (No CLI)

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

Step 1: Initialize and Install

```
npm init -y
npm install tina4-nodejs typescript tsx
```

Step 2: Create `app.ts`

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

```
import { startServer } from "@tina4/core";

const port = parseInt(process.env.PORT || "7148", 10);
const host = process.env.HOST || "0.0.0.0";
startServer({ port, host });
```

Three lines of real code. `startServer` boots the framework, scans for routes, and starts listening.

Step 3: Create `tsconfig.json`

```
{
  "compilerOptions": {
    "target": "ES2022",
    "module": "ESNext",
    "moduleResolution": "bundler",
    "strict": true,
    "esModuleInterop": true,
    "outDir": "dist"
  },
  "include": ["src/**/*", "app.ts"]
}
```

Step 4: Create the Folder Structure

Tina4 expects this layout:

```
my-project/
■■■ app.ts
■■■ tsconfig.json
■■■ package.json
■■■ .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 5: Create `.env`

```
TINA4_DEBUG=true
```

Step 6: Run It

```
tina4 serve
```

The server starts on `http://localhost:7148`. 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 Node.js 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* `npx tsx app.ts` *directly.*

---

10. Request & Response Fundamentals

Before jumping into the exercises, let's consolidate how route handlers work in Tina4 Node.js. Every handler receives two arguments: `req` (what the client sent) and `res` (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 `req.query`:

```
// URL: /api/search?q=laptop&page=2
req.query.q          // "laptop"
req.query.page       // "2" (always a string)
req.query.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 `req.params`:

```
import { Router } from "tina4-nodejs";

Router.get("/users/{id:int}/posts/{slug}", async (req, res) => {
    const id = req.params.id;      // 5 (number, because of :int)
    const slug = req.params.slug;  // "hello-world" (string)
    return res.json({ user_id: id, slug: slug });
});
```

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

Reading the Request Body

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

```
Router.post("/api/items", async (req, res) => {
    const name = req.body.name ?? "";
    const price = req.body.price ?? 0;
    return res.json({ received_name: name, received_price: price });
});
```

Reading Headers

Headers are available as an object. In Node.js, header names are normalized to lowercase:

```
const contentType = req.headers["content-type"] ?? "not set";
const authToken = req.headers["authorization"] ?? "";
const custom = req.headers["x-custom-header"] ?? "";
```

Sending JSON Responses

`res.json()` converts an object to JSON and sets the correct `Content-Type`. Chain with `res.status()` for a custom status code:

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

Sending HTML / Template Responses

`res.html()` renders a Frond template from `src/templates/` when given a filename and data:

```
return res.html("products.html", { products: productList, title: "Our Products" });
```

For raw HTML without a template file:

```
return res.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.ts`:

```
import { Router } from "tina4-nodejs";

// In-memory data store
const 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 }
];

Router.get("/api/books", async (req, res) => {
    // List all books. Supports ?author= filter and ?sort=year.
    const author = (req.query.author ?? "") as string;
    const sortBy = (req.query.sort ?? "") as string;

    let result = [...books];

    // Filter by author if the query param is present
    if (author) {
        result = result.filter(b =>
            b.author.toLowerCase().includes(author.toLowerCase())
        );
    }

    // Sort by year if requested
    if (sortBy === "year") {
        result.sort((a, b) => a.year - b.year);
    }

    return res.json({ books: result, count: result.length });
});

Router.get("/api/books/{id:int}", async (req, res) => {
    // Get a single book by ID. Returns 404 if not found.
    const id = req.params.id;
    const book = books.find(b => b.id === id);

    if (!book) {
        return res.status(404).json({ error: `Book with id ${id} not found` });
    }

    return res.json(book);
});

Router.post("/api/books", async (req, res) => {
    // Create a new book from the JSON body. Returns 201 on success.
    const title = req.body.title ?? "";
    const author = req.body.author ?? "";
    const year = req.body.year ?? 0;

    if (!title || !author) {
        return res.status(400).json({ error: "title and author are required" });
    }

    const newBook = {
        id: Math.max(...books.map(b => b.id)) + 1,
        title,
        author,
        year
    };
```

```
    books.push(newBook);

    return res.status(201).json(newBook);
});
```

Test it:

```
# List all books
curl http://localhost:7148/api/books

# Filter by author
curl "http://localhost:7148/api/books?author=gibson"

# Sort by year
curl "http://localhost:7148/api/books?sort=year"

# Get a single book
curl http://localhost:7148/api/books/2

# Get a book that does not exist (returns 404)
curl http://localhost:7148/api/books/99

# Create a new book
curl -X POST http://localhost:7148/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.

---

11. Exercise: Greeting API + Product List Template

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

Exercise Part A: Greeting API

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

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

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

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

- 21:00 - 4:59 = "night"

Test your endpoint with:

```
curl "http://localhost:7148/api/greet?name=Sarah"
curl "http://localhost:7148/api/greet"
```

Exercise Part B: Product List Page

Create a page at `GET /store` that:

- Displays a list of at least 5 products (hardcoded for now)
- Each product has: name, category, price, and a boolean `featured` flag
- Featured products should be highlighted (different background color, border, or badge)
- The page should show the total number of products and the number of featured products
- Use template inheritance -- create a layout template and a page template that extends it
- Include `tina4.css` and `frond.js`

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

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

---

12. Solutions

Solution A: Greeting API

Create `src/routes/greet.ts`:

```
import { Router } from "tina4-nodejs";

Router.get("/api/greet", async (req, res) => {
    const name = req.query.name ?? "Stranger";
    const hour = new Date().getHours();

    let timeOfDay: string;
    if (hour >= 5 && hour < 12) {
```

```
        timeOfDay = "morning";
    } else if (hour >= 12 && hour < 17) {
        timeOfDay = "afternoon";
    } else if (hour >= 17 && hour < 21) {
        timeOfDay = "evening";
    } else {
        timeOfDay = "night";
    }

    return res.json({
        greeting: `Welcome, ${name}!`,
        time_of_day: timeOfDay
    });
});
```

Test:

```
curl "http://localhost:7148/api/greet?name=Sarah"

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

curl "http://localhost:7148/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.ts`:

```
import { Router } from "tina4-nodejs";

Router.get("/store", async (req, res) => {
    const products = [
        { name: "Espresso Machine", category: "Kitchen", price: 299.99, featured: true },
        { name: "Yoga Mat", category: "Fitness", price: 29.99, featured: false },
        { name: "Standing Desk", category: "Office", price: 549.99, featured: true },
        { name: "Noise-Canceling Headphones", category: "Electronics", price: 199.99, featured:
        { name: "Water Bottle", category: "Fitness", price: 24.99, featured: false }
    ];

    const featuredCount = products.filter(p => p.featured).length;

    return res.html("store.html", {
        products,
        featured_count: featuredCount
    });
});
```

Open `http://localhost:7148/store` in your browser. You should see:

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

---

13. Gotchas

1. File not auto-discovered

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

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

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

2. "Module not found" errors

Problem: `Cannot find module 'tina4-nodejs'` or similar.

Cause: Missing npm install or incorrect import path.

Fix: Run `npm install` in your project root. Make sure your import uses the exact package name: `import { Router } from "tina4-nodejs"`.

3. JSON response shows HTML

Problem: Your JSON endpoint returns HTML instead of JSON.

Cause: You returned a string instead of using `res.json()`. A plain string tells Tina4 to treat it as HTML.

Fix: Use `res.json(data)` for JSON endpoints. `console.log()` is not a response mechanism.

4. Template not found

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

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

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

5. Port already in use

Problem: `Error: Address already in use (port 7148)`

Cause: Another process is occupying port 7148.

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

```
TINA4_PORT=8080
```

Or use the CLI flag: `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 the browser (`Ctrl+Shift+R` or `Cmd+Shift+R`). If that fails, restart the dev server with `Ctrl+C` and `tina4 serve`.

7. .env not loaded

Problem: Environment variables have no effect.

Cause: The `.env` file must be at the project root (same directory as `package.json`).

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

Routing

1. How Routing Works in Tina4

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

In Tina4 Node.js, you define routes in TypeScript files inside `src/routes/`. Every `.ts` file in that directory -- and its subdirectories -- is auto-loaded when the server starts. No registration files. No central config. Drop a file in. It works.

Tina4 supports two routing styles: **explicit registration** with the `Router` class, and **file-based routing** where the file path becomes the URL.

The simplest possible route using explicit registration:

```
import { Router } from "tina4-nodejs";

Router.get("/hello", async (req, res) => {
    return res.json({ message: "Hello, World!" });
});
```

Save that as `src/routes/hello.ts`, start the server with `tina4 serve`, and visit `http://localhost:7148/hello`:

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

The same thing using file-based routing. Create `src/routes/hello/get.ts`:

```
export default async (req, res) => {
    return res.json({ message: "Hello, World!" });
};
```

Both approaches produce identical results. Use whichever you prefer -- or mix them.

---

2. HTTP Methods

Tina4 supports all five standard HTTP methods. Each one has a static method on the `Router` class:

```
import { Router } from "tina4-nodejs";

Router.get("/products", async (req, res) => {
    return res.json({ action: "list all products" });
});

Router.post("/products", async (req, res) => {
    return res.status(201).json({ action: "create a product" });
});

Router.put("/products/{id}", async (req, res) => {
    const id = req.params.id;
    return res.json({ action: `replace product ${id}` });
});
```

```
Router.patch("/products/{id}", async (req, res) => {
    const id = req.params.id;
    return res.json({ action: `update product ${id}` });
});

Router.delete("/products/{id}", async (req, res) => {
    const id = req.params.id;
    return res.json({ action: `delete product ${id}` });
});
```

For file-based routing, the HTTP method is the filename:

```
src/routes/products/get.ts      → GET /products
src/routes/products/post.ts     → POST /products
src/routes/products/[id]/put.ts → PUT /products/{id}
```

Test each one:

```
curl http://localhost:7148/products
```

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

```
curl -X POST http://localhost:7148/products \
  -H "Content-Type: application/json" \
  -d '{"name": "Widget"}'
```

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

```
curl -X PUT http://localhost:7148/products/42
```

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

```
curl -X PATCH http://localhost:7148/products/42
```

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

```
curl -X DELETE http://localhost:7148/products/42
```

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

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

---

3. Path Parameters

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

```
import { Router } from "tina4-nodejs";

Router.get("/users/{id}/posts/{postId}", async (req, res) => {
    const userId = req.params.id;
    const postId = req.params.postId;

    return res.json({
        user_id: userId,
        post_id: postId
    });
});
```

For file-based routing, wrap the parameter name in brackets:

```
src/routes/users/[id]/posts/[postId]/get.ts
```

```
curl http://localhost:7148/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 numbers. For example, requesting `/users/42/posts/99` will give you `req.params.id` as the number `42` and `req.params.postId` as the number `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 by adding a colon and the type after the parameter name:

```
import { Router } from "tina4-nodejs";

Router.get("/orders/{id:int}", async (req, res) => {
    const id = req.params.id; // This is now a number
    return res.json({
        order_id: id,
        type: typeof id
    });
});
```

```
curl http://localhost:7148/orders/42
```

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

Pass a non-integer value and the route refuses to match. You get a 404:

```
curl http://localhost:7148/orders/abc
```

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

Supported types:

Type	Matches	Auto-cast	Example
`int`	Digits only	Number	`{id:int}` matches `42` but not `abc`
`float`	Decimal numbers	Number	`{price:float}` matches `19.99`
`path`	All remaining path segments (catch-all)	String	`{slug:path}` matches `docs/api/auth`
`alpha`	Letters only	String	`{slug:alpha}` matches `hello` but not `hello123`

`alphanumeric`	Letters and digits	String	`{code:alphanumeric}` matches `abc123`

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:

```
import { Router } from "tina4-nodejs";

// Integer parameter -- only digits match, auto-cast to number
Router.get("/products/{id:int}", async (req, res) => {
    const id = req.params.id; // number, e.g. 42
    return res.json({
        product_id: id,
        type: typeof id
    });
});

// Float parameter -- decimal numbers, auto-cast to number
Router.get("/products/{id:int}/price/{price:float}", async (req, res) => {
    const id = req.params.id;
    const price = req.params.price;
    return res.json({
        product_id: id,
        price,
        type: typeof price
    });
});

// Path parameter -- catch-all, captures remaining segments as a string
Router.get("/files/{filepath:path}", async (req, res) => {
    const filepath = req.params.filepath;
    // filepath could be "images/photos/cat.jpg"
    return res.json({
        filepath,
        type: typeof filepath
    });
});
```

```
# Integer route -- matches digits, returns a number
curl http://localhost:7148/products/42
```

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

```
# Integer route -- non-integer gives a 404
curl http://localhost:7148/products/abc
```

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

```
# Path catch-all -- captures everything after /files/
curl http://localhost:7148/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 the key-value pairs after the `?` in a URL. Access them via `req.query`:

```
import { Router } from "tina4-nodejs";

Router.get("/search", async (req, res) => {
    const q = req.query.q ?? "";
    const page = parseInt(req.query.page ?? "1", 10);
    const limit = parseInt(req.query.limit ?? "10", 10);

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

```
curl "http://localhost:7148/search?q=keyboard&page=2&limit=20"
```

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

A missing query parameter yields `undefined` from `req.query.key`. The nullish coalescing operator (`??`) provides defaults.

---

5. Route Groups

A set of routes sharing a common prefix belongs in a group. `Router.group()` eliminates repetition:

```
import { Router } from "tina4-nodejs";

Router.group("/api/v1", (group) => {

    group.get("/users", async (req, res) => {
        return res.json({ users: [] });
    });

    group.get("/users/{id:int}", async (req, res) => {
        const id = req.params.id;
        return res.json({ user: { id, name: "Alice" } });
    });

    group.post("/users", async (req, res) => {
        return res.status(201).json({ created: true });
    });

    group.get("/products", async (req, res) => {
        return res.json({ products: [] });
    });
```

```
});
```

The 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:7148/api/v1/users
```

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

Groups nest:

```
import { Router } from "tina4-nodejs";

Router.group("/api", (api) => {
    api.group("/v1", (v1) => {
        v1.get("/status", async (req, res) => {
            return res.json({ version: "1.0" });
        });
    });

    api.group("/v2", (v2) => {
        v2.get("/status", async (req, res) => {
            return res.json({ version: "2.0" });
        });
    });
});
```

```
curl http://localhost:7148/api/v1/status
```

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

```
curl http://localhost:7148/api/v2/status
```

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

---

6. Middleware on Routes

Middleware is code that runs before (or after) your route handler. Authentication. Logging. Rate limiting. Input validation. Anything that belongs on multiple routes.

Middleware on a Single Route

Pass middleware as the third argument to any route method:

```
import { Router } from "tina4-nodejs";

function logRequest(req, res, next) {
    const start = Date.now();
    console.log(`[${new Date().toISOString()}] ${req.method} ${req.path}`);
    next();
    const duration = Date.now() - start;
    console.log(`  Completed in ${duration}ms`);
}

Router.get("/api/data", async (req, res) => {
    return res.json({ data: [1, 2, 3] });
}, [logRequest]);
```

The middleware function receives `req`, `res`, and `next`. Call `next()` to continue to the next middleware or route handler. Skip the call and the handler never runs -- the chain stops. A locked gate for unauthorized requests.

Blocking Middleware

Middleware that checks for an API key:

```
import { Router } from "tina4-nodejs";

function requireApiKey(req, res, next) {
    const apiKey = req.headers["x-api-key"] ?? "";

    if (apiKey !== "my-secret-key") {
        res({ error: "Invalid API key" }, 401);
        return;
    }

    next();
}

Router.get("/api/secret", async (req, res) => {
    return res.json({ secret: "The answer is 42" });
}, [requireApiKey]);
```

```
curl http://localhost:7148/api/secret
```

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

```
curl http://localhost:7148/api/secret -H "X-API-Key: my-secret-key"
```

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

Middleware on a Group

Apply middleware to an entire group:

```
import { Router } from "tina4-nodejs";

Router.group("/api/admin", (group) => {

    group.get("/dashboard", async (req, res) => {
        return res.json({ page: "admin dashboard" });
    });

    group.get("/users", async (req, res) => {
        return res.json({ page: "user management" });
    });

}, [requireAuth]);
```

Multiple Middleware

Chain multiple middleware by passing an array:

```
Router.get("/api/important", async (req, res) => {
    return res.json({ data: "important stuff" });
}, [logRequest, requireApiKey, requireAuth]);
```

Middleware runs in order: `logRequest` first, then `requireApiKey`, then `requireAuth`, then the route handler.

---

7. Route Decorators: @noauth and @secured

Tina4 provides two special decorators for controlling authentication on routes.

@noauth -- Public Routes

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

```
import { Router } from "tina4-nodejs";

/**
 * @noauth
 */
Router.get("/api/public/info", async (req, res) => {
    return res.json({
        app: "My Store",
        version: "1.0.0"
    });
});
```

The `@noauth` decorator tells Tina4 to skip authentication checks for this route, even with global auth middleware configured in `.env`.

@secured -- Protected GET Routes

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

```
import { Router } from "tina4-nodejs";

/**
 * @secured
 */
Router.get("/api/profile", async (req, res) => {
    return res.json({
        user: req.user
    });
});
```

By default, `POST`, `PUT`, `PATCH`, and `DELETE` routes are considered secured. `GET` routes are not -- they are public unless you add `@secured`.

---

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:

```
import { Router } from "tina4-nodejs";

Router.get("/api/account", async (req, res) => {
    return res.json({ account: req.user });
}).secure();
```

```
curl http://localhost:7148/api/account
# 401 Unauthorized

curl http://localhost:7148/api/account -H "Authorization: Bearer eyJhbGci..."
# 200 OK
```

.cache()

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

```
Router.get("/api/catalog", async (req, res) => {
    // Expensive database query
    return res.json({ products });
}).cache();
```

Chaining Both

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

```
Router.get("/api/data", async (req, res) => {
    return res.json({ data });
}).secure().cache();
```

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

---

9. Wildcard and Catch-All Routes

Wildcard Routes

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

```
import { Router } from "tina4-nodejs";

Router.get("/docs/*", async (req, res) => {
    const path = req.params["*"] ?? "";
    return res.json({
        section: "docs",
        path
    });
});
```

```
curl http://localhost:7148/docs/getting-started
```

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

curl http://localhost:7148/docs/api/authentication/jwt

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

Catch-All Route (Custom 404)

Register a catch-all to handle any unmatched URL:

```
import { Router } from "tina4-nodejs";

Router.get("/*", async (req, res) => {
    return res.status(404).json({
        error: "Page not found",
        path: req.path
    });
});
```

Define this route last. Tina4 matches routes in registration order -- first match wins.

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

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

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

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

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

---

10. Route Listing via CLI

As your application grows, you need visibility into all registered routes. The CLI provides it:

```
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          requireAuth         public
GET      /api/admin/users              requireAuth         public
GET      /api/public/info              -                   @noauth
GET      /api/profile                  -                   @secured
GET      /search                       -                   public
```

```
GET      /docs/*                    -                  public
```

Filter by method or search for a path pattern:

```
tina4 routes --method POST
tina4 routes --filter users
```

---

11. Organizing Route Files

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

Pattern 1: One File Per Resource

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

Pattern 2: File-Based Routing by Feature

```
src/routes/
■■■ api/
■   ■■■ products/
■   ■   ■■■ get.ts          # GET /api/products
■   ■   ■■■ post.ts         # POST /api/products
■   ■   ■■■ [id]/
■   ■       ■■■ get.ts      # GET /api/products/{id}
■   ■       ■■■ put.ts      # PUT /api/products/{id}
■   ■       ■■■ delete.ts   # DELETE /api/products/{id}
■   ■■■ users/
■       ■■■ get.ts
■       ■■■ post.ts
■■■ admin/
■   ■■■ dashboard/get.ts
■   ■■■ settings/get.ts
■■■ pages/
    ■■■ home/get.ts
    ■■■ about/get.ts
```

Both patterns work identically. Choose whichever keeps your project navigable.

---

12. Exercise: Build a Full CRUD API for Products

Build a complete REST API for managing products. All data lives in a TypeScript array (no database yet -- that comes in Chapter 5).

Requirements

Create a file `src/routes/product-api.ts` with the following routes:

Method	Path	Description

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

Each product has: `id` (number), `name` (string), `category` (string), `price` (number), `inStock` (boolean).

Start with this seed data:

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

Test with:

```
# List all
curl http://localhost:7148/api/products

# Filter by category
curl "http://localhost:7148/api/products?category=Fitness"

# Get one
curl http://localhost:7148/api/products/3

# Create
curl -X POST http://localhost:7148/api/products \
  -H "Content-Type: application/json" \
  -d '{"name": "Desk Lamp", "category": "Office", "price": 39.99, "inStock": true}'

# Update
curl -X PUT http://localhost:7148/api/products/3 \
  -H "Content-Type: application/json" \
  -d '{"name": "Burr Coffee Grinder", "category": "Kitchen", "price": 59.99, "inStock": true}'

# Delete
curl -X DELETE http://localhost:7148/api/products/3

# Not found
curl http://localhost:7148/api/products/999
```

---

13. Solution

Create `src/routes/product-api.ts`:

```
import { Router } from "tina4-nodejs";

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

let nextId = 6;

// List all products, optionally filter by category
Router.get("/api/products", async (req, res) => {
    const category = req.query.category ?? null;

    if (category !== null) {
        const filtered = products.filter(
            p => p.category.toLowerCase() === String(category).toLowerCase()
        );
        return res.json({ products: filtered, count: filtered.length });
    }

    return res.json({ products, count: products.length });
});

// Get a single product by ID
Router.get("/api/products/{id:int}", async (req, res) => {
    const id = req.params.id;
    const product = products.find(p => p.id === id);

    if (!product) {
        return res.status(404).json({ error: "Product not found", id });
    }

    return res.json(product);
});

// Create a new product
Router.post("/api/products", async (req, res) => {
    const body = req.body;

    if (!body.name) {
        return res.status(400).json({ error: "Name is required" });
    }

    const product = {
        id: nextId++,
        name: body.name,
        category: body.category ?? "Uncategorized",
        price: parseFloat(body.price ?? 0),
        inStock: Boolean(body.inStock ?? true)
    };
```

```
    products.push(product);

    return res.status(201).json(product);
});

// Replace a product
Router.put("/api/products/{id:int}", async (req, res) => {
    const id = req.params.id;
    const body = req.body;
    const index = products.findIndex(p => p.id === id);

    if (index === -1) {
        return res.status(404).json({ error: "Product not found", id });
    }

    products[index] = {
        id,
        name: body.name ?? products[index].name,
        category: body.category ?? products[index].category,
        price: parseFloat(body.price ?? products[index].price),
        inStock: Boolean(body.inStock ?? products[index].inStock)
    };

    return res.json(products[index]);
});

// Delete a product
Router.delete("/api/products/{id:int}", async (req, res) => {
    const id = req.params.id;
    const index = products.findIndex(p => p.id === id);

    if (index === -1) {
        return res.status(404).json({ error: "Product not found", id });
    }

    products.splice(index, 1);

    return res.status(204).json(null);
});
```

Expected output for the test commands:

List all:

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

Filter by category:

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

Create (Status: `201 Created`):

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

Not found (Status: `404 Not Found`):

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

---

14. Gotchas

1. Trailing Slashes Matter

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

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

Fix: Pick one convention and stick with it. 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}` gives wrong results -- both parameters share the same name.

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

Fix: Use distinct names: `/users/{userId}/posts/{postId}`.

3. Method Conflicts

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

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

Fix: Use typed parameters to disambiguate: `Router.get("/items/{id:int}", ...)` matches integers only, leaving `/items/export` free for the other route.

4. Route Handler Must Return a Response

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

Cause: You forgot the `return` statement. Without `return`, the handler produces `undefined` and Tina4 has nothing to send back.

Fix: Every handler must `return res.json(...)` or `return res.html(...)`.

5. Async Handlers

Problem: Your handler calls an async function but the response returns before it completes.

Cause: You forgot to `await` the async operation. The handler returns before the work finishes.

Fix: `await` all async operations inside handlers. All Tina4 route handlers should be `async` functions.

6. Middleware Must Be Passed as Function References in an Array

Problem: Passing a middleware function name as a string causes an error.

Cause: Tina4 expects middleware as an array of function references, not as strings.

Fix: Pass middleware as an array of function references: `[myMiddleware]`, not `"myMiddleware"`.

---

7. Group Prefix Must Start with a Slash

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

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

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

Request & Response

1. The Two Objects You Always Get

Every route handler in Tina4 receives two arguments: `req` and `res`. The request tells you what the client sent. The response is how you talk back. Together they are the entire HTTP conversation.

```
import { Router } from "tina4-nodejs";

Router.get("/echo", async (req, res) => {
    return res.json({
        method: req.method,
        path: req.path,
        your_ip: req.ip
    });
});
```

```
curl http://localhost:7148/echo
```

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

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

---

2. The Request Object

The `req` object gives you everything the client sent. Here is the complete inventory.

method

The HTTP method as an uppercase string: `"GET"`, `"POST"`, `"PUT"`, `"PATCH"`, or `"DELETE"`.

```
req.method // "GET"
```

path

The URL path without query parameters:

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

params

Path parameters from the URL pattern (see Chapter 2):

```
// Route: /users/{id}/posts/{postId}
// Request: /users/5/posts/42
req.params.id     // "5" (or 5 if typed as :id:int)
req.params.postId // "42"
```

query

Query string parameters as an object:

```
// Request: /search?q=keyboard&page=2&sort=price
req.query.q    // "keyboard"
```

```
req.query.page // "2"
req.query.sort // "price"
```

body

The parsed request body. JSON requests produce an object. Form submissions contain form fields:

```
// POST with {"name": "Widget", "price": 9.99}
req.body.name  // "Widget"
req.body.price // 9.99
```

headers

Request headers as an object. Header names are normalized to lowercase:

```
req.headers["content-type"]  // "application/json"
req.headers["authorization"] // "Bearer eyJhbGci..."
req.headers["x-custom"]      // "my-value"
```

ip

The client's IP address:

```
req.ip // "127.0.0.1"
```

Tina4 respects `X-Forwarded-For` and `X-Real-IP` headers when behind a reverse proxy.

cookies

Cookies sent by the client:

```
req.cookies.session_id  // "abc123"
req.cookies.preferences // "dark-mode"
```

files

Uploaded files (covered in detail in section 7):

```
req.files.avatar // File object with name, type, size, tmpPath
```

Inspecting the Full Request

A route that dumps everything:

```
import { Router } from "tina4-nodejs";

Router.post("/debug/request", async (req, res) => {
    return res.json({
        method: req.method,
        path: req.path,
        params: req.params,
        query: req.query,
        body: req.body,
        headers: req.headers,
        ip: req.ip,
        cookies: req.cookies
    });
});
```

```
curl -X POST "http://localhost:7148/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:7148",
    "user-agent": "curl/8.4.0",
    "accept": "*/*",
    "content-length": "16"
  },
  "ip": "127.0.0.1",
  "cookies": {}
}
```

---

3. The Response Object

The `res` object is your toolkit for sending data back to the client. Every method returns the response, so you can chain calls.

json() -- JSON Response

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

```
return res.json({ name: "Alice", age: 30 });
```

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

Chain with `status()` for a custom status code:

```
return res.status(201).json({ id: 7, name: "Widget" });
```

This returns `201 Created` with the JSON body.

html() -- Template or Raw HTML Response

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

```
return res.html("products.html", {
    products,
    title: "Our Products"
});
```

Or return raw HTML:

```
return res.html("<h1>Hello</h1><p>This is HTML.</p>");
```

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

text() -- Plain Text Response

Return plain text:

```
return res.text("Just a plain string.");
```

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

redirect() -- Redirect Response

Send the client to a different URL:

```
return res.redirect("/login");
```

This sends a `302 Found` redirect by default. Pass a different status code for permanent redirects:

```
return res.redirect("/new-location", 301);
```

file() -- File Download Response

Send a file to the client for download:

```
return res.file("/path/to/report.pdf");
```

Tina4 sets the appropriate `Content-Type` based on the file extension and adds a `Content-Disposition` header so the browser downloads the file.

Set a custom filename:

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

---

4. Status Codes

Every response method chains with `status()`. The most common ones:

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

403	Forbidden	Authenticated but not allowed.
404	Not Found	Resource does not exist.
409	Conflict	Duplicate or conflicting data.
422	Unprocessable Entity	Valid JSON but fails business rules.
500	Internal Server Error	Something broke on the server.

```
return res.status(201).json({ id: 7, created: true });
```

---

5. Custom Headers

Set response headers with the `header()` method:

```
Router.get("/api/data", async (req, res) => {
    return res
        .header("X-Request-Id", crypto.randomUUID())
        .header("X-Rate-Limit-Remaining", "57")
        .header("Cache-Control", "no-cache")
        .json({ data: [1, 2, 3] });
});
```

CORS Headers

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

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

---

6. Cookies

Set cookies on the response:

```
Router.post("/login", async (req, res) => {
    return res
        .cookie("session_id", "abc123xyz", {
            httpOnly: true,
            secure: true,
            sameSite: "Strict",
            maxAge: 3600,
            path: "/"
        })
        .json({ message: "Logged in" });
});
```

Read cookies from the request:

```
Router.get("/profile", async (req, res) => {
```

```
    const sessionId = req.cookies.session_id ?? null;

    if (sessionId === null) {
        return res.status(401).json({ error: "Not logged in" });
    }

    return res.json({ session: sessionId });
});
```

Delete a cookie by setting its `maxAge` to `0`:

```
return res
    .cookie("session_id", "", { maxAge: 0, path: "/" })
    .json({ message: "Logged out" });
```

---

7. File Uploads

Uploaded files arrive via `req.files`. Each file is an object with metadata and a temporary path.

Multiple File Uploads

Handle multiple files from a single form field:

```
Router.post("/api/gallery", async (req, res) => {
    const files = req.files?.photos;

    if (!files) {
        return res.status(400).json({ error: "No files uploaded" });
    }

    // files is a dict keyed by field name — multiple files under one name become an array
    const fileList = Object.values(files).flat();

    const results = [];
    for (const file of fileList) {
        const ext = extname(file.name);
        const savedName = `gallery_${randomUUID()}${ext}`;
        const dest = join(process.cwd(), "src/public/uploads", savedName);
        await rename(file.tmpPath, dest);
        results.push({
            original: file.name,
            saved: savedName,
            size: file.size,
            url: `/uploads/${savedName}`,
        });
    }

    return res.status(201).json({ uploaded: results, count: results.length });
});
```

```
curl -X POST http://localhost:7148/api/gallery \
  -F "photos=@/path/to/photo1.jpg" \
  -F "photos=@/path/to/photo2.jpg" \
  -F "photos=@/path/to/photo3.jpg"
```

Handling a Single File Upload

```
import { Router } from "tina4-nodejs";

Router.post("/api/upload", async (req, res) => {
    if (!req.files?.image) {
        return res.status(400).json({ error: "No file uploaded" });
    }

    const file = req.files.image;

    return res.json({
        name: file.name,
        type: file.type,
        size: file.size,
        tmp_path: file.tmpPath
    });
});
```

```
curl -X POST http://localhost:7148/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

```
import { Router } from "tina4-nodejs";
import { rename, mkdir } from "fs/promises";
import { existsSync } from "fs";
import { join, extname } from "path";
import { randomUUID } from "crypto";

Router.post("/api/upload", async (req, res) => {
    if (!req.files?.image) {
        return res.status(400).json({ error: "No file uploaded" });
    }

    const file = req.files.image;

    // Validate file type
    const allowedTypes = ["image/jpeg", "image/png", "image/gif", "image/webp"];
    if (!allowedTypes.includes(file.type)) {
        return res.status(400).json({ error: "Invalid file type. Allowed: JPEG, PNG, GIF, WebP"
    }

    // Validate file size (max 5MB)
    const maxSize = 5 * 1024 * 1024;
    if (file.size > maxSize) {
        return res.status(400).json({ error: "File too large. Maximum size: 5MB" });
    }

    // Generate a unique filename
    const ext = extname(file.name);
    const filename = `img_${randomUUID()}${ext}`;
    const uploadDir = join(process.cwd(), "src/public/uploads");
    const destination = join(uploadDir, filename);

    if (!existsSync(uploadDir)) {
        await mkdir(uploadDir, { recursive: true });
    }

    await rename(file.tmpPath, destination);

    return res.status(201).json({
        message: "File uploaded successfully",
        filename,
        url: `/uploads/${filename}`,
        size: file.size
    });
});
```

---

8. File Downloads

Send files to the client using `res.file()`:

```
import { Router } from "tina4-nodejs";
import { existsSync } from "fs";
import { join } from "path";

Router.get("/api/reports/{filename}", async (req, res) => {
    const filename = req.params.filename;
```

```
    const filepath = join(process.cwd(), "data/reports", filename);

    if (!existsSync(filepath)) {
        return res.status(404).json({ error: "Report not found" });
    }

    return res.file(filepath);
});
```

To force a specific download filename:

```
return res.file(filepath, "Q1-2026-Sales-Report.pdf");
```

---

9. XML Responses

Return XML when your API needs to serve legacy clients or SOAP integrations:

```
Router.get("/api/products/{id:int}.xml", async (req, res) => {
    const product = { id: req.params.id, name: "Wireless Keyboard", price: 79.99 };

    const xml = `<?xml version="1.0" encoding="UTF-8"?>
<product>
    <id>${product.id}</id>
    <name>${product.name}</name>
    <price>${product.price}</price>
</product>`;

    return res.header("Content-Type", "application/xml").text(xml);
});
```

---

10. Content Negotiation

Check the `Accept` header to return different formats from the same endpoint:

```
import { Router } from "tina4-nodejs";

Router.get("/api/products/{id:int}", async (req, res) => {
    const id = req.params.id;
    const product = { id, name: "Wireless Keyboard", price: 79.99 };

    const accept = req.headers["accept"] ?? "application/json";

    if (accept.includes("text/html")) {
        return res.html("product-detail.html", { product });
    }

    if (accept.includes("application/xml")) {
        const xml = `<?xml version="1.0"?><product><id>${id}</id><name>${product.name}</name><pr
        return res.header("Content-Type", "application/xml").text(xml);
    }

    if (accept.includes("text/plain")) {
        return res.text(`Product #${id}: ${product.name} - $${product.price}`);
    }
```

```
    return res.json(product);
});
```

---

11. Input Validation

Tina4 includes a `Validator` class for declarative input validation. Chain rules together and check the result. If validation fails, use `res.error()` to return a structured error envelope.

The Validator Class

```
import { Validator } from "tina4-nodejs";

Router.post("/api/users", (req, res) => {
    const v = new Validator(req.body);
    v.required("name").required("email").email("email").minLength("name", 2);

    if (!v.isValid()) {
        return res.error("VALIDATION_FAILED", v.errors()[0].message, 400);
    }

    // proceed with valid data
});
```

The `Validator` accepts the request body (an object) and provides chainable methods:

Method	Description
`required(field)`	Field must be present and non-empty
`email(field)`	Field must be a valid email address
`minLength(field, n)`	Field must have at least `n` characters
`maxLength(field, n)`	Field must have at most `n` characters
`numeric(field)`	Field must be a number
`inList(field, values)`	Field must be one of the allowed values

Call `v.isValid()` to check all rules. Call `v.errors()` to get the array of failures, each with a `field` and `message` property.

The Error Response Envelope

`res.error()` returns a consistent JSON error envelope:

```
return res.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 via the `TINA4_MAX_UPLOAD_SIZE` environment variable. The value is in bytes. The default is `10485760` (10 MB).

```
TINA4_MAX_UPLOAD_SIZE=10485760
```

If a client sends a file larger than this limit, Tina4 returns a `413 Payload Too Large` response before your handler runs. To allow larger uploads, increase the value in `.env`:

```
TINA4_MAX_UPLOAD_SIZE=52428800
```

This sets the limit to 50 MB.

---

12. 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 files only
- Maximum file size: 2MB
- Save files to `src/public/uploads/` with a unique filename
- Return the original filename, the saved filename, file size in KB, and the URL
- The GET endpoint should serve the file directly (not JSON)

---

13. Solution

Create `src/routes/images.ts`:

```
import { Router } from "tina4-nodejs";
import { rename, mkdir } from "fs/promises";
import { existsSync } from "fs";
import { join, extname } from "path";
import { randomUUID } from "crypto";

Router.post("/api/images", async (req, res) => {
    if (!req.files?.image) {
```

```
        return res.status(400).json({ error: "No image file provided. Use field name 'image'." }
    }

    const file = req.files.image;

    const allowedTypes = ["image/jpeg", "image/png", "image/webp"];
    if (!allowedTypes.includes(file.type)) {
        return res.status(400).json({
            error: "Invalid file type",
            received: file.type,
            allowed: allowedTypes
        });
    }

    const maxSize = 2 * 1024 * 1024;
    if (file.size > maxSize) {
        return res.status(400).json({
            error: "File too large",
            size_bytes: file.size,
            max_bytes: maxSize
        });
    }

    const ext = extname(file.name).toLowerCase();
    const savedName = `img_${randomUUID()}${ext}`;
    const uploadDir = join(process.cwd(), "src/public/uploads");
    const destination = join(uploadDir, savedName);

    if (!existsSync(uploadDir)) {
        await mkdir(uploadDir, { recursive: true });
    }

    await rename(file.tmpPath, destination);

    return res.status(201).json({
        message: "Image uploaded successfully",
        original_name: file.name,
        saved_name: savedName,
        size_kb: Math.round(file.size / 1024 * 10) / 10,
        type: file.type,
        url: `/uploads/${savedName}`
    });
});

Router.get("/api/images/{filename}", async (req, res) => {
    const filename = req.params.filename;

    if (filename.includes("..") || filename.includes("/")) {
        return res.status(400).json({ error: "Invalid filename" });
    }

    const filepath = join(process.cwd(), "src/public/uploads", filename);

    if (!existsSync(filepath)) {
        return res.status(404).json({ error: "Image not found", filename });
    }

    return res.file(filepath);
});
```

Expected output for upload:

```
{
  "message": "Image uploaded successfully",
  "original_name": "photo.jpg",
  "saved_name": "img_a1b2c3d4-e5f6-7890-abcd-ef1234567890.jpg",
  "size_kb": 240.0,
  "type": "image/jpeg",
  "url": "/uploads/img_a1b2c3d4-e5f6-7890-abcd-ef1234567890.jpg"
}
```

(Status: `201 Created`)

---

14. Gotchas

1. Forgetting `return`

Problem: Your handler runs (log output appears) but the browser shows an empty response or a 500 error.

Cause: You wrote `res.json({...})` without `return`.

Fix: Write `return res.json({...})`. The response object must be returned from the handler for Tina4 to send it.

2. Body Is Undefined for JSON Requests

Problem: `req.body` is `undefined` or empty even though you are sending 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"` when sending JSON with curl. In frontend JavaScript, `fetch()` with `JSON.stringify()` requires `headers: {"Content-Type": "application/json"}`.

3. File Uploads Return Empty

Problem: `req.files` is empty even though you are uploading a file.

Cause: The form is not using `enctype="multipart/form-data"`, or the curl command uses `-d` instead of `-F`.

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

4. Redirect Loops

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

Cause: Route A redirects to route B, which redirects back to route A.

Fix: Trace the redirect chain in the browser's network inspector. Break the cycle.

5. Cookie Not Set

Problem: You called `res.cookie(...)` but the browser does not show the cookie.

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

Fix: Set `secure: false` during development.

6. Large Request Body Rejected

Problem: POST requests with large bodies return a 413 error.

Cause: The request body exceeds the configured maximum size.

Fix: Increase `TINA4_MAX_UPLOAD_SIZE` in `.env`. The default is `10mb`.

7. Headers Are Lowercase in Node.js

Problem: `req.headers["Content-Type"]` is `undefined` even though the header was sent.

Cause: Node.js normalizes all header names to lowercase.

Fix: Use lowercase header names: `req.headers["content-type"]`.

Templates

1. Why Templates

In Chapter 1, you saw `res.html("products.html", data)` produce a full HTML page. That rendering was done by **Frond**, Tina4's built-in template engine. Zero dependencies. Twig-compatible. Built from scratch. If you know Twig, Jinja2, or Nunjucks, you know 90% of Frond.

Templates live in `src/templates/`. Call `res.html("page.html", data)` and Frond loads `src/templates/page.html`, processes the tags and expressions, and returns the final HTML.

This chapter covers every feature of the template engine. After this, you build real pages.

---

2. Variables and Expressions

Output a variable with double curly braces:

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

Route handler:

```
import { Router } from "tina4-nodejs";

Router.get("/welcome", async (req, res) => {
    return res.html("welcome.html", {
        name: "Alice"
    });
});
```

Create `src/templates/welcome.html`:

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

Expected browser output:

```
Hello, Alice!
```

Accessing Nested Data

Dot notation reaches into nested objects:

```
const data = {
    user: {
        name: "Alice",
        email: "alice@example.com",
        address: {
            city: "Cape Town",
            country: "South Africa"
```

```
        }
    }
};

return res.html("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 Objects

If an object value is a function, Frond calls it automatically when accessed via dot notation. You can also call methods with arguments:

```
const data = {
    user: {
        name: "Alice",
        t: (key: string) => translations[key] ?? key
    }
};

return res.html("page.html", data);
```

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

This works with any callable property. Arguments are evaluated as expressions, so you can pass variables, strings, or numbers.

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 engine's 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>
```

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:

```
import { Router } from "tina4-nodejs";

Router.get("/about", async (req, res) => {
    return res.html("about.twig", {
        founded_year: 2020,
        team_size: 12,
        office_count: 3
    });
});
```

Using `{{ parent() }}`

Add to a block rather than replace 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 %}
```

---

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.inStock %}
        <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 %}
```

Passing Variables to Includes

```
{% 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 %}
```

---

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 %}
```

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

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

Testing for Existence

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

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

```
{{ 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 inspection of the value. Unlike `JSON.stringify`, the built-in inspector handles everything safely:

- **Circular references** — marked `[Circular]` (no crash)
- **BigInt** — shown as `123n` (no crash)
- **Map / Set** — `Map(2) { "a" => 1, "b" => 2 }` / `Set(3) { 1, 2, 3 }`
- **Date** — `Date(2026-04-09T13:00:00.000Z)` (type retained)
- **Error** — `Error("message")` (type + message)
- **Class instances** — `User { name: "Alice" }` (class name preserved)
- **Functions / Symbols / undefined** — shown inline, not silently dropped

```
{{ dump(order) }}

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

dump is gated on `TINA4_DEBUG=true`. In production (env var unset or `false`) **both** the filter and function form silently return an empty `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. Define once. 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

```
{% 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) }}

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

---

9. Special Tags

{% raw %} -- Literal Output

When you need to output literal `{{ }}` (for a Vue.js template, for example):

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

{% spaceless %} -- Remove Whitespace Between Tags

The `spaceless` tag strips whitespace between HTML tags. Useful for inline elements where whitespace affects layout:

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

Output:

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

Only whitespace between tags is removed. Whitespace inside text content stays intact.

{% autoescape %} -- Control HTML Escaping

By default, Frond escapes HTML in `{{ }}` output to prevent XSS attacks. The `autoescape` tag controls this behavior for a block:

```
{% autoescape false %}
    {{ raw_html }}
{% endautoescape %}
```

With `autoescape false`, the variable outputs as raw HTML without escaping. Use this when you trust the content -- rendering Markdown-to-HTML output, for example.

To re-enable escaping inside a `false` block:

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

Whitespace Control

Use hyphens in tag delimiters to trim whitespace:

```
{%- if condition -%}
    No leading or trailing whitespace around this block
{%- endif -%}
```

The `-` on the left trims whitespace before the tag. The `-` on the right trims whitespace after the tag. Works with `{{ }}`, `{% %}`, and `{# #}` delimiters.

Comments

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

---

10. Template Route Export Pattern

Tina4 Node.js has a special pattern for file-based routes with templates. Export a `template` constant and return data from the handler:

Create `src/routes/catalog/get.ts`:

```
export const template = "catalog.twig";

export default async (req, res) => {
    const category = req.query.category ?? "";

    const 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 }
    ];

    const filtered = category
        ? products.filter(p => p.category.toLowerCase() === category.toLowerCase())
        : products;

    return {
        products: filtered,
        active_category: category,
        categories: [...new Set(products.map(p => p.category))]
    };
};
```

Tina4 renders `src/templates/catalog.twig` with the returned data. The route handler stays clean -- it returns data, the framework handles rendering.

---

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 and a `categoryFilter(categories, active)` macro
- Create a page template at `src/templates/catalog.twig` that extends the base, uses the macros, shows category filter buttons, and shows product cards in a grid
- Create a route at `GET /catalog` that accepts an optional `?category=` filter

Data

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

---

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; }
        .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.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; }
        .product-price { font-size: 1.2em; font-weight: bold; color: #27ae60; }
        .badge-featured { background: #f39c12; color: white; display: inline-block; padding: 2px
        .badge-stock { background: #d4edda; color: #155724; display: inline-block; padding: 2px
        .badge-nostock { background: #f8d7da; color: #721c24; display: inline-block; padding: 2p
        .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-featured">Featured</span>
            {% endif %}
        </p>
        <p class="product-category">{{ product.category }}</p>
        <p class="product-price">
            ${{ product.price | number_format(2) }}
            {% if product.inStock %}
                <span class="badge-stock">In Stock</span>
            {% else %}
                <span class="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.ts`:

```
import { Router } from "tina4-nodejs";

Router.get("/catalog", async (req, res) => {
    const allProducts = [
        { name: "Espresso Machine", category: "Kitchen", price: 299.99, inStock: true, featured:
        { name: "Yoga Mat", category: "Fitness", price: 29.99, inStock: true, featured: false },
        { name: "Standing Desk", category: "Office", price: 549.99, inStock: true, featured: tru
        { name: "Blender", category: "Kitchen", price: 89.99, inStock: false, featured: false },
        { name: "Running Shoes", category: "Fitness", price: 119.99, inStock: true, featured: fa
        { name: "Desk Lamp", category: "Office", price: 39.99, inStock: true, featured: true },
        { name: "Cast Iron Skillet", category: "Kitchen", price: 44.99, inStock: true, featured:
    ];

    const categories = [...new Set(allProducts.map(p => p.category))].sort();
    const activeCategory = req.query.category ?? "";

    const products = activeCategory
        ? allProducts.filter(p => p.category.toLowerCase() === String(activeCategory).toLowerCas
        : allProducts;

    return res.html("catalog.twig", {
        products,
        categories,
        active_category: activeCategory
    });
});
```

Expected 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) have a gold border and "Featured" badge

Expected browser output for `/catalog?category=Kitchen`:

- Header shows "3 products in Kitchen"
- The 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. No exceptions.

Fix: Make `{% extends %}` the absolute first thing in the file.

2. Undefined Variables Show Nothing

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

Cause: Frond outputs nothing for undefined variables. By design.

Fix: Use the `default` filter: `{{ username | default("Guest") }}`.

3. Auto-Escaping Prevents HTML Output

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

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

Fix: For trusted content, use `{{ content | raw }}`. Never use `raw` on user-supplied input.

4. Variable Scope in Includes

Problem: A variable defined inside a `{% for %}` loop is not accessible after the loop ends.

Cause: Loop variables are scoped to the loop.

Fix: Use `{% set %}` before the loop to accumulate values.

5. Macro Arguments Are Positional

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

Cause: Frond macros use positional arguments, not keyword arguments.

Fix: Pass arguments in the order defined: `{{ button("Click", "/url", "danger") }}`.

6. Template File Extension Does Not Matter

Problem: Not sure whether to use `.html`, `.twig`, or `.tpl`.

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

Fix: Pick one extension. Be consistent. This book uses `.twig` for templates with Twig syntax and `.html` for simple HTML files.

7. Filters Are Not JavaScript Functions

Problem: You try `{{ items | count }}` or `{{ name | toUpperCase }}` and get an error.

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

Fix: Use `{{ items | length }}` instead of `count`. Use `{{ name | upper }}` instead of `toUpperCase`.

Database

1. From Arrays to Real Data

In Chapters 2 and 3, all your data lived in TypeScript arrays. Server restart. Data gone. That works for learning routing and responses. Real applications need persistent storage.

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 of them. One line in `.env` switches the engine.

---

2. Connecting to a Database

The Default: SQLite

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

```
TINA4_DEBUG=true
```

With no explicit `TINA4_DATABASE_URL`, Tina4 defaults to `sqlite:///data/app.db`. That is why the health check at `/health` shows `"database": "connected"` with zero configuration.

SQLite support uses Node's built-in `node:sqlite` module (Node 22+). No native C++ addons are needed. No `node-gyp`. No platform-specific binaries. This is what makes Tina4 Node.js truly zero runtime dependencies -- even the database driver ships with Node itself.

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
```

The Firebird adapter requires the `node-firebird` package (`npm install node-firebird`). SQL dialect differences are handled automatically: `LIMIT`/`OFFSET` is translated to `ROWS X TO Y`, boolean values are converted to integers, and `ILIKE` is converted to `LOWER() LIKE LOWER()`.

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 `new URL()`, 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

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

Connection Pooling

For applications that handle many concurrent requests, enable connection pooling by passing a pool size to `Database.create()`:

```
const db = await Database.create("postgres://localhost/mydb", undefined, undefined, 5);
```

The fourth argument controls how many database connections are maintained:

- `0` (the default) -- a single connection is used for all queries
- `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

```
curl http://localhost:7148/health
```

```
{
  "status": "ok",
  "database": "connected",
  "uptime_seconds": 3,
```

```
  "version": "3.10.3",
  "framework": "tina4-nodejs"
}
```

---

3. Getting the Database Object

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

```
import { Router } from "tina4-nodejs";
import { Database } from "tina4-nodejs/orm";

Router.get("/api/test-db", async (req, res) => {
    const db = Database.getConnection();

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

    return res.json(result);
});
```

```
curl http://localhost:7148/api/test-db
```

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

`Database.getConnection()` returns the active database connection. Call `fetch()`, `execute()`, and `fetchOne()` on this object. All database methods are async. All return Promises.

---

4. Raw Queries

fetch() -- Get Multiple Rows

```
const db = Database.getConnection();

const products = await db.fetch("SELECT * FROM products WHERE price > 50");
```

Each row is a plain object 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

```
const result = await db.fetch("SELECT * FROM users WHERE active = ?", [1]);

result.records;      // [{ id: 1, name: "Alice" }, { id: 2, name: "Bob" }]
result.columns;      // ["id", "name", "email", "active"]
result.count;        // total number of matching rows
result.limit;        // query limit (if set)
result.offset;       // query offset (if set)
```

Iteration

A `DatabaseResult` is iterable. Use it directly in `for...of`:

```
for (const user of result) {
    console.log(user.name);
}
```

Index Access

Access rows by index like a regular array:

```
const firstUser = result[0];
```

Countable

The `length` property works on the result:

```
console.log(result.length); // number of records in this result set
```

Conversion Methods

```
result.toJson();      // JSON string of all records
result.toCsv();       // CSV string with column headers
result.toArray();     // plain array of objects
result.toPaginate();  // { records: [...], count: 42, limit: 10, offset: 0 }
```

`toPaginate()` is designed for building paginated API responses. It bundles the records with the total count, limit, and offset in a single object.

Schema Metadata with columnInfo()

`columnInfo()` 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:

```
const info = await result.columnInfo();
// [
//     { name: "id", type: "INTEGER", size: null, decimals: null, nullable: false, primaryKey: t
//     { name: "name", type: "TEXT", size: null, decimals: null, nullable: false, primaryKey: fa
//     { name: "email", type: "TEXT", size: 255, decimals: null, nullable: true, primaryKey: fal
//     ...
// ]
```

Each column entry contains:

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

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

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

fetchOne() -- Get a Single Row

```
const product = await db.fetchOne("SELECT * FROM products WHERE id = 1");
// Returns: { id: 1, name: "Keyboard", price: 79.99 }
```

If no row matches, `fetchOne()` returns `null`.

execute() -- Run a Statement

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

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

Full Example: A Simple Query Route

```
import { Router } from "tina4-nodejs";
import { Database } from "tina4-nodejs/orm";

Router.get("/api/products", async (req, res) => {
    const db = Database.getConnection();

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

    return res.json({
        products,
        count: products.length
    });
});
```

---

5. Parameterised Queries

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

```
// NEVER do this:
await db.fetch(`SELECT * FROM products WHERE name = '${userInput}'`);
```

Instead, use parameterised queries:

```
const db = Database.getConnection();

// Named parameters
const product = await db.fetchOne(
    "SELECT * FROM products WHERE id = :id",
```

```
    { id: 42 }
);

// Positional parameters
const products = await db.fetch(
    "SELECT * FROM products WHERE price BETWEEN ? AND ? ORDER BY price",
    [10.00, 100.00]
);
```

A Safe Search Endpoint

```
import { Router } from "tina4-nodejs";
import { Database } from "tina4-nodejs/orm";

Router.get("/api/products/search", async (req, res) => {
    const db = Database.getConnection();

    const q = req.query.q ?? "";
    const maxPrice = parseFloat(req.query.max_price ?? "99999");

    if (!q) {
        return res.status(400).json({ error: "Query parameter 'q' is required" });
    }

    const products = await db.fetch(
        "SELECT * FROM products WHERE name LIKE :query AND price <= :maxPrice ORDER BY name",
        { query: `%${q}%`, maxPrice }
    );

    return res.json({
        query: q,
        max_price: maxPrice,
        results: products,
        count: products.length
    });
});
```

```
curl "http://localhost:7148/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

When you need multiple operations to succeed or fail together, use transactions:

```
import { Router } from "tina4-nodejs";
import { Database } from "tina4-nodejs/orm";

Router.post("/api/orders", async (req, res) => {
    const db = Database.getConnection();
    const body = req.body;
```

```
    try {
        await db.startTransaction();

        await db.execute(
            "INSERT INTO orders (customer_id, total, status) VALUES (:customerId, :total, 'pendi
            { customerId: body.customer_id, total: body.total }
        );

        const order = await db.fetchOne("SELECT last_insert_rowid() AS id");
        const orderId = order.id;

        for (const item of body.items) {
            await db.execute(
                "INSERT INTO order_items (order_id, product_id, quantity, price) VALUES (:orderI
                {
                    orderId,
                    productId: item.product_id,
                    qty: item.quantity,
                    price: item.price
                }
            );

            await db.execute(
                "UPDATE products SET stock = stock - :qty WHERE id = :productId",
                { qty: item.quantity, productId: item.product_id }
            );
        }

        await db.commit();

        return res.status(201).json({ order_id: orderId, status: "created" });
    } catch (e) {
        await db.rollback();
        return res.status(500).json({ error: `Order failed: ${e.message}` });
    }
});
```

---

7. Schema Inspection

getTables()

```
const db = Database.getConnection();
const tables = await db.getTables();
// Returns: ["orders", "order_items", "products", "users"]
```

getColumns()

```
const columns = await db.getColumns("products");
// Returns: [
//     { name: "id", type: "INTEGER", nullable: false, primary: true },
//     { name: "name", type: "TEXT", nullable: false, primary: false },
//     ...
// ]
```

tableExists()

```
if (await db.tableExists("products")) {
    // Table exists, safe to query
}
```

getDatabaseType()

Identify the connected database engine at runtime:

```
const dbType = db.getDatabaseType();
// Returns: "sqlite", "postgresql", "mysql", or "mssql"
```

Useful when you need database-specific SQL syntax or want to display the engine in a status page.

Schema Info Endpoint

Combine schema inspection methods to build an introspection API:

```
import { Router } from "tina4-nodejs";
import { Database } from "tina4-nodejs/orm";

Router.get("/api/schema", async (req, res) => {
    const db = Database.getConnection();

    const tables = await db.getTables();
    const schema: Record<string, any> = {};

    for (const table of tables) {
        schema[table] = await db.getColumns(table);
    }

    return res.json({
        database_type: db.getDatabaseType(),
        tables: schema,
        table_count: tables.length,
    });
});
```

This endpoint returns the full database schema -- every table and every column with its type and constraints. Useful for debugging, admin dashboards, and auto-generating documentation.

---

8. FakeData Seeder

Tina4 includes a `FakeData` generator for populating tables with realistic test data:

```
import { FakeData } from "tina4-nodejs";

const fake = new FakeData();

fake.name();       // "Alice Johnson"
fake.email();      // "alice.johnson@example.com"
fake.phone();      // "+1-555-0142"
fake.address();    // "742 Evergreen Terrace, Springfield"
fake.company();    // "Acme Corp"
fake.sentence();   // "The quick brown fox jumps over the lazy dog."
```

```
fake.paragraph();  // Multiple sentences of lorem text
fake.number(1, 100);   // Random integer between 1 and 100
fake.decimal(0, 1000); // Random decimal
fake.boolean();    // true or false
fake.date("2024-01-01", "2026-12-31"); // Random date in range
fake.uuid();       // "a1b2c3d4-e5f6-..."
```

Seeding a Table

```
import { Database } from "tina4-nodejs/orm";
import { FakeData } from "tina4-nodejs";

const db = Database.getConnection();
const fake = new FakeData();

for (let i = 0; i < 50; i++) {
    db.execute(
        "INSERT INTO users (name, email, company) VALUES (:name, :email, :company)",
        {
            name: fake.name(),
            email: fake.email(),
            company: fake.company(),
        }
    );
}

console.log("Seeded 50 users");
```

Run it as a script:

```
npx tsx scripts/seed-users.ts
```

FakeData generates consistent-looking data without external packages. Use it for development, demos, and test setup.

---

9. Batch Operations with executeMany()

Insert or update many rows efficiently:

```
const db = Database.getConnection();

const products = [
    { name: "Widget A", price: 9.99 },
    { name: "Widget B", price: 14.99 },
    { name: "Widget C", price: 19.99 },
    { name: "Widget D", price: 24.99 }
];

await db.executeMany(
    "INSERT INTO products (name, price) VALUES (:name, :price)",
    products
);
```

---

10. Helper Methods: insert(), update(), delete()

insert()

```
const db = Database.getConnection();

await db.insert("products", {
    name: "Wireless Mouse",
    price: 34.99,
    in_stock: 1
});

// Insert multiple rows
await db.insert("products", [
    { name: "USB Cable", price: 9.99, in_stock: 1 },
    { name: "HDMI Cable", price: 14.99, in_stock: 1 }
]);
```

update()

```
await db.update("products", { price: 39.99, in_stock: 1 }, "id = :id", { id: 7 });
```

delete()

```
await db.delete("products", "id = :id", { id: 7 });
```

---

11. Migrations

Migrations are versioned SQL scripts. They evolve your database schema over time. Each migration runs once. Never again.

File Naming

Tina4 supports two naming patterns for migration files:

- **Sequential:** `000001_create_products.sql`
- **Timestamp:** `YYYYMMDDHHMMSS_create_products.sql`

Both patterns sort correctly. Tina4 uses BigInt comparison internally, so you can mix them in the same project without issues.

Generating a Migration

```
tina4 generate migration create_products_table
```

This creates two files in the `migrations/` folder:

```
migrations/20260324120000_create_products_table.sql
migrations/20260324120000_create_products_table.down.sql
```

The first file is the forward (up) migration. The second is the down migration used for rollbacks. Edit each one separately.

Forward migration (`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
);
```

Down migration (`20260324120000_create_products_table.down.sql`):

```
DROP TABLE IF EXISTS products;
```

Down migrations are optional. If you skip them, rollback will warn you but still remove the tracking record.

Running Migrations

```
tina4 migrate
# or
tina4nodejs migrate
```

Each run increments a batch number. Every migration applied during that run belongs to the same batch. This matters for rollback.

Checking Status

```
tina4nodejs migrate:status
```

This shows which migrations have been applied and which are still pending.

Rolling Back

```
tina4nodejs migrate:rollback
```

Rollback undoes the entire last batch. It finds each migration in the batch, runs its `.down.sql` file, and removes the tracking record. If a `.down.sql` file is missing, rollback warns but still cleans up the tracking entry.

The Tracking Table

Tina4 creates a `tina4_migration` table automatically. It has these columns:

Column	Purpose
`id`	Auto-incrementing primary key
`description`	The migration filename
`content`	Full SQL text of the migration (for audit)
`passed`	Whether the migration ran successfully
`batch`	Which batch this migration belongs to
`run_at`	When the migration was applied

Advanced SQL Splitting

Migration files can contain multiple statements. Tina4 splits them on semicolons, but it is smart about edge cases:

- `$$` **delimited blocks** for PostgreSQL stored procedures and functions
- `//` **blocks** for procedure definitions
- `/* */` **block comments** are preserved
- `--` **line comments** are preserved

This means you can write PostgreSQL stored procedures in your migration files without Tina4 breaking on internal semicolons.

---

12. Query Caching

Enable query caching in `.env`:

```
TINA4_DB_CACHE=true
```

Identical queries with identical parameters return cached results. The cache invalidates itself when you call `execute()`, `insert()`, `update()`, or `delete()` on the same table.

```
// Force a fresh query (bypass cache)
const products = await db.fetch("SELECT * FROM products", [], { noCache: true });

// Clear the entire cache
await db.clearCache();
```

---

13. 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 for a `notes` table with: `id`, `title`, `content`, `tag`, `created_at`, `updated_at`
- 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.

---

14. Solution

Migration

Generate the migration files:

```
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;
```

```
tina4 migrate
```

Routes

Create `src/routes/notes.ts`:

```
import { Router } from "tina4-nodejs";
import { Database } from "tina4-nodejs/orm";

Router.get("/api/notes", async (req, res) => {
    const db = Database.getConnection();
    const tag = req.query.tag ?? "";
    const search = req.query.search ?? "";

    let sql = "SELECT * FROM notes";
    const params: Record<string, any> = {};
    const conditions: string[] = [];

    if (tag) {
        conditions.push("tag = :tag");
        params.tag = tag;
    }

    if (search) {
        conditions.push("(title LIKE :search OR content LIKE :search)");
        params.search = `%${search}%`;
    }
```

```
    if (conditions.length > 0) {
        sql += " WHERE " + conditions.join(" AND ");
    }

    sql += " ORDER BY updated_at DESC";

    const notes = await db.fetch(sql, params);

    return res.json({ notes, count: notes.length });
});

Router.get("/api/notes/{id:int}", async (req, res) => {
    const db = Database.getConnection();
    const id = req.params.id;

    const note = await db.fetchOne("SELECT * FROM notes WHERE id = :id", { id });

    if (note === null) {
        return res.status(404).json({ error: "Note not found", id });
    }

    return res.json(note);
});

Router.post("/api/notes", async (req, res) => {
    const db = Database.getConnection();
    const body = req.body;

    const errors: string[] = [];
    if (!body.title) errors.push("Title is required");
    if (!body.content) errors.push("Content is required");

    if (errors.length > 0) {
        return res.status(400).json({ errors });
    }

    await db.execute(
        "INSERT INTO notes (title, content, tag) VALUES (:title, :content, :tag)",
        {
            title: body.title,
            content: body.content,
            tag: body.tag ?? "general"
        }
    );

    const note = await db.fetchOne("SELECT * FROM notes WHERE id = last_insert_rowid()");

    return res.status(201).json(note);
});

Router.put("/api/notes/{id:int}", async (req, res) => {
    const db = Database.getConnection();
    const id = req.params.id;
    const body = req.body;

    const existing = await db.fetchOne("SELECT * FROM notes WHERE id = :id", { id });

    if (existing === null) {
        return res.status(404).json({ error: "Note not found", id });
```

```
    }

    await db.execute(
        "UPDATE notes SET title = :title, content = :content, tag = :tag, updated_at = CURRENT_T
        {
            title: body.title ?? existing.title,
            content: body.content ?? existing.content,
            tag: body.tag ?? existing.tag,
            id
        }
    );

    const note = await db.fetchOne("SELECT * FROM notes WHERE id = :id", { id });

    return res.json(note);
});

Router.delete("/api/notes/{id:int}", async (req, res) => {
    const db = Database.getConnection();
    const id = req.params.id;

    const existing = await db.fetchOne("SELECT * FROM notes WHERE id = :id", { id });

    if (existing === null) {
        return res.status(404).json({ error: "Note not found", id });
    }

    await db.execute("DELETE FROM notes WHERE id = :id", { id });

    return res.status(204).json(null);
});
```

Expected output for create (Status: `201 Created`):

```
{
  "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"
}
```

---

15. Gotchas

1. Forgetting await

Problem: Database operations return `Promise {<pending>}` instead of results.

Cause: You forgot to `await` the database call. All Tina4 database methods are async.

Fix: Always `await` database calls: `const result = await db.fetch(...)`.

2. Connection String Formats

Problem: The database will not connect.

Cause: Each database engine expects a specific URL format. A common mistake is omitting the port.

Fix: Always include the port. Default ports: PostgreSQL 5432, MySQL 3306, MSSQL 1433, Firebird 3050.

3. SQLite File Paths

Problem: SQLite creates a new empty database instead of using the existing one.

Cause: Use three slashes for a relative path: `sqlite:///data/app.db`. Four slashes for absolute: `sqlite:////var/data/app.db`.

4. Parameterised Queries with LIKE

Problem: `WHERE name LIKE '%:q%'` does not work.

Cause: Parameters inside quotes are literal text, not placeholders.

Fix: Include the `%` in the parameter value: `{ q: "%" + search + "%" }`. The SQL should be `WHERE name LIKE :q`.

5. Boolean Values in SQLite

Problem: SQLite stores booleans as integers (1 and 0).

Fix: Cast in your TypeScript code: `inStock: Boolean(row.in_stock)`.

6. Migration Already Applied

Problem: You edited a migration file and ran `tina4 migrate` again, but nothing changed.

Cause: Once applied, a migration will not run again.

Fix: Create a new migration for schema changes. Do not edit applied migrations. If you need to undo, run `tina4nodejs migrate:rollback` first, then fix the migration and re-run.

8. Down Migration Missing on Rollback

Problem: You ran `tina4nodejs migrate:rollback` but the table was not dropped.

Cause: The `.down.sql` file is missing. Rollback removes the tracking record but cannot undo the schema change without it.

Fix: Always generate migrations with `tina4 generate migration`, which creates both files. If you created the migration manually, add the `.down.sql` file before you need to roll back.

7. fetch() Returns Empty Array, Not Null

Problem: You check `if (result === null)` but it never matches when the table is empty.

Cause: `fetch()` always returns an array. Only `fetchOne()` returns `null`.

Fix: Check with `if (result.length === 0)`.

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 object keys. Over and over.

Tina4's ORM turns database rows into TypeScript 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/models/`. Every `.ts` file in that directory is auto-loaded at startup.

Create `src/models/Note.ts`:

```
import { BaseModel } from "tina4-nodejs/orm";

export default class Note extends BaseModel {
  static tableName = "notes";
  static fields = {
    id:        { type: "integer" as const, primaryKey: true, autoIncrement: true },
    title:     { type: "string" as const, required: true, maxLength: 200 },
    content:   { type: "string" as const, default: "" },
    category:  { type: "string" as const, default: "general" },
    pinned:    { type: "boolean" as const, default: false },
    createdAt: { type: "datetime" as const },
    updatedAt: { type: "datetime" as const },
  };
}
```

A complete model. Here is what each piece does:

- `static tableName` -- the database table this model maps to. If omitted, the ORM uses the lowercase class name (e.g. `Contact` -> `contact`).
- `primaryKey: true` on a field marks it as the primary key (defaults to `id` if none is specified)
- Each field is a property in the `static fields` object with a config object describing its type and constraints

Field Types

Field Type	TypeScript Type	SQL Type	Description
`"integer"`	`number`	`INTEGER`	Whole numbers
`"string"`	`string`	`TEXT`	Text strings
`"text"`	`string`	`TEXT`	Long text
`"number"`	`number`	`REAL`	Decimal numbers
`"boolean"`	`boolean`	`INTEGER` (0/1)	True/False
`"datetime"`	`string`	`TEXT`	Date and time

For foreign keys, use `"integer"`. There is no separate foreign key type -- the relationship is defined through `hasMany`, `hasOne`, and `belongsTo` methods instead.

Field Options

Option	Type	Description
`primaryKey`	`boolean`	Marks this field as the primary key
`required`	`boolean`	Field must have a value (not undefined)
`default`	any	Default value when not provided
`maxLength`	`number`	Maximum string length
`minLength`	`number`	Minimum string length
`min`	`number`	Minimum numeric value
`max`	`number`	Maximum numeric value
`choices`	array	Allowed values
`autoIncrement`	`boolean`	Auto-incrementing integer
`pattern`	`string`	Regex pattern the value must match

Field Mapping

When your TypeScript property names do not match the database column names, use `static fieldMapping` to define the translation:

```
import { BaseModel } from "tina4-nodejs/orm";

export default class User extends BaseModel {
  static tableName = "user_accounts";
```

```
  static fieldMapping = {
    firstName: "fname",        // TS property -> DB column
    lastName:  "lname",
    emailAddress: "email",
  };

  static fields = {
    id:           { type: "integer" as const, primaryKey: true, autoIncrement: true },
    firstName:    { type: "string" as const, required: true },
    lastName:     { type: "string" as const, required: true },
    emailAddress: { type: "string" as const, required: true },
  };
}
```

With this mapping, `user.firstName` reads from and writes to the `fname` column. The ORM handles the conversion in both directions -- on `findById()`, `save()`, `select()`, and `toDict()`. This is useful with legacy databases or third-party schemas where you cannot rename the columns.

autoMap and Case Conversion

The `static autoMap = true` flag auto-generates `fieldMapping` entries from camelCase field names to snake_case database column names. Explicit `fieldMapping` entries always take precedence.

```
import { BaseModel } from "tina4-nodejs/orm";

export default class User extends BaseModel {
  static tableName = "users";
  static autoMap = true;  // firstName -> first_name, lastName -> last_name

  static fields = {
    id:        { type: "integer" as const, primaryKey: true, autoIncrement: true },
    firstName: { type: "string" as const, required: true },
    lastName:  { type: "string" as const, required: true },
    email:     { type: "string" as const, required: true },
  };
}
```

Two utility functions handle case conversion directly:

```
import { snakeToCamel, camelToSnake } from "tina4-nodejs/orm";

snakeToCamel("first_name");   // "firstName"
camelToSnake("firstName");    // "first_name"
```

A common use case is Firebird or Oracle, which store column names in uppercase:

```
import { BaseModel } from "tina4-nodejs/orm";

export default class Account extends BaseModel {
  static tableName    = "ACCOUNTS";
  static fieldMapping: Record<string, string> = {
    accountNo:   "ACCOUNTNO",
    storeName:   "STORENAME",
    creditLimit: "CREDITLIMIT",
  };

  static fields = {
```

```
    accountNo:   { type: "string" as const },
    storeName:   { type: "string" as const },
    creditLimit: { type: "number" as const, default: 0 },
  };
}
```

TypeScript code uses clean camelCase names (`account.accountNo`, `account.creditLimit`). The ORM maps them to the uppercase DB columns automatically.

getDbColumn and getDbData

Two helpers expose the fieldMapping in custom code:

```
// Get the DB column name for a JS property
const col = Account.getDbColumn("accountNo");   // "ACCOUNTNO"

// Get all instance fields using DB column names as keys
const data = account.getDbData();
// { ACCOUNTNO: "A001", STORENAME: "Main Store", CREDITLIMIT: 5000 }
```

These are used internally by `save()` and `createTable()`, but available for custom queries.

find() vs where() -- naming convention

The two query methods have a deliberate difference in how they handle column names:

- `find(filterObj)` uses **TypeScript property names**. The ORM translates them via `fieldMapping`.
- `where(sql)` uses **raw DB column names** in the SQL string. No translation is applied.

```
// find() -- use TS property names (fieldMapping applied)
const accounts = Account.find({ accountNo: "A001" });   // translates to ACCOUNTNO = ?

// where() -- use DB column names directly in the SQL
const accounts2 = 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.

---

3. createTable -- Schema from Models

You can create the database table directly from your model definition:

```
Note.createTable();
```

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.

If the table already exists, `createTable()` does nothing.

---

4. CRUD Operations

save -- Create or Update

```
// src/routes/api/notes/post.ts
import type { Tina4Request, Tina4Response } from "tina4-nodejs";
import Note from "../../models/Note.js";

export default async function (req: Tina4Request, res: Tina4Response) {
  const body = req.body as Record<string, unknown>;

  const note = new Note();
  note.title = body.title;
  note.content = body.content ?? "";
  note.category = body.category ?? "general";
  note.pinned = body.pinned ?? false;
  note.save();

  res.json({ message: "Note created", note: note.toDict() }, 201);
}
```

`save()` detects whether the record is new (INSERT) or existing (UPDATE) based on whether the primary key has a value. It returns `this` on success for chaining. It returns `false` on failure.

create -- Build and Save in One Step

When you have a data object ready, `create()` builds the model and saves it in one call:

```
const note = Note.create({
  title: "Quick Note",
  content: "Created in one step",
  category: "general",
});
```

findById -- Fetch One Record by Primary Key

```
// src/routes/api/notes/[id]/get.ts
import type { Tina4Request, Tina4Response } from "tina4-nodejs";
import Note from "../../../models/Note.js";

export default async function (req: Tina4Request, res: Tina4Response) {
  const note = Note.findById(req.params.id);

  if (note === null) {
    return res.json({ error: "Note not found" }, 404);
  }

  res.json(note.toDict());
}
```

`findById()` takes a primary key value and returns a model instance, or `null` if no row matches. If soft delete is enabled, it excludes soft-deleted records.

Use `findOrFail()` when you want an Error thrown instead of `null`:

```
const note = Note.findOrFail(id);  // Throws Error if not found
```

find -- Query by Filter Dict

The `find()` method accepts an object of column-value pairs and returns an array of matching records:

```
// Find all notes in the "work" category
const workNotes = Note.find({ category: "work" });

// Find with pagination and ordering
const recent = Note.find({ pinned: true }, 10, 0, "created_at DESC");

// Find all records (no filter)
const allNotes = Note.find();
```

The full signature is `find(filter?, limit?, offset?, orderBy?, include?)`.

where -- Query with SQL Conditions

For more complex queries, `where()` takes a SQL WHERE clause with `?` placeholders:

```
const notes = Note.where("category = ?", ["work"]);
```

delete -- Remove a Record

```
// src/routes/api/notes/[id]/delete.ts
import type { Tina4Request, Tina4Response } from "tina4-nodejs";
import Note from "../../../models/Note.js";

export default async function (req: Tina4Request, res: Tina4Response) {
  const note = Note.findById(req.params.id);

  if (note === null) {
    return res.json({ error: "Note not found" }, 404);
  }

  note.delete();

  res.json(null, 204);
}
```

Listing Records

```
// src/routes/api/notes/get.ts
import type { Tina4Request, Tina4Response } from "tina4-nodejs";
import Note from "../../models/Note.js";

export default async function (req: Tina4Request, res: Tina4Response) {
  const category = req.query.category;

  let notes;
  if (category) {
    notes = Note.where("category = ?", [category]);
  } else {
    notes = Note.all();
  }

  res.json({
    notes: notes.map((n) => n.toDict()),
    count: notes.length,
  });
}
```

`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
const notes = Note.where("category = ?", ["work"], 20, 40);

// Fetch all with pagination -- all() takes an optional where clause string
const notes2 = Note.all("category = ?", ["work"]);

// SQL-first query -- full control over the SQL
const notes3 = Note.select(
  "SELECT * FROM notes WHERE pinned = ? ORDER BY created_at DESC",
  [1],
);
```

selectOne -- Fetch a Single Record by SQL

When you need exactly one record from a custom SQL query:

```
const note = Note.selectOne("SELECT * FROM notes WHERE slug = ?", ["my-note"]);
```

Returns a model instance or `null`.

load -- Populate an Existing Instance

The `load()` method fills an existing model instance from the database:

```
const note = new Note();
note.id = 42;
note.load();  // Loads data for id=42

// Or with a filter string
const note2 = new Note();
note2.load("slug = ?", ["my-note"]);
```

Returns `true` if a record was found, `false` otherwise.

count -- Count Records

```
const total = Note.count();
const workCount = Note.count("category = ?", ["work"]);
```

Respects soft delete -- only counts non-deleted records.

---

5. toDict, toJson, and Other Serialisation

toDict

Convert a model instance to a plain object:

```
const note = Note.findById(1);

const data = note.toDict();
// { id: 1, title: "Shopping List", content: "Milk, eggs", category: "personal", pinned: false,
```

The `include` parameter adds relationship data to the output (see Eager Loading below). Pass an array of relationship names:

```
// Include relationships in the output
const data = note.toDict(["comments"]);
```

toJson

Convert directly to a JSON string:

```
const jsonString = note.toJson();
// '{"id":1,"title":"Shopping List",...}'
```

Other Serialisation Methods

Method	Returns	Description
`toDict(include?)`	`Record<string, unknown>`	Primary dict method with optional relationship includes
`toAssoc(include?)`	`Record<string, unknown>`	Alias for `toDict()`
`toObject()`	`Record<string, unknown>`	Alias for `toDict()`
`toJson(include?)`	`string`	JSON string
`toArray()`	`unknown[]`	Flat array of values (no keys)
`toList()`	`unknown[]`	Alias for `toArray()`

---

6. Relationships

foreignKey Field Type — Auto-Wired Relationships

Declaring a field with `type: "foreignKey"` and a `references` model name automatically wires both sides of the relationship. The declaring model gets a `belongsTo` entry (the column name with `_id` stripped → the association name), and the referenced model gets a `hasMany` entry (the declaring model's table name, or whatever you pass via `relatedName`).

```
import { BaseModel } from "tina4-nodejs/orm";

export class User extends BaseModel {
    static tableName = "users";
    static fields = {
        id:   { type: "integer", primaryKey: true, autoIncrement: true },
        name: { type: "string", required: true },
    };
}

export class Post extends BaseModel {
    static tableName = "posts";
    static fields = {
        id:      { type: "integer", primaryKey: true, autoIncrement: true },
        title:   { type: "string", required: true },
        // Auto-wires Post.belongsTo User and User.hasMany Post
        user_id: { type: "foreignKey", references: "User" },
    };
}
```

With just the `foreignKey` field, both sides are accessible:

```
const post = Post.findById(1);
const user = post.belongsTo(User, "user_id");
console.log(user?.name);     // "Alice"

// Or via toDict with include
const postData = post.toDict(["user"]);

const alice = User.findById(1);
const posts = alice.hasMany(Post, "user_id");
posts.forEach(p => console.log(p.title));
```

For a custom `hasMany` key, pass `relatedName`:

```
user_id: { type: "foreignKey", references: "User", relatedName: "blog_posts" }
// User.hasMany entry will use "blog_posts" instead of the default
```

Models must be registered via `BaseModel.registerModel("User", User)` before eager loading can resolve the reference by name.

hasMany

An author has many posts:

Create `src/models/Author.ts`:

```
import { BaseModel } from "tina4-nodejs/orm";

export default class Author extends BaseModel {
```

```
  static tableName = "authors";
  static fields = {
    id:        { type: "integer" as const, primaryKey: true, autoIncrement: true },
    name:      { type: "string" as const, required: true },
    email:     { type: "string" as const, required: true },
    bio:       { type: "string" as const, default: "" },
    createdAt: { type: "datetime" as const },
  };
}
```

Create `src/models/BlogPost.ts`:

```
import { BaseModel } from "tina4-nodejs/orm";

export default class BlogPost extends BaseModel {
  static tableName = "posts";
  static fields = {
    id:        { type: "integer" as const, primaryKey: true, autoIncrement: true },
    authorId:  { type: "integer" as const, required: true },
    title:     { type: "string" as const, required: true, maxLength: 300 },
    slug:      { type: "string" as const, required: true },
    content:   { type: "string" as const, default: "" },
    status:    { type: "string" as const, default: "draft", choices: ["draft", "published", "arc
    createdAt: { type: "datetime" as const },
    updatedAt: { type: "datetime" as const },
  };
}
```

Now use `hasMany` to get an author's posts:

```
// src/routes/api/authors/[id]/get.ts
import type { Tina4Request, Tina4Response } from "tina4-nodejs";
import Author from "../../../models/Author.js";
import BlogPost from "../../../models/BlogPost.js";

export default async function (req: Tina4Request, res: Tina4Response) {
  const author = Author.findById(req.params.id);

  if (author === null) {
    return res.json({ error: "Author not found" }, 404);
  }

  const posts = author.hasMany(BlogPost, "author_id");

  const data = author.toDict();
  data.posts = posts.map((p) => p.toDict());

  res.json(data);
}
```

```
{
  "id": 1,
  "name": "Alice",
  "email": "alice@example.com",
  "bio": "Tech writer",
  "posts": [
    {"id": 1, "title": "Getting Started with Tina4", "slug": "getting-started", "status": "publi
    {"id": 2, "title": "Advanced Routing", "slug": "advanced-routing", "status": "draft"}
  ]
}
```

`hasMany()` accepts an optional `limit` and `offset` for pagination: `author.hasMany(BlogPost, "author_id", 10, 0)`.

hasOne

A user has one profile:

```
const profile = user.hasOne(Profile, "user_id");
```

Returns a single model instance or `null`.

belongsTo

A post belongs to an author:

```
// src/routes/api/posts/[id]/get.ts
import type { Tina4Request, Tina4Response } from "tina4-nodejs";
import Author from "../../../models/Author.js";
import BlogPost from "../../../models/BlogPost.js";

export default async function (req: Tina4Request, res: Tina4Response) {
  const post = BlogPost.findById(req.params.id);

  if (post === null) {
    return res.json({ error: "Post not found" }, 404);
  }

  const author = post.belongsTo(Author, "author_id");

  const data = post.toDict();
  data.author = author ? author.toDict() : null;

  res.json(data);
}
```

```
{
  "id": 1,
  "authorId": 1,
  "title": "Getting Started with Tina4",
  "slug": "getting-started",
  "content": "...",
  "status": "published",
  "author": {
    "id": 1,
    "name": "Alice",
    "email": "alice@example.com"
  }
}
```

---

7. Eager Loading

Calling relationship methods inside a loop creates the N+1 problem. Load 10 authors. Call `hasMany(BlogPost, "author_id")` for each one. That fires 11 queries -- 1 for authors, 10 for posts. The page drags.

The `include` parameter on `all()`, `where()`, `findById()`, and `selectOne()` solves this. It eager-loads relationships in bulk.

Eager loading requires two things: declarative relationship definitions on the model, and model registration.

Declarative Relationships

Define relationships as static arrays on your model class:

```
import { BaseModel } from "tina4-nodejs/orm";

export default class Author extends BaseModel {
  static tableName = "authors";
  static hasMany = [{ model: "BlogPost", foreignKey: "author_id" }];
  static fields = {
    id:    { type: "integer" as const, primaryKey: true, autoIncrement: true },
    name:  { type: "string" as const, required: true },
    email: { type: "string" as const, required: true },
    bio:   { type: "string" as const, default: "" },
  };
}
```

```
import { BaseModel } from "tina4-nodejs/orm";

export default class BlogPost extends BaseModel {
  static tableName = "posts";
  static belongsTo = [{ model: "Author", foreignKey: "author_id" }];
  static hasMany = [{ model: "Comment", foreignKey: "post_id" }];
  static fields = {
    id:       { type: "integer" as const, primaryKey: true, autoIncrement: true },
    authorId: { type: "integer" as const, required: true },
    title:    { type: "string" as const, required: true },
    status:   { type: "string" as const, default: "draft" },
  };
}
```

Register the models so eager loading can resolve them by name:

```
BaseModel.registerModel("Author", Author);
BaseModel.registerModel("BlogPost", BlogPost);
BaseModel.registerModel("Comment", Comment);
```

Now use `include` to eager-load:

```
// Eager load posts when fetching all authors
const authors = Author.all(undefined, undefined, ["posts"]);

// Eager load author and comments when finding a single post
const post = BlogPost.findById(1, ["author", "comments"]);
```

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
const authors = Author.all(undefined, undefined, ["posts", "posts.comments"]);
```

Authors, their posts, and each post's comments. Three queries total instead of hundreds.

toDict with Nested Includes

When eager loading is active, `toDict(include)` embeds the related data:

```
const post = BlogPost.findById(1, ["author", "comments"]);
const data = post.toDict(["author", "comments"]);
```

```
{
  "id": 1,
  "title": "Getting Started with Tina4",
  "author": {
    "id": 1,
    "name": "Alice",
    "email": "alice@example.com"
  },
  "comments": [
    {"id": 1, "body": "Great post!", "authorName": "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.

```
import { BaseModel } from "tina4-nodejs/orm";

export default class Task extends BaseModel {
  static tableName = "tasks";
  static softDelete = true;  // Enable soft delete

  static fields = {
    id:        { type: "integer" as const, primaryKey: true, autoIncrement: true },
    title:     { type: "string" as const, required: true },
    completed: { type: "boolean" as const, default: false },
    isDeleted: { type: "integer" as const, default: 0 },  // Required for soft delete (0 = activ
    createdAt: { type: "string" as const },
  };
}
```

When `static softDelete = 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.findById()` filter out records where `is_deleted = 1`
- `task.restore()` sets `is_deleted` back to `0` and makes the record visible again
- `task.forceDelete()` permanently removes the row from the database
- `Task.withTrashed()` includes soft-deleted records in query results

Deleting and Restoring

```
// Soft delete -- sets is_deleted = 1, row stays in the database
const task = Task.findById(1);
task.delete();

// Restore -- sets is_deleted = 0, record is visible again
task.restore();

// Permanently delete -- removes the row, no recovery possible
task.forceDelete();
```

`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()`, `findById()`) exclude soft-deleted records. When you need to see everything -- for admin dashboards, audit logs, or data recovery -- use `withTrashed()`:

```
// All tasks, including soft-deleted ones
const allTasks = Task.withTrashed();

// Soft-deleted tasks matching a condition
const deletedTasks = Task.withTrashed("completed = ?", [1]);
```

`withTrashed()` accepts the same filter parameters as `where()`: `withTrashed(conditions?, params?, limit?, offset?)`. The only difference: it ignores the `is_deleted` filter that standard queries apply.

Counting with Soft Delete

The `count()` method respects soft delete. It only counts non-deleted records:

```
const activeCount = Task.count();
const activeWork = Task.count("category = ?", ["work"]);
```

When to Use Soft Delete

Soft delete suits data that users might want to recover -- emails, documents, user accounts. It also serves audit requirements where regulations demand retention. For temporary data (sessions, cache entries, logs), hard delete keeps the table lean.

---

9. Auto-CRUD

Writing the same five REST endpoints for every model gets tedious. Auto-CRUD generates them from your model class. Define the model. Set the flag. Five routes appear.

The autoCrud Flag

Set `static autoCrud = true` on your model class:

```
import { BaseModel } from "tina4-nodejs/orm";

export default class Note extends BaseModel {
  static tableName = "notes";
```

```
  static autoCrud = true;  // Generates REST endpoints automatically

  static fields = {
    id:      { type: "integer" as const, primaryKey: true, autoIncrement: true },
    title:   { type: "string" as const, required: true },
    content: { type: "string" as const, default: "" },
  };
}
```

When the ORM discovers this model at startup, it registers CRUD routes. Five routes appear.

Manual Registration

You can also register routes explicitly using `generateCrudRoutes()`:

```
import { generateCrudRoutes } from "tina4-nodejs/orm";
```

Both approaches produce the same result:

Method	Path	Description
`GET`	`/api/notes`	List all with filtering and pagination
`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`.

What the Generated Routes Do

GET /api/notes returns paginated results with filtering and sorting:

```
curl "http://localhost:7148/api/notes?limit=10&page=1"
```

```
{
  "records": [...],
  "data": [...],
  "total": 42,
  "count": 42,
  "limit": 10,
  "page": 1,
  "perPage": 10,
  "totalPages": 5
}
```

The list endpoint supports query parameters:

- `?filter[field]=value` -- filter by field value
- `?sort=-name` -- sort by field (prefix `-` for descending)
- `?page=2&limit=10` -- pagination

POST /api/notes validates input ~~before saving:~~

```
curl -X POST http://localhost:7148/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:

```
{"error": "Validation failed", "statusCode": 422, "errors": ["title: This field is required"]}
```

DELETE /api/notes/1 respects soft delete. If the model has `static softDelete = 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 as a file. Auto-CRUD handles the rest.

Table Filter

The `static tableFilter` property adds a permanent WHERE condition to all auto-CRUD queries:

```
export default class ActiveUser extends BaseModel {
  static tableName = "users";
  static tableFilter = "active = 1";
  static autoCrud = true;
  // ...
}
```

Every auto-CRUD query for this model appends `AND active = 1`.

---

10. Scopes

Scopes are reusable query filters baked into the model. In TypeScript, you can define them as static methods:

```
import { BaseModel } from "tina4-nodejs/orm";

export default class BlogPost extends BaseModel {
  static tableName = "posts";
  static fields = {
    id:        { type: "integer" as const, primaryKey: true, autoIncrement: true },
    title:     { type: "string" as const, required: true },
    status:    { type: "string" as const, default: "draft" },
    createdAt: { type: "datetime" as const },
  };

  static published() {
    return this.where("status = ?", ["published"]);
  }

  static drafts() {
    return this.where("status = ?", ["draft"]);
  }
```

```
  static recent(days = 7) {
    return this.where(
      "created_at > datetime('now', ?)",
      [`-${days} days`],
    );
  }
}
```

Use them in your routes:

```
// src/routes/api/posts/published/get.ts
import type { Tina4Request, Tina4Response } from "tina4-nodejs";
import BlogPost from "../../../models/BlogPost.js";

export default async function (req: Tina4Request, res: Tina4Response) {
  const posts = BlogPost.published();
  res.json({ posts: posts.map((p) => p.toDict()) });
}
```

You can also register scopes dynamically with the `scope()` static method:

```
BlogPost.scope("active", "status != ?", ["archived"]);

// Now call it (cast needed since it's dynamically added):
const activePosts = (BlogPost as any).active();

// With limit and offset:
const activePosts2 = (BlogPost as any).active(10, 5);
```

`scope()` returns void. It registers a method on the class that calls `where()` with the given filter. The registered method accepts optional `limit` and `offset` parameters.

Scopes keep query logic in the model where it belongs. Route handlers stay thin.

---

11. Input Validation

Field definitions carry validation rules. Call `validate()` before `save()` and the ORM checks every constraint:

```
import { BaseModel } from "tina4-nodejs/orm";

export default class Product extends BaseModel {
  static tableName = "products";
  static fields = {
    id:       { type: "integer" as const, primaryKey: true, autoIncrement: true },
    name:     { type: "string" as const, required: true, minLength: 2, maxLength: 200 },
    sku:      { type: "string" as const, required: true, pattern: "^[A-Z]{2}-\\d{4}$" },  // e.g
    price:    { type: "number" as const, required: true, min: 0.01, max: 999999.99 },
    category: { type: "string" as const, choices: ["Electronics", "Kitchen", "Office", "Fitness"
  };
}
```

```
// src/routes/api/products/post.ts
import type { Tina4Request, Tina4Response } from "tina4-nodejs";
import Product from "../../models/Product.js";

export default async function (req: Tina4Request, res: Tina4Response) {
```

```
  const body = req.body as Record<string, unknown>;

  const product = new Product();
  product.name = body.name;
  product.sku = body.sku;
  product.price = body.price;
  product.category = body.category;

  const errors = product.validate();
  if (errors.length > 0) {
    return res.json({ errors }, 400);
  }

  product.save();
  res.json({ product: product.toDict() }, 201);
}
```

If validation fails, `validate()` returns an array of error strings:

```
{
  "errors": [
    "name Must be at least 2 characters",
    "sku Must match pattern ^[A-Z]{2}-\\d{4}$",
    "price Must be at least 0.01",
    "category Must be one of: Electronics, Kitchen, Office, Fitness"
  ]
}
```

---

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`, `createdAt`

BlogPost: `id`, `authorId` (integer foreign key), `title` (required, max 300), `slug` (required), `content`, `status` (choices: draft/published/archived, default draft), `createdAt`, `updatedAt`

Comment: `id`, `postId` (integer foreign key), `authorName` (required), `authorEmail` (required), `body` (required, min 5 chars), `createdAt`

- Build these endpoints:

Method	Path	Description
`POST`	`/api/authors`	Create an author
`GET`	`/api/authors/{id}`	Get author with their posts
`POST`	`/api/posts`	Create a post (requires authorId)

`GET`	`/api/posts`	List published posts with author info
`GET`	`/api/posts/{id}`	Get post with author and comments
`POST`	`/api/posts/{id}/comments`	Add comment to a post

---

13. Solution

Create `src/models/Author.ts`:

```
import { BaseModel } from "tina4-nodejs/orm";

export default class Author extends BaseModel {
  static tableName = "authors";
  static fields = {
    id:        { type: "integer" as const, primaryKey: true, autoIncrement: true },
    name:      { type: "string" as const, required: true, minLength: 2 },
    email:     { type: "string" as const, required: true },
    bio:       { type: "string" as const, default: "" },
    createdAt: { type: "datetime" as const },
  };
}
```

Create `src/models/BlogPost.ts`:

```
import { BaseModel } from "tina4-nodejs/orm";

export default class BlogPost extends BaseModel {
  static tableName = "posts";
  static fields = {
    id:        { type: "integer" as const, primaryKey: true, autoIncrement: true },
    authorId:  { type: "integer" as const, required: true },
    title:     { type: "string" as const, required: true, maxLength: 300 },
    slug:      { type: "string" as const, required: true },
    content:   { type: "string" as const, default: "" },
    status:    { type: "string" as const, default: "draft", choices: ["draft", "published", "arc
    createdAt: { type: "datetime" as const },
    updatedAt: { type: "datetime" as const },
  };

  static published() {
    return this.where("status = ?", ["published"]);
  }
}
```

Create `src/models/Comment.ts`:

```
import { BaseModel } from "tina4-nodejs/orm";

export default class Comment extends BaseModel {
  static tableName = "comments";
  static fields = {
```

```
    id:          { type: "integer" as const, primaryKey: true, autoIncrement: true },
    postId:      { type: "integer" as const, required: true },
    authorName:  { type: "string" as const, required: true },
    authorEmail: { type: "string" as const, required: true },
    body:        { type: "string" as const, required: true, minLength: 5 },
    createdAt:   { type: "datetime" as const },
  };
}
```

Create `src/routes/api/authors/post.ts`:

```
import type { Tina4Request, Tina4Response } from "tina4-nodejs";
import Author from "../../../models/Author.js";

export default async function (req: Tina4Request, res: Tina4Response) {
  const body = req.body as Record<string, unknown>;

  const author = new Author();
  author.name = body.name;
  author.email = body.email;
  author.bio = body.bio ?? "";

  const errors = author.validate();
  if (errors.length > 0) {
    return res.json({ errors }, 400);
  }

  author.save();
  res.json({ author: author.toDict() }, 201);
}
```

Create `src/routes/api/authors/[id]/get.ts`:

```
import type { Tina4Request, Tina4Response } from "tina4-nodejs";
import Author from "../../../../models/Author.js";
import BlogPost from "../../../../models/BlogPost.js";

export default async function (req: Tina4Request, res: Tina4Response) {
  const author = Author.findById(req.params.id);

  if (author === null) {
    return res.json({ error: "Author not found" }, 404);
  }

  const posts = BlogPost.where("author_id = ?", [author.id]);

  const data = author.toDict();
  data.posts = posts.map((p) => p.toDict());

  res.json(data);
}
```

Create `src/routes/api/posts/post.ts`:

```
import type { Tina4Request, Tina4Response } from "tina4-nodejs";
import Author from "../../../models/Author.js";
import BlogPost from "../../../models/BlogPost.js";

export default async function (req: Tina4Request, res: Tina4Response) {
  const body = req.body as Record<string, unknown>;
```

```
  // Verify author exists
  const author = Author.findById(body.authorId);
  if (author === null) {
    return res.json({ error: "Author not found" }, 404);
  }

  const post = new BlogPost();
  post.authorId = body.authorId;
  post.title = body.title;
  post.slug = body.slug;
  post.content = body.content ?? "";
  post.status = body.status ?? "draft";

  const errors = post.validate();
  if (errors.length > 0) {
    return res.json({ errors }, 400);
  }

  post.save();
  res.json({ post: post.toDict() }, 201);
}
```

Create `src/routes/api/posts/get.ts`:

```
import type { Tina4Request, Tina4Response } from "tina4-nodejs";
import Author from "../../../models/Author.js";
import BlogPost from "../../../models/BlogPost.js";

export default async function (req: Tina4Request, res: Tina4Response) {
  const posts = BlogPost.published();
  const data = [];

  for (const p of posts) {
    const postDict = p.toDict();
    const author = p.belongsTo(Author, "author_id");
    postDict.author = author ? author.toDict() : null;
    data.push(postDict);
  }

  res.json({ posts: data, count: data.length });
}
```

Create `src/routes/api/posts/[id]/get.ts`:

```
import type { Tina4Request, Tina4Response } from "tina4-nodejs";
import Author from "../../../../models/Author.js";
import BlogPost from "../../../../models/BlogPost.js";
import Comment from "../../../../models/Comment.js";

export default async function (req: Tina4Request, res: Tina4Response) {
  const post = BlogPost.findById(req.params.id);

  if (post === null) {
    return res.json({ error: "Post not found" }, 404);
  }

  const author = post.belongsTo(Author, "author_id");
  const comments = post.hasMany(Comment, "post_id");

  const data = post.toDict();
```

```
  data.author = author ? author.toDict() : null;
  data.comments = comments.map((c) => c.toDict());
  data.commentCount = comments.length;

  res.json(data);
}
```

Create `src/routes/api/posts/[id]/comments/post.ts`:

```
import type { Tina4Request, Tina4Response } from "tina4-nodejs";
import BlogPost from "../../../../../models/BlogPost.js";
import Comment from "../../../../../models/Comment.js";

export default async function (req: Tina4Request, res: Tina4Response) {
  const post = BlogPost.findById(req.params.id);

  if (post === null) {
    return res.json({ error: "Post not found" }, 404);
  }

  const body = req.body as Record<string, unknown>;

  const comment = new Comment();
  comment.postId = req.params.id;
  comment.authorName = body.authorName;
  comment.authorEmail = body.authorEmail;
  comment.body = body.body;

  const errors = comment.validate();
  if (errors.length > 0) {
    return res.json({ errors }, 400);
  }

  comment.save();
  res.json({ comment: comment.toDict() }, 201);
}
```

---

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 TypeScript object. The database remains unchanged until you call `note.save()`.

Fix: Call `save()` after modifying properties. Check the return value -- `save()` returns `this` on success and `false` on failure.

2. findById() returns null

Problem: You call `Note.findById(id)` but get `null` instead of a note object.

Cause: `findById()` returns `null` when no row matches the given primary key. If soft delete is enabled, `findById()` also excludes soft-deleted records.

Fix: Check for `null` after `findById()`: `if (note === null) return res.json({error: "Not found"}, 404)`. Use `findOrFail()` if you want an Error thrown instead.

3. find() vs findById()

Problem: You call `Note.find(42)` expecting a single record, but get unexpected results.

Cause: `find()` takes a filter object (`find({ id: 42 })`), not a bare primary key value. For single-record lookups by primary key, use `findById(42)`.

Fix: Use `findById(id)` for primary key lookups. Use `find({ column: value })` for filter-based queries.

4. all() not findAll()

Problem: You call `Note.findAll()` and get an error.

Cause: The method is `all()`, not `findAll()`. There is no `findAll()` method on BaseModel.

Fix: Use `Note.all()` to fetch all records.

5. toDict() includes everything

Problem: `user.toDict()` includes `passwordHash` in the API response.

Cause: `toDict()` includes all fields by default.

Fix: Build the response object manually, omitting sensitive fields: `{ id: user.id, name: user.name, email: user.email }`. Or create a helper method on your model class that returns only safe fields.

6. Validation only runs on validate()

Problem: You call `save()` without calling `validate()` first, and invalid data gets into the database.

Cause: `save()` does not validate. This is by design -- sometimes you need to save partial data or bypass validation for bulk operations.

Fix: Call `const errors = model.validate()` before `save()` in your route handlers. Or create a helper method that validates and saves in one step.

7. Foreign key not enforced

Problem: You save a post with `authorId = 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 `hasMany`/`belongsTo` methods, but the database itself may not enforce it.

Fix: Enable SQLite foreign keys with `PRAGMA foreign_keys = ON;` in a migration, or validate the foreign key in your route handler before saving.

8. N+1 query problem

Problem: Listing 100 authors with their posts runs 101 queries (1 for authors + 100 for posts), and the page loads slowly.

Cause: You call `author.hasMany(BlogPost, "author_id")` inside a loop for each author.

Fix: Use eager loading with the `include` parameter on `all()`, `where()`, or `findById()`. Define declarative relationships on the model and register models with `BaseModel.registerModel()`.

9. Auto-CRUD endpoint conflicts

Problem: Custom route at `/api/notes/{id}` stops working after enabling auto-CRUD for the Note model.

Cause: Both routes match the same path. The first registered route wins.

Fix: Custom routes in `src/routes/` load before auto-CRUD routes. They take precedence. If you want different behaviour, use a different path for the custom route.

10. Soft-deleted records appearing in queries

Problem: You soft-deleted a record, but queries still return it.

Cause: Soft delete requires the `static softDelete = true` flag on the model class and an `is_deleted` column in the database (integer, 0 or 1). Without both, soft delete is inactive.

Fix: Verify both the `static softDelete = true` flag and the `is_deleted` column exist. The column stores `0` for active records and `1` for deleted ones.

---

15. QueryBuilder Integration

ORM models provide a `query()` static 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
const results = User.query()
  .select("id", "name", "email")
  .where("active = ?", [1])
  .orderBy("name")
  .limit(50)
  .get();

// First matching record
const user = User.query()
  .where("email = ?", ["alice@example.com"])
  .first();

// Count
const total = User.query()
  .where("role = ?", ["admin"])
  .count();
```

```
// Check existence
const exists = User.query()
  .where("email = ?", ["test@example.com"])
  .exists();
```

The `limit()` method accepts count and an optional offset: `limit(10, 20)`. There is no separate `offset()` method.

Note that `get()` returns plain objects, not model instances. Use `findById()`, `find()`, `all()`, or `where()` when you need model instances with `save()`, `delete()`, and relationship methods.

See the QueryBuilder chapter for the full fluent API including joins, grouping, having, and MongoDB support.

Multiple Database Connections

A model can target a named database connection using `static _db`:

```
export default class AuditLog extends BaseModel {
  static tableName = "audit_logs";
  static _db = "secondary";  // Uses the "secondary" named adapter
  static fields = {
    id:    { type: "integer" as const, primaryKey: true, autoIncrement: true },
    event: { type: "string" as const, required: true },
  };
}
```

Register the named adapter at startup with `initDatabase()`.

QueryBuilder

1. Why a Query Builder?

In Chapter 5 you wrote raw SQL. In Chapter 6 you used ORM models with `select()`. Both work. Both have limits.

Raw SQL gives you full control but it is brittle. Concatenate a WHERE clause wrong and you get a syntax error -- or worse, a SQL injection. The ORM's `select()` method handles simple cases, but building dynamic queries with optional filters, joins, and aggregations turns into a mess of string concatenation and empty-parameter guards.

The QueryBuilder sits between raw SQL and the ORM. It gives you a fluent, chainable API that builds correct SQL. You write TypeScript. It writes SQL. Every method returns `this`, so you chain calls in any order. When you are done, call `get()` to execute or `toSql()` to inspect.

---

2. Creating a QueryBuilder

There are two entry points.

Standalone: QueryBuilder.fromTable()

Import `QueryBuilder` from the ORM package and call the static `from()` factory:

```
import { QueryBuilder } from "tina4-nodejs/orm";

const users = QueryBuilder.fromTable("users", db)
    .select("id", "name", "email")
    .where("active = ?", [1])
    .orderBy("name ASC")
    .limit(10)
    .get();
```

The first argument is the table name. The second is the database adapter -- the same `db` object you get from `Database.create()` or the globally initialised adapter. If you omit the adapter, the QueryBuilder falls back to the global adapter registered by `initDatabase()`.

From an ORM Model: Model.query()

Every model that extends `BaseModel` has a static `query()` method. It returns a QueryBuilder pre-configured with the model's table name and database adapter:

```
import { User } from "../orm/User";

const activeUsers = User.query()
    .where("active = ?", [1])
    .orderBy("name ASC")
    .get();
```

No need to pass the table name or database. The model knows both.

---

3. Selecting Columns

By default, the QueryBuilder selects all columns (`*`). Use `select()` to pick specific columns:

```
QueryBuilder.fromTable("products", db)
    .select("id", "name", "price")
    .get();
```

This generates:

```
SELECT id, name, price FROM products
```

Pass as many column names as you need. Each is a separate string argument, not a comma-separated string. If you call `select()` with no arguments, it keeps the default `*`.

You can also use expressions:

```
QueryBuilder.fromTable("orders", db)
    .select("customer_id", "SUM(total) as revenue")
    .groupBy("customer_id")
    .get();
```

---

4. WHERE Conditions

where() -- AND

Add a condition with `where()`. Use `?` as the placeholder for parameters:

```
QueryBuilder.fromTable("users", db)
    .where("age > ?", [18])
    .where("active = ?", [1])
    .get();
```

This generates:

```
SELECT * FROM users WHERE age > ? AND active = ?
```

With parameters `[18, 1]`.

Multiple `where()` calls are joined with AND. The first call starts the WHERE clause. Every subsequent call adds `AND condition`.

orWhere() -- OR

Use `orWhere()` to add an OR condition:

```
QueryBuilder.fromTable("products", db)
    .where("category = ?", ["Electronics"])
    .orWhere("category = ?", ["Books"])
    .get();
```

This generates:

```
SELECT * FROM products WHERE category = ? OR category = ?
```

Combining AND and OR

```
QueryBuilder.fromTable("products", db)
    .where("price < ?", [100])
    .where("in_stock = ?", [1])
    .orWhere("featured = ?", [1])
    .get();
```

Generates:

```
SELECT * FROM products WHERE price < ? AND in_stock = ? OR featured = ?
```

If you need grouping with parentheses, write the grouped expression as a single condition:

```
QueryBuilder.fromTable("products", db)
    .where("price < ?", [100])
    .where("(category = ? OR category = ?)", ["Electronics", "Books"])
    .get();
```

Generates:

```
SELECT * FROM products WHERE price < ? AND (category = ? OR category = ?)
```

---

5. Joins

Inner Join

```
QueryBuilder.fromTable("orders", db)
    .select("orders.id", "users.name", "orders.total")
    .join("users", "users.id = orders.user_id")
    .get();
```

Generates:

```
SELECT orders.id, users.name, orders.total FROM orders INNER JOIN users ON users.id = orders.use
```

Left Join

```
QueryBuilder.fromTable("users", db)
    .select("users.name", "orders.total")
    .leftJoin("orders", "orders.user_id = users.id")
    .get();
```

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:

```
QueryBuilder.fromTable("orders", db)
    .select("orders.id", "users.name", "products.name as product_name")
    .join("users", "users.id = orders.user_id")
    .join("order_items", "order_items.order_id = orders.id")
    .join("products", "products.id = order_items.product_id")
    .where("orders.status = ?", ["completed"])
    .get();
```

---

6. Grouping and Aggregation

groupBy()

```
QueryBuilder.fromTable("orders", db)
    .select("status", "COUNT(*) as total")
    .groupBy("status")
    .get();
```

Generates:

```
SELECT status, COUNT(*) as total FROM orders GROUP BY status
```

Call `groupBy()` multiple times to group by multiple columns:

```
QueryBuilder.fromTable("orders", db)
    .select("status", "customer_id", "SUM(total) as revenue")
    .groupBy("status")
    .groupBy("customer_id")
    .get();
```

having()

Filter grouped results with `having()`:

```
QueryBuilder.fromTable("products", db)
    .select("category", "AVG(price) as avg_price")
    .groupBy("category")
    .having("AVG(price) > ?", [50])
    .get();
```

Generates:

```
SELECT category, AVG(price) as avg_price FROM products GROUP BY category HAVING AVG(price) > ?
```

Multiple `having()` calls are joined with AND.

---

7. Ordering

```
QueryBuilder.fromTable("products", db)
    .orderBy("price DESC")
    .get();
```

Generates:

```
SELECT * FROM products ORDER BY price DESC
```

Call `orderBy()` multiple times for multi-column sorting:

```
QueryBuilder.fromTable("products", db)
    .orderBy("category ASC")
    .orderBy("price DESC")
    .get();
```

Generates:

```
SELECT * FROM products ORDER BY category ASC, price DESC
```

---

8. Limit and Offset

Limit Only

```
QueryBuilder.fromTable("products", db)
    .limit(10)
    .get();
```

Limit with Offset

```
QueryBuilder.fromTable("products", db)
    .limit(10, 20)
    .get();
```

The first argument is the maximum number of rows. The second is the number of rows to skip. This is how you implement pagination:

```
const page = 3;
const perPage = 25;
const offset = (page - 1) * perPage;

const products = QueryBuilder.fromTable("products", db)
    .orderBy("name ASC")
    .limit(perPage, offset)
    .get();
```

---

9. Executing Queries

get\<T\>() -- All Rows

`get()` executes the query and returns an array of row objects:

```
const users = QueryBuilder.fromTable("users", db)
    .where("active = ?", [1])
    .get();
// users: Record<string, unknown>[]
```

Use the TypeScript generic to type the result:

```
interface User {
    id: number;
    name: string;
    email: string;
    active: boolean;
}

const users = QueryBuilder.fromTable("users", db)
    .where("active = ?", [1])
    .get<User>();
// users: User[]
```

The generic is optional. Without it, rows are typed as `Record<string, unknown>[]`.

first\<T\>() -- Single Row

`first()` returns one row or `null`:

```
const user = QueryBuilder.fromTable("users", db)
    .where("email = ?", ["alice@example.com"])
    .first<User>();

if (!user) {
    // Not found
}
```

count() -- Row Count

`count()` returns the number of matching rows without fetching the data:

```
const total = QueryBuilder.fromTable("users", db)
    .where("active = ?", [1])
    .count();
// total: number
```

It rewrites the query internally to `SELECT COUNT(*) as cnt` and extracts the value.

exists() -- Boolean Check

`exists()` returns `true` if at least one row matches:

```
const hasAdmin = QueryBuilder.fromTable("users", db)
    .where("role = ?", ["admin"])
    .exists();
// hasAdmin: boolean
```

---

10. Inspecting SQL with toSql()

Call `toSql()` to get the generated SQL string without executing the query. Useful for debugging:

```
const sql = QueryBuilder.fromTable("users", db)
    .select("id", "name")
    .where("active = ?", [1])
    .orderBy("name ASC")
    .limit(10)
    .toSql();

console.log(sql);
// SELECT id, name FROM users WHERE active = ? ORDER BY name ASC
```

Note that `toSql()` returns the SQL with `?` placeholders. The actual parameter values are bound at execution time by the database adapter.

The `LIMIT` and `OFFSET` clauses are not included in `toSql()` output -- they are passed directly to the adapter's `fetch()` method as separate arguments.

---

11. Using QueryBuilder in Route Handlers

The QueryBuilder pairs naturally with route handlers and `res.json()`:

List Endpoint with Filters

```
// src/routes/api/products/get.ts
import { QueryBuilder } from "tina4-nodejs/orm";
import type { Tina4Request, Tina4Response } from "tina4-nodejs";

export default async function (req: Tina4Request, res: Tina4Response) {
    const category = req.query.category;
    const minPrice = req.query.min_price;
    const sort = req.query.sort ?? "name";
    const page = parseInt(req.query.page ?? "1", 10);
    const perPage = parseInt(req.query.per_page ?? "20", 10);

    const qb = QueryBuilder.fromTable("products");

    if (category) {
        qb.where("category = ?", [category]);
    }

    if (minPrice) {
        qb.where("price >= ?", [parseFloat(minPrice)]);
    }

    const allowedSorts = ["name", "price", "category", "created_at"];
    const safeSort = allowedSorts.includes(sort) ? sort : "name";

    const total = qb.count();
    const products = qb
        .orderBy(`${safeSort} ASC`)
        .limit(perPage, (page - 1) * perPage)
        .get();

    return res.json({ products, total, page, per_page: perPage });
}
```

Single Record Lookup

```
// src/routes/api/users/[id]/get.ts
import { QueryBuilder } from "tina4-nodejs/orm";
import type { Tina4Request, Tina4Response } from "tina4-nodejs";

export default async function (req: Tina4Request, res: Tina4Response) {
    const user = QueryBuilder.fromTable("users")
        .where("id = ?", [req.params.id])
        .first();

    if (!user) {
        return res.status(404).json({ error: "User not found" });
    }

    return res.json(user);
}
```

Dashboard with Aggregation

```
// src/routes/api/dashboard/get.ts
import { QueryBuilder } from "tina4-nodejs/orm";
import type { Tina4Request, Tina4Response } from "tina4-nodejs";

export default async function (req: Tina4Request, res: Tina4Response) {
    const totalUsers = QueryBuilder.fromTable("users").count();

    const revenueByCategory = QueryBuilder.fromTable("orders")
        .select("category", "SUM(total) as revenue", "COUNT(*) as order_count")
        .join("order_items", "order_items.order_id = orders.id")
        .join("products", "products.id = order_items.product_id")
        .where("orders.status = ?", ["completed"])
        .groupBy("category")
        .having("SUM(total) > ?", [0])
        .orderBy("revenue DESC")
        .get();

    const recentOrders = QueryBuilder.fromTable("orders")
        .select("orders.id", "users.name as customer", "orders.total", "orders.created_at")
        .join("users", "users.id = orders.user_id")
        .orderBy("orders.created_at DESC")
        .limit(5)
        .get();

    return res.json({
        total_users: totalUsers,
        revenue_by_category: revenueByCategory,
        recent_orders: recentOrders,
    });
}
```

---

12. QueryBuilder vs ORM select()

Both query the database. When do you use which?

Scenario	Use
Simple CRUD on a single model	ORM `select()`
Need `toDict()` / `toObject()` on results	ORM `select()`
Eager loading relationships	ORM `select()` with `include`
Multi-table joins	QueryBuilder
Aggregations (SUM, COUNT, AVG)	QueryBuilder
Complex dynamic filters	QueryBuilder
Sub-selects in columns	QueryBuilder
You want typed results without a model	QueryBuilder with `get<T>()`

They are not mutually exclusive. Use ORM models for data that maps cleanly to a single table. Use the QueryBuilder when you need to reach across tables or aggregate.

---

13. Exercise: Sales Report API

Build a sales report endpoint using the QueryBuilder.

Setup

Assume these tables exist:

- `customers` -- id, name, email, region, created_at
- `orders` -- id, customer*id, status, total, created*at
- `order_items` -- id, order*id, product*id, quantity, unit_price
- `products` -- id, name, category, price

Requirements

Create three route files:

Method	Path	Description
`GET`	`/api/reports/revenue`	Revenue by product category with optional date range
`GET`	`/api/reports/top-customers`	Top 10 customers by total spend
`GET`	`/api/reports/summary`	Order count, total revenue, average order value

Query parameters for `/api/reports/revenue`:

- `after` -- only orders after this date (e.g. `2025-01-01`)
- `before` -- only orders before this date
- `min_revenue` -- only categories with revenue above this amount

---

14. Solution

Revenue by Category

Create `src/routes/api/reports/revenue/get.ts`:

```
import { QueryBuilder } from "tina4-nodejs/orm";
import type { Tina4Request, Tina4Response } from "tina4-nodejs";

export default async function (req: Tina4Request, res: Tina4Response) {
    const qb = QueryBuilder.fromTable("order_items")
```

```
        .select(
            "products.category",
            "SUM(order_items.quantity * order_items.unit_price) as revenue",
            "SUM(order_items.quantity) as units_sold",
            "COUNT(DISTINCT orders.id) as order_count"
        )
        .join("orders", "orders.id = order_items.order_id")
        .join("products", "products.id = order_items.product_id")
        .where("orders.status = ?", ["completed"]);

    if (req.query.after) {
        qb.where("orders.created_at >= ?", [req.query.after]);
    }

    if (req.query.before) {
        qb.where("orders.created_at < ?", [req.query.before]);
    }

    qb.groupBy("products.category");

    if (req.query.min_revenue) {
        qb.having("SUM(order_items.quantity * order_items.unit_price) > ?", [
            parseFloat(req.query.min_revenue),
        ]);
    }

    const results = qb.orderBy("revenue DESC").get();

    return res.json({ categories: results });
}
```

Top Customers

Create `src/routes/api/reports/top-customers/get.ts`:

```
import { QueryBuilder } from "tina4-nodejs/orm";
import type { Tina4Request, Tina4Response } from "tina4-nodejs";

export default async function (req: Tina4Request, res: Tina4Response) {
    const customers = QueryBuilder.fromTable("orders")
        .select(
            "customers.id",
            "customers.name",
            "customers.email",
            "customers.region",
            "SUM(orders.total) as total_spend",
            "COUNT(orders.id) as order_count"
        )
        .join("customers", "customers.id = orders.customer_id")
        .where("orders.status = ?", ["completed"])
        .groupBy("customers.id")
        .groupBy("customers.name")
        .groupBy("customers.email")
        .groupBy("customers.region")
        .orderBy("total_spend DESC")
        .limit(10)
        .get();

    return res.json({ top_customers: customers });
```

```
}
```

Summary

Create `src/routes/api/reports/summary/get.ts`:

```
import { QueryBuilder } from "tina4-nodejs/orm";
import type { Tina4Request, Tina4Response } from "tina4-nodejs";

export default async function (req: Tina4Request, res: Tina4Response) {
    const stats = QueryBuilder.fromTable("orders")
        .select(
            "COUNT(*) as order_count",
            "SUM(total) as total_revenue",
            "AVG(total) as avg_order_value"
        )
        .where("status = ?", ["completed"])
        .first();

    const ordersByStatus = QueryBuilder.fromTable("orders")
        .select("status", "COUNT(*) as count")
        .groupBy("status")
        .orderBy("count DESC")
        .get();

    const hasOrders = QueryBuilder.fromTable("orders").exists();

    return res.json({
        summary: stats,
        by_status: ordersByStatus,
        has_orders: hasOrders,
    });
}
```

Testing It

```
# Revenue by category
curl "http://localhost:7148/api/reports/revenue"

# Revenue with date range and minimum
curl "http://localhost:7148/api/reports/revenue?after=2025-01-01&before=2026-01-01&min_revenue=1

# Top customers
curl "http://localhost:7148/api/reports/top-customers"

# Summary
curl "http://localhost:7148/api/reports/summary"
```

---

15. NoSQL: MongoDB Queries

The QueryBuilder can generate MongoDB-compatible query documents with `toMongo()`. This returns an object containing the filter, projection, sort, limit, and skip -- ready to pass to the MongoDB Node.js 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

```
const query = QueryBuilder.fromTable("users")
    .select("name", "email")
    .where("age > ?", [25])
    .where("status = ?", ["active"])
    .orderBy("name ASC")
    .limit(10)
    .offset(5);

const mongo = query.toMongo();
```

The returned object:

```
{
    filter: { age: { $gt: 25 }, status: "active" },
    projection: { name: 1, email: 1 },
    sort: { name: 1 },
    limit: 10,
    skip: 5,
}
```

Pass it directly to the MongoDB driver:

```
const collection = db.collection("users");
const cursor = collection.find(mongo.filter, {
    projection: mongo.projection,
    sort: mongo.sort,
    limit: mongo.limit,
    skip: mongo.skip,
});
```

---

16. Gotchas

1. Forgetting the Database Adapter

Problem: You call `QueryBuilder.fromTable("users").get()` and get `"QueryBuilder: No database adapter provided."`.

Cause: No database adapter was passed to `from()` and no global adapter has been initialised.

Fix: Either pass the adapter explicitly -- `QueryBuilder.fromTable("users", db)` -- or ensure `initDatabase()` has been called before any queries run. When Tina4 boots your server, the global adapter is initialised automatically from `TINA4_DATABASE_URL`.

2. Parameter Order Matters

Problem: Your WHERE clause has three `?` placeholders but the results are wrong.

Cause: Parameters are collected in the order you call `where()` and `orWhere()`. If you add conditions in one order but pass parameters assuming a different order, values bind to the wrong placeholders.

Fix: Each `where()` call takes its own parameter array. The QueryBuilder concatenates them in call order. Keep the parameters next to their condition:

```
// Correct
qb.where("age > ?", [18]).where("country = ?", ["ZA"]);

// Wrong -- do not batch parameters separately from conditions
```

3. toSql() Does Not Include LIMIT

Problem: You call `toSql()` and the output has no LIMIT clause, but `get()` returns limited rows.

Cause: LIMIT and OFFSET are passed to the database adapter's `fetch()` method as separate arguments, not appended to the SQL string.

Fix: This is by design. The adapter handles LIMIT/OFFSET according to the database engine's dialect. Use `toSql()` for debugging the WHERE, JOIN, and ORDER BY clauses.

4. Reusing a QueryBuilder

Problem: You call `count()` and then `get()` on the same QueryBuilder, but the results are inconsistent.

Cause: `count()` temporarily replaces the selected columns with `COUNT(*)` and restores them. This is safe for a single call, but interleaving multiple execution methods on the same builder can produce unexpected SQL if you modify the builder between calls.

Fix: For separate queries, create separate builders. Use `QueryBuilder.fromTable()` or `Model.query()` fresh for each independent query.

5. Mixing QueryBuilder with ORM select()

Problem: You expect `get()` to return model instances with `toDict()`.

Cause: The QueryBuilder returns plain objects (`Record<string, unknown>` or your generic type `T`). It does not return model instances.

Fix: Use `get<T>()` with an interface to type the results. If you need model methods like `toDict()`, `save()`, or relationship loading, use the ORM's `select()` method instead.

6. OR Without AND

Problem: You start with `orWhere()` instead of `where()` and the query behaves unexpectedly.

Cause: The first condition in the WHERE clause ignores the connector (AND/OR). Starting with `orWhere()` works syntactically but reads misleadingly.

Fix: Always start with `where()` for the first condition. Use `orWhere()` for subsequent alternatives.

Authentication

1. Locking the Door

Every endpoint you have built so far is public. Anyone with the URL can read, create, update, and delete data. Fine for a tutorial. Unacceptable in production.

A real application needs identity. Who is making this request? And are they allowed to make it?

This chapter covers Tina4's authentication system: JWT tokens, password hashing, middleware-based route protection, CSRF tokens for forms, and session management.

---

2. JWT Tokens

Tina4 uses JSON Web Tokens (JWT) for authentication. A JWT is a signed string carrying a payload -- a user ID, a role, whatever you need. The server creates the token at login. The client sends it with every request. The server verifies it without touching the database.

Generating a Token

```
import { Auth } from "tina4-nodejs";

const payload = {
    user_id: 42,
    email: "alice@example.com",
    role: "admin"
};

const token = Auth.getToken(payload, secret);
```

`getToken()` signs the payload with HS256 (HMAC-SHA256) and returns a JWT string like:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VyX2lkIjo0MiwiZW1haWwiOiJhbGljZUBleGFtcGxlLmNvbSIsInJ
```

The token has three parts separated by dots: header, payload, and signature. The signature ensures the token has not been tampered with.

The `secret` parameter is required -- pass your secret key directly or read it from the `TINA4_SECRET` env var.

> ***Legacy aliases:*** `Auth.getToken()` *and the standalone* `getToken()` *function still work. Use the primary name in new code.*

Token Expiry

Pass the expiry time as the third argument to `getToken()` in **minutes**:

```
const token = Auth.getToken(payload, secret, 60); // 1 hour (default)
```

Value	Duration
`15`	15 minutes

`60`	1 hour (default)
`1440`	24 hours
`10080`	7 days

You can also configure the default expiry in `.env`:

```
TINA4_TOKEN_LIMIT=1440
```

The value is in minutes. `1440` is 24 hours.

Validating a Token

```
const payload = Auth.validToken(token, secret);
// Returns the payload object if valid, null if invalid or expired
```

`validToken()` returns the decoded payload on success, not a boolean. This lets you validate and read the token in one step. Returns `null` if the token is invalid or expired.

> ***Legacy alias:*** `Auth.validateToken()` *and the standalone* `validToken()` *function work the same way.*

Reading the Payload

```
const payload = Auth.getPayload(token);
```

Returns the decoded payload **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, `getPayload()` returns `null`.

> ***Important:*** `getPayload()` *does not verify the signature or check expiry. Use* `validToken()` *when you need to confirm the token is trustworthy.*

The Secret Key and Algorithm

Tina4 Node.js uses **HS256** (HMAC-SHA256) for JWT signing. It uses only the standard library -- zero external dependencies.

Set the secret key in `.env`:

```
TINA4_SECRET=my-super-secret-key-at-least-32-chars
```

The `secret` parameter on `getToken()` and `validToken()` is optional -- if omitted, Tina4 reads from the `TINA4_SECRET` env var. If neither is set, Tina4 falls back to generating a random key at `secrets/jwt.key` on first run. Setting `TINA4_SECRET` explicitly is recommended for production so all server instances share the same key.

Keep this key secret. If someone gets it, they can forge tokens.

---

3. Password Hashing

Plain text passwords are a breach waiting to happen. Tina4 provides two functions for secure password handling.

Hashing a Password

```
import { Auth } from "tina4-nodejs";

const hash = await Auth.hashPassword("my-secure-password");
```

Uses PBKDF2 from the standard library -- no external dependencies. Each hash includes a random salt, so hashing the same password twice produces different results.

Checking a Password

```
const isCorrect = await Auth.checkPassword("my-secure-password", storedHash);
// Returns true if the password matches the hash
```

Registration Example

```
import { Router, Auth } from "tina4-nodejs";
import { Database } from "tina4-nodejs/orm";

/**
 * @noauth
 */
Router.post("/api/register", async (req, res) => {
    const body = req.body;

    // Validate input
    if (!body.name || !body.email || !body.password) {
        return res.status(400).json({ error: "Name, email, and password are required" });
    }

    if (body.password.length < 8) {
        return res.status(400).json({ error: "Password must be at least 8 characters" });
    }

    const db = Database.getConnection();

    // Check if email already exists
    const existing = await db.fetchOne("SELECT id FROM users WHERE email = :email", { email: bod
    if (existing !== null) {
        return res.status(409).json({ error: "Email already registered" });
    }

    // Hash the password
    const passwordHash = await Auth.hashPassword(body.password);

    // Create the user
    await db.execute(
        "INSERT INTO users (name, email, password_hash) VALUES (:name, :email, :hash)",
        { name: body.name, email: body.email, hash: passwordHash }
    );

    const user = await db.fetchOne("SELECT id, name, email FROM users WHERE id = last_insert_row

    return res.status(201).json({ message: "Registration successful", user });
});
```

```
curl -X POST http://localhost:7148/api/register \
  -H "Content-Type: application/json" \
  -d '{"name": "Alice", "email": "alice@example.com", "password": "securePass123"}'
```

```
{
  "message": "Registration successful",
  "user": {"id": 1, "name": "Alice", "email": "alice@example.com"}
}
```

---

4. The Login Flow

The complete login flow. Client sends credentials. Server validates them. Server returns a JWT token.

```
import { Router, Auth } from "tina4-nodejs";
```

```
import { Database } from "tina4-nodejs/orm";

/**
 * @noauth
 */
Router.post("/api/login", async (req, res) => {
    const body = req.body;

    if (!body.email || !body.password) {
        return res.status(400).json({ error: "Email and password are required" });
    }

    const db = Database.getConnection();

    // Find the user
    const user = await db.fetchOne(
        "SELECT id, name, email, password_hash FROM users WHERE email = :email",
        { email: body.email }
    );

    if (user === null) {
        return res.status(401).json({ error: "Invalid email or password" });
    }

    // Check the password
    if (!(await Auth.checkPassword(body.password, user.password_hash))) {
        return res.status(401).json({ error: "Invalid email or password" });
    }

    // Generate a token
    const secret = process.env.TINA4_SECRET || "tina4-default-secret";
    const token = Auth.getToken({
        user_id: user.id,
        email: user.email,
        name: user.name
    }, secret);

    return res.json({
        message: "Login successful",
        token,
        user: { id: user.id, name: user.name, email: user.email }
    });
});
```

Notice `@noauth` in the JSDoc comment. The login endpoint must be public. You cannot require a token to get a token.

```
curl -X POST http://localhost:7148/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:7148/api/profile \
  -H "Authorization: Bearer eyJhbGciOiJIUzI1NiIs..."
```

Or in frontend JavaScript:

```
// Frontend JavaScript
const response = await fetch("/api/profile", {
    headers: {
        "Authorization": `Bearer ${token}`,
        "Content-Type": "application/json",
    },
});
```

The token travels in the `Authorization` header with the `Bearer` prefix. The middleware extracts it, verifies it, and populates `req.user` with the decoded payload.

---

6. Protecting Routes

Auth Middleware

Create a reusable auth middleware:

```
import { Auth } from "tina4-nodejs";

function authMiddleware(req, res, next) {
    const authHeader = req.headers["authorization"] ?? "";

    if (!authHeader || !authHeader.startsWith("Bearer ")) {
        res({ error: "Authorization header required" }, 401);
        return;
    }

    const token = authHeader.substring(7);  // Remove "Bearer " prefix
    const secret = process.env.TINA4_SECRET || "tina4-default-secret";

    const payload = Auth.validToken(token, secret);
    if (payload === null) {
        res({ error: "Invalid or expired token" }, 401);
        return;
    }

    (req as any).user = payload;  // Attach user data to the request

    next();
}
```

Applying Middleware to Routes

```
import { Router } from "tina4-nodejs";

Router.get("/api/profile", async (req, res) => {
    return res.json({
        user_id: (req as any).user.user_id,
        email: (req as any).user.email,
        name: (req as any).user.name
    });
}, [authMiddleware]);
```

```
# Without token -- 401
curl http://localhost:7148/api/profile
```

```
{"error":"Authorization header required"}
```

```
# With valid token -- 200
curl http://localhost:7148/api/profile \
  -H "Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9..."
```

```
{
  "user_id": 1,
  "email": "alice@example.com",
  "name": "Alice"
}
```

Applying Middleware to a Group

```
import { Router } from "tina4-nodejs";

Router.group("/api/admin", [authMiddleware], (router) => {

    router.get("/stats", async (req, res) => {
        return res.json({ active_users: 42 });
    });

    router.get("/logs", async (req, res) => {
        return res.json({ logs: [] });
    });

});
```

Every route in the group requires a valid token.

---

7. @noauth and @secured Decorators

These JSDoc annotations control authentication at the route level.

@noauth -- Skip Authentication

Use `@noauth` for public endpoints that should bypass any global or group-level auth:

```
import { Router } from "tina4-nodejs";

/**
 * @noauth
 */
Router.get("/api/public/health", async (req, res) => {
```

```
    return res.json({ status: "ok" });
});

/**
 * @noauth
 */
Router.post("/api/login", async (req, res) => {
    // Login logic
});

/**
 * @noauth
 */
Router.post("/api/register", async (req, res) => {
    // Registration logic
});
```

@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:

```
import { Router } from "tina4-nodejs";

/**
 * @secured
 */
Router.get("/api/me", async (req, res) => {
    // This GET route requires authentication
    return res.json((req as any).user);
});
```

Role-Based Authorization

Combine auth middleware with role checks:

```
import { Auth } from "tina4-nodejs";

function requireRole(role: string) {
    return function (req, res, next) {
        const authHeader = req.headers["authorization"] ?? "";
        if (!authHeader || !authHeader.startsWith("Bearer ")) {
            res({ error: "Authorization required" }, 401);
            return;
        }

        const token = authHeader.substring(7);
        const secret = process.env.TINA4_SECRET || "tina4-default-secret";
        const payload = Auth.validToken(token, secret);
        if (payload === null) {
            res({ error: "Invalid or expired token" }, 401);
            return;
        }

        if ((payload.role ?? "") !== role) {
            res({ error: `Forbidden. Required role: ${role}` }, 403);
            return;
        }

        (req as any).user = payload;
```

```
        next();
    };
}
```

Use it as middleware:

```
import { Router } from "tina4-nodejs";

Router.delete("/api/users/:id", async (req, res) => {
    // Only admins can delete users
    return res.json({ deleted: true });
}, [requireRole("admin")]);
```

---

8. CSRF Protection

Traditional form-based applications (not SPAs) need CSRF protection. Cross-Site Request Forgery attacks trick a user's browser into submitting a form to your server. The browser sends cookies automatically -- the attacker rides on the user's session.

CSRF tokens stop this. Each form gets a unique token. When the form submits, the server checks the token. If it does not match, the request is rejected.

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:

```
import { Router, Auth } from "tina4-nodejs";

Router.post("/profile/update", async (req, res) => {
    // Validate CSRF token
    if (!Auth.validateFormToken(req.body._token ?? "")) {
        return res.status(403).json({ error: "Invalid form token. Please refresh and try again."
    }

    // Process the form...
    return res.redirect("/profile");
});
```

The CSRF token is tied to the session and expires after a single use. A malicious site cannot forge a form submission because it cannot guess the token.

CSRF Middleware

For automatic validation on all POST/PUT/DELETE requests, use the CSRF middleware:

```
import { csrfMiddleware } from "tina4-nodejs";

// Apply globally to all form-based routes
Router.post("/profile/update", async (req, res) => {
    // CSRF validation happens automatically
    return res.redirect("/profile");
}, [csrfMiddleware]);
```

When to Use CSRF Tokens

Use CSRF tokens for:

- HTML forms submitted by browsers
- Any POST/PUT/DELETE from server-rendered pages

You do not need CSRF tokens for:

- API endpoints that use JWT (the Bearer token already proves the request is intentional)
- Single-page applications that use `fetch()` with custom headers

---

9. Sessions

Tina4 supports server-side sessions for storing per-user state between requests. JWTs handle API authentication. Sessions handle stateful web pages. Both work side by side.

Session Configuration

Set the session backend in `.env`:

```
# File-based sessions (default)
TINA4_SESSION_BACKEND=file

# Redis
TINA4_SESSION_BACKEND=redis
TINA4_SESSION_REDIS_HOST=localhost
TINA4_SESSION_REDIS_PORT=6379

# MongoDB
TINA4_SESSION_BACKEND=mongodb
TINA4_SESSION_REDIS_HOST=localhost
TINA4_SESSION_REDIS_PORT=27017

# Valkey
TINA4_SESSION_BACKEND=valkey
TINA4_SESSION_REDIS_HOST=localhost
TINA4_SESSION_REDIS_PORT=6379
```

File-based sessions work out of the box. No extra dependencies. For production deployments with multiple servers, use Redis or Valkey so sessions are shared across instances.

Using Sessions

Access session data via `req.session`:

```
import { Router } from "tina4-nodejs";

Router.post("/login-form", async (req, res) => {
    // After validating credentials...
    req.session.userId = 42;
    req.session.userName = "Alice";
    req.session.loggedIn = true;

    return res.redirect("/dashboard");
});

Router.get("/dashboard", async (req, res) => {
    if (!req.session.loggedIn) {
        return res.redirect("/login");
    }

    return res.html("dashboard.html", {
        userName: req.session.userName
    });
});

Router.post("/logout", async (req, res) => {
    // Clear all session data
    req.session = {};

    return res.redirect("/login");
});
```

Sessions complement JWT tokens. Use JWT for stateless API authentication. Use sessions for traditional web applications with server-rendered pages.

Session Options

```
TINA4_SESSION_TTL=3600       # Session lifetime in seconds (default: 3600)
TINA4_SESSION_NAME=tina4_session  # Cookie name for the session ID
```

---

10. Token Refresh

Tokens expire. When they do, `Auth.validToken()` returns `null` and the middleware rejects the request with a `401`. The user must log in again -- unless you implement token refresh.

A refresh endpoint issues a new token before the current one expires:

```
import { Router, Auth } from "tina4-nodejs";

Router.post("/api/auth/refresh", async (req, res) => {
    // req.user is populated by auth middleware
    const secret = process.env.TINA4_SECRET || "tina4-default-secret";
    const newToken = Auth.getToken({
```

```
        user_id: req.user.user_id,
        email: req.user.email,
        role: req.user.role,
    }, secret);

    return res.json({ token: newToken });
}, [authMiddleware]);
```

The frontend can call this endpoint periodically to keep the session alive without requiring re-login. A common pattern: a short-lived access token (15 minutes) paired with a long-lived refresh token (7 days). The refresh token can mint new access tokens without re-entering credentials.

---

11. 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 `req.user`.

Test with:

```
# Register
curl -X POST http://localhost:7148/api/register \
  -H "Content-Type: application/json" \
  -d '{"name": "Alice", "email": "alice@example.com", "password": "securePass123"}'

# Login
curl -X POST http://localhost:7148/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:7148/api/profile \
  -H "Authorization: Bearer YOUR_TOKEN_HERE"

# Update profile
curl -X PUT http://localhost:7148/api/profile \
  -H "Authorization: Bearer YOUR_TOKEN_HERE" \
  -H "Content-Type: application/json" \
  -d '{"name": "Alice Smith"}'

# Change password
curl -X PUT http://localhost:7148/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:7148/api/profile
```

---

12. Solution

Migration

Create `migrations/20260322160000_create_users_table.sql`:

```
-- UP
CREATE TABLE users (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    name TEXT NOT NULL,
    email TEXT NOT NULL UNIQUE,
    password_hash TEXT NOT NULL,
    role TEXT NOT NULL DEFAULT 'user',
    created_at TEXT DEFAULT CURRENT_TIMESTAMP
);

-- DOWN
DROP TABLE IF EXISTS users;
```

```
tina4 migrate
```

Routes

Create `src/routes/auth.ts`: _____

```
import { Router, Auth } from "tina4-nodejs";
import { Database } from "tina4-nodejs/orm";


function authMiddleware(req, res, next) {
    const authHeader = req.headers["authorization"] ?? "";

    if (!authHeader || !authHeader.startsWith("Bearer ")) {
        res({ error: "Authorization required. Send: Authorization: Bearer <token>" }, 401);
        return;
    }

    const token = authHeader.substring(7);
    const secret = process.env.TINA4_SECRET || "tina4-default-secret";

    const payload = Auth.validToken(token, secret);
    if (payload === null) {
        res({ error: "Invalid or expired token. Please login again." }, 401);
        return;
    }

    req.user = payload;

    next();
}


/**
 * @noauth
 */
Router.post("/api/register", async (req, res) => {
    const body = req.body;
    const errors: string[] = [];

    if (!body.name) errors.push("Name is required");
    if (!body.email) errors.push("Email is required");
    if (!body.password) errors.push("Password is required");
    else if (body.password.length < 8) errors.push("Password must be at least 8 characters");

    if (errors.length > 0) {
        return res.status(400).json({ errors });
    }

    const db = Database.getConnection();

    const existing = await db.fetchOne("SELECT id FROM users WHERE email = :email", { email: bod
    if (existing !== null) {
        return res.status(409).json({ error: "Email already registered" });
    }

    const hash = await Auth.hashPassword(body.password);

    await db.execute(
        "INSERT INTO users (name, email, password_hash) VALUES (:name, :email, :hash)",
        { name: body.name, email: body.email, hash }
    );

    const user = await db.fetchOne("SELECT id, name, email, role, created_at FROM users WHERE id
```

```
    return res.status(201).json({ message: "Registration successful", user });
});


/**
 * @noauth
 */
Router.post("/api/login", async (req, res) => {
    const body = req.body;

    if (!body.email || !body.password) {
        return res.status(400).json({ error: "Email and password are required" });
    }

    const db = Database.getConnection();

    const user = await db.fetchOne(
        "SELECT id, name, email, password_hash, role FROM users WHERE email = :email",
        { email: body.email }
    );

    if (user === null || !(await Auth.checkPassword(body.password, user.password_hash))) {
        return res.status(401).json({ error: "Invalid email or password" });
    }

    const secret = process.env.TINA4_SECRET || "tina4-default-secret";
    const token = Auth.getToken({
        user_id: user.id,
        email: user.email,
        name: user.name,
        role: user.role
    }, secret);

    return res.json({
        message: "Login successful",
        token,
        user: { id: user.id, name: user.name, email: user.email, role: user.role }
    });
});


Router.get("/api/profile", async (req, res) => {
    const db = Database.getConnection();

    const user = await db.fetchOne(
        "SELECT id, name, email, role, created_at FROM users WHERE id = :id",
        { id: req.user.user_id }
    );

    if (user === null) {
        return res.status(404).json({ error: "User not found" });
    }

    return res.json(user);
}, [authMiddleware]);


Router.put("/api/profile", async (req, res) => {
    const db = Database.getConnection();
```

```
    const body = req.body;
    const userId = req.user.user_id;

    // Check if the new email is already taken by another account
    if (body.email) {
        const existing = await db.fetchOne(
            "SELECT id FROM users WHERE email = :email AND id != :id",
            { email: body.email, id: userId }
        );
        if (existing !== null) {
            return res.status(409).json({ error: "Email already in use by another account" });
        }
    }

    const current = await db.fetchOne("SELECT * FROM users WHERE id = :id", { id: userId });

    await db.execute(
        "UPDATE users SET name = :name, email = :email WHERE id = :id",
        { name: body.name ?? current.name, email: body.email ?? current.email, id: userId }
    );

    const updated = await db.fetchOne(
        "SELECT id, name, email, role, created_at FROM users WHERE id = :id",
        { id: userId }
    );

    return res.json({ message: "Profile updated", user: updated });
}, [authMiddleware]);


Router.put("/api/profile/password", async (req, res) => {
    const db = Database.getConnection();
    const body = req.body;
    const userId = req.user.user_id;

    if (!body.current_password || !body.new_password) {
        return res.status(400).json({ error: "Current password and new password are required" })
    }

    if (body.new_password.length < 8) {
        return res.status(400).json({ error: "New password must be at least 8 characters" });
    }

    const user = await db.fetchOne("SELECT password_hash FROM users WHERE id = :id", { id: userI

    if (!(await Auth.checkPassword(body.current_password, user.password_hash))) {
        return res.status(401).json({ error: "Current password is incorrect" });
    }

    const newHash = await Auth.hashPassword(body.new_password);
    await db.execute("UPDATE users SET password_hash = :hash WHERE id = :id", { hash: newHash, i

    return res.json({ message: "Password changed successfully" });
}, [authMiddleware]);
```

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`)

---

13. Gotchas

1. Token Expiry Confusion

Problem: Tokens that worked yesterday now return 401.

Cause: The default token lifetime is 60 minutes. After that, the token is invalid even if the signature is correct.

Fix: Issue a new token at login. If your application needs long-lived sessions, use refresh tokens: a short-lived access token (15 minutes) paired with a long-lived refresh token (7 days) that can be used to get a new access token without re-entering credentials.

2. Secret Key Management

Problem: Tokens generated on one server are invalid on another, or tokens stop working after a deployment.

Cause: Each server generated its own random `secrets/jwt.key` file. Or the key file was deleted or 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` in a JSDoc comment above your login and register routes. Without it, users cannot authenticate because authentication is required to authenticate -- a catch-22.

6. Password Hash Column Too Short

Problem: Registration fails with a database error about the password hash being too long.

Cause: PBKDF2 hashes can be long. If your `password_hash` column is defined as `VARCHAR(50)`, it gets truncated.

Fix: Use `TEXT` for the password hash column, or at minimum `VARCHAR(255)`. Never constrain the hash length.

7. Token in URL Query Parameters

Problem: Tokens in URLs like `/api/profile?token=eyJ...` leak through browser history, server logs, and the Referer header.

Cause: Query parameters are visible in many places where headers are not.

Fix: Always send tokens in the `Authorization` header, never in the URL. The only exception is WebSocket connections, where the initial HTTP upgrade request cannot carry custom headers -- use a short-lived token for that case.

Sessions & Cookies

1. State in a Stateless World

JWT tokens handle APIs. Traditional web applications need more. A shopping cart that persists across pages. A flash message that appears once after a redirect. A "remember me" checkbox on the login form. These features run on sessions and cookies -- server-side state tied to a browser.

Picture an e-commerce site. A customer adds three items to their cart. Navigates to a product page. Comes back to the cart. Without sessions, the cart is empty every time. Sessions give the server memory. Each browser gets its own state, preserved across requests.

Sessions are auto-started. Every route handler receives `req.session` ready to use. No manual setup. No configuration required for the default file backend.

---

2. Session Configuration

The default backend is file-based sessions. They work out of the box with no additional dependencies.

To change the backend, set `TINA4_SESSION_BACKEND` in `.env`:

```
TINA4_SESSION_BACKEND=file
```

Available Backends

Backend	Value	Package Required	Best For
File	`file`	None	Development, single server
Redis	`redis`	`ioredis`	Production, multi-server
MongoDB	`mongodb`	`mongodb`	Production, document stores
Valkey	`valkey`	`iovalkey`	Production, Redis alternative
Database	`database`	None	Production, using existing DB

Redis Configuration

```
TINA4_SESSION_BACKEND=redis
TINA4_SESSION_REDIS_HOST=localhost
TINA4_SESSION_REDIS_PORT=6379
TINA4_SESSION_REDIS_PASSWORD=
```

Install the Redis driver:

```
npm install ioredis
```

MongoDB Configuration

```
TINA4_SESSION_BACKEND=mongodb
TINA4_SESSION_REDIS_HOST=localhost
TINA4_SESSION_REDIS_PORT=27017
TINA4_SESSION_MONGO_DB=tina4_sessions
```

Valkey Configuration

```
TINA4_SESSION_BACKEND=valkey
TINA4_SESSION_REDIS_HOST=localhost
TINA4_SESSION_REDIS_PORT=6379
```

Database Sessions

```
TINA4_SESSION_BACKEND=database
```

Stores sessions in the `tina4_session` table using your existing database connection (`TINA4_DATABASE_URL`). The table is auto-created on first use. Works with all 5 database engines (SQLite, PostgreSQL, MySQL, MSSQL, Firebird).

Session Lifetime

```
TINA4_SESSION_TTL=3600  # 1 hour in seconds (default)
```

Session Cookie

The session cookie is named `tina4_session`. It is `HttpOnly` and `SameSite=Lax` by default. Tina4 manages it -- you never need to set or read it yourself.

---

3. The Session API

Access session data through `req.session`. It is available in every route handler with zero configuration.

Full API Reference

Method	Description
`req.session.set(key, value)`	Store a value
`req.session.get(key, default)`	Retrieve a value (with optional default)
`req.session.delete(key)`	Remove a key
`req.session.has(key)`	Check if a key exists
`req.session.clear()`	Remove all session data
`req.session.destroy()`	Destroy the session entirely
`req.session.save()`	Persist session data (auto-called after response)
`req.session.regenerate()`	Generate a new session ID, preserve data
`req.session.flash(key, value)`	Set flash data (one-time read)
`req.session.flash(key)`	Read and remove flash data (dual-mode)
`req.session.getFlash(key)`	Explicit getter alias for flash(key)
`req.session.getSessionId()`	Get current session ID
`req.session.cookieHeader()`	Get Set-Cookie header value
`req.session.all()`	Get all session data as an object

---

4. Reading and Writing Session Data

Writing to the Session

```
import { Router } from "tina4-nodejs";

Router.post("/login-form", async (req, res) => {
    // After validating credentials...
    req.session.set("user_id", 42);
    req.session.set("user_name", "Alice");
    req.session.set("role", "admin");
    req.session.set("logged_in", true);

    return res.redirect("/dashboard");
});
```

Reading from the Session

```
import { Router } from "tina4-nodejs";

Router.get("/dashboard", async (req, res) => {
    if (!req.session.get("logged_in")) {
        return res.redirect("/login");
    }

    return res.html("dashboard.html", {
        user_name: req.session.get("user_name"),
        role: req.session.get("role")
    });
});
```

Deleting Session Data

```
// Delete a single key
req.session.delete("temp_data");

// Clear all session data (logout)
Router.post("/logout", async (req, res) => {
    req.session.clear();
    return res.redirect("/login");
});
```

Checking if a Key Exists

```
if (req.session.has("user_id")) {
    const userId = req.session.get("user_id");
} else {
    const userId = null;
}

// Or use .get() with a default
const userId = req.session.get("user_id", null);
```

Getting All Session Data

```
const allData = req.session.all();
```

Preferences Example

```
import { Router } from "tina4-nodejs";

Router.post("/api/preferences", async (req, res) => {
    const body = req.body;

    req.session.set("language", body.language ?? "en");
    req.session.set("theme", body.theme ?? "light");
    req.session.set("items_per_page", parseInt(body.items_per_page ?? "20", 10));

    return res.json({
        message: "Preferences saved",
        preferences: {
            language: req.session.get("language"),
            theme: req.session.get("theme"),
            items_per_page: req.session.get("items_per_page")
        }
    });
});

Router.get("/api/preferences", async (req, res) => {
    return res.json({
        language: req.session.get("language", "en"),
        theme: req.session.get("theme", "light"),
        items_per_page: req.session.get("items_per_page", 20)
    });
});
```

Visit Counter Example

```
import { Router } from "tina4-nodejs";

Router.get("/visit-counter", async (req, res) => {
    const count = (req.session.get("visit_count", 0)) + 1;
    req.session.set("visit_count", count);

    return res.json({
        visit_count: count,
        message: `You have visited this page ${count} time${count === 1 ? "" : "s"}`
    });
});
```

```
curl http://localhost:7148/visit-counter -c cookies.txt -b cookies.txt
```

```
{"visit_count":1,"message":"You have visited this page 1 time"}
```

---

5. Flash Messages

Flash messages are session values that exist for exactly one read. Set them before a redirect. Read them on the next request. They vanish after being read.

Setting a Flash Message

```
import { Router } from "tina4-nodejs";

Router.post("/api/contact", async (req, res) => {
    // Process the form...

    req.session.flash("message", "Your message has been sent!");
    req.session.flash("message_type", "success");

    return res.redirect("/contact");
});
```

Reading a Flash Message

Flash is dual-mode: `flash(key, value)` sets, `flash(key)` gets and removes. `getFlash()` is an explicit alias.

```
Router.get("/contact", async (req, res) => {
    const flashMessage = req.session.flash("message");       // get + auto-remove
    const flashType = req.session.flash("message_type") ?? "info";

    return res.html("contact.html", {
        flash_message: flashMessage,
        flash_type: flashType
    });
});
```

The `getFlash()` method reads the value and removes it in one step. The next time the user visits `/contact`, the flash message will be gone.

Flash Messages in Templates

```
{% if flash_message %}
    <div class="alert alert-{{ flash_type }}">
        {{ flash_message }}
    </div>
{% endif %}
```

---

6. Cookies

Cookies are small data fragments stored in the browser. Unlike sessions, cookies travel with every request and are visible to JavaScript (unless `httpOnly` is set).

Setting a Cookie

```
import { Router } from "tina4-nodejs";

Router.post("/preferences", async (req, res) => {
    const theme = req.body.theme ?? "light";
    const language = req.body.language ?? "en";

    return res
        .cookie("theme", theme, {
            maxAge: 365 * 24 * 60 * 60 * 1000,  // 1 year in milliseconds
            path: "/",
            sameSite: "Lax"
        })
        .cookie("language", language, {
            maxAge: 365 * 24 * 60 * 60 * 1000,
            path: "/",
            sameSite: "Lax"
        })
        .json({ message: "Preferences saved" });
});
```

Reading a Cookie

```
import { Router } from "tina4-nodejs";

Router.get("/", async (req, res) => {
    const theme = req.cookies.theme ?? "light";
    const language = req.cookies.language ?? "en";

    return res.html("home.html", {
        theme,
        language
    });
});
```

Deleting a Cookie

Set `maxAge` to 0:

```
Router.post("/clear-preferences", async (req, res) => {
    return res
        .cookie("theme", "", { maxAge: 0, path: "/" })
        .cookie("language", "", { maxAge: 0, path: "/" })
        .json({ message: "Preferences cleared" });
});
```

Cookie Options

Option	Type	Description
`maxAge`	number	Lifetime in milliseconds. 0 = delete. Omit = session cookie (deleted when browser closes).
`expires`	Date	Expiry date. `maxAge` is preferred -- it is simpler and more predictable.
`path`	string	URL path scope. `/` means the whole site. `/admin` means only under `/admin`.
`domain`	string	Domain scope. `.example.com` includes subdomains.
`secure`	boolean	Only send over HTTPS. Always `true` in production.
`httpOnly`	boolean	Not accessible via JavaScript. Use for session cookies.
`sameSite`	string	`"Strict"`, `"Lax"`, or `"None"`. Controls cross-site sending.

When to Use Cookies vs Sessions

Use Case	Use Cookies	Use Sessions
User preferences (theme, language)	Yes	No
Shopping cart	No	Yes
Authentication state	No (use JWT in header)	Yes (for form-based auth)
Flash messages	No	Yes
Tracking consent	Yes	No
Sensitive data (user ID, role)	No	Yes

The rule: sensitive data or data the client should not see goes in sessions. Non-sensitive data that should persist beyond session expiry goes in cookies.

---

7. Remember Me

The "remember me" pattern extends the session lifetime when the user checks a box on the login form:

```
import { Router } from "tina4-nodejs";

Router.post("/login", async (req, res) => {
    const { email, password, remember } = req.body;

    // Validate credentials...
    const user = await authenticateUser(email, password);
    if (!user) {
        req.session.flash("message", "Invalid email or password");
        req.session.flash("message_type", "error");
        return res.redirect("/login");
    }

    // Regenerate session ID (prevents session fixation)
    req.session.regenerate();

    // Set session data
    req.session.set("user_id", user.id);
    req.session.set("user_name", user.name);
    req.session.set("logged_in", true);

    // Handle "remember me"
    if (remember) {
        // Generate a long-lived token
        const Auth = require("tina4-nodejs").Auth;
        const rememberToken = Auth.getToken({ user_id: user.id });

        // Store it in a cookie that lasts 30 days
        return res
            .cookie("remember_token", rememberToken, {
                maxAge: 30 * 24 * 60 * 60 * 1000,
                httpOnly: true,
                secure: true,
                path: "/",
                sameSite: "Lax"
            })
            .redirect("/dashboard");
    }

    return res.redirect("/dashboard");
});
```

Then, on every page load, check for the remember token if the session has expired:

```
async function rememberMeMiddleware(req, res, nextHandler) {
    if (!req.session.get("logged_in")) {
        const rememberToken = req.cookies.remember_token;

        if (rememberToken && Auth.validToken(rememberToken)) {
            const payload = Auth.getPayload(rememberToken);
            const db = new Database();
```

```
            const user = await db.fetchOne(
                "SELECT id, name FROM users WHERE id = ?",
                [payload.user_id]
            );

            if (user) {
                req.session.set("user_id", user.id);
                req.session.set("user_name", user.name);
                req.session.set("logged_in", true);
            }
        }
    }

    return await nextHandler(req, res);
}
```

The form:

```
<form method="POST" action="/login">
    {{ form_token() }}
    <div class="form-group">
        <label for="email">Email</label>
        <input type="email" id="email" name="email" class="form-control" required>
    </div>
    <div class="form-group">
        <label for="password">Password</label>
        <input type="password" id="password" name="password" class="form-control" required>
    </div>
    <div class="form-check">
        <input type="checkbox" id="remember" name="remember" value="1" class="form-check-input">
        <label for="remember" class="form-check-label">Remember me for 30 days</label>
    </div>
    <button type="submit" class="btn btn-primary">Login</button>
</form>
```

Without the checkbox, the session uses the default TTL (1 hour). With the checkbox, the remember token cookie persists for 30 days. The server controls the duration -- the client cannot extend it.

---

8. Session Security

Regenerate Session ID After Login

To prevent session fixation attacks, regenerate the session ID after login:

```
Router.post("/login-form", async (req, res) => {
    // After validating credentials...

    // Regenerate session ID (prevents session fixation)
    req.session.regenerate();

    req.session.set("user_id", user.id);
    req.session.set("logged_in", true);

    return res.redirect("/dashboard");
});
```

Destroy a Session

To destroy a session entirely (not just clear its data):

```
Router.post("/logout", async (req, res) => {
    req.session.destroy();
    return res.redirect("/login");
});
```

Secure Cookie Settings

The session cookie (`tina4_session`) is `HttpOnly` and `SameSite=Lax` by default. For production, ensure HTTPS:

```
TINA4_SESSION_SECURE=true     # Only send over HTTPS
TINA4_SESSION_HTTPONLY=true   # Not accessible via JavaScript (default)
TINA4_SESSION_SAMESITE=Lax   # Prevent CSRF via cross-site requests (default)
```

`TINA4_SESSION_SAMESITE` controls cross-site cookie behavior:

Value	Behavior
`Strict`	Never sent with cross-site requests. Safest. Breaks some flows (clicking links from email).
`Lax`	Sent with top-level navigations (clicking links) but not cross-site API calls. Good default.
`None`	Always sent. Requires `TINA4_SESSION_SECURE=true`. Only for cross-site cookie access.

Session Configuration Options

Variable	Default	Description
`TINA4_SESSION_BACKEND`	`file`	Storage backend: `file`, `redis`, `mongodb`, `valkey`, `database`
`TINA4_SESSION_TTL`	`3600`	Session lifetime in seconds
`TINA4_SESSION_REDIS_HOST`	`localhost`	Host for Redis/MongoDB/Valkey
`TINA4_SESSION_REDIS_PORT`	`6379`	Port for Redis/MongoDB/Valkey
`TINA4_SESSION_REDIS_PASSWORD`	(none)	Password for Redis/Valkey
`TINA4_SESSION_SECURE`	`false`	Cookie sent only over HTTPS
`TINA4_SESSION_HTTPONLY`	`true`	Cookie inaccessible to JavaScript
`TINA4_SESSION_SAMESITE`	`Lax`	Cross-site cookie policy

---

9. Exercise: Build a Shopping Cart

Build a shopping cart using sessions.

Requirements

- Create these endpoints:

Method	Path	Description
`GET`	`/cart`	View cart contents (rendered HTML page)
`POST`	`/cart/add`	Add an item to the cart
`POST`	`/cart/update`	Update item quantity
`POST`	`/cart/remove`	Remove an item from the cart
`POST`	`/cart/clear`	Clear the entire cart
`GET`	`/cart/api`	Get cart as JSON (for AJAX)

- The cart is stored in the session via `req.session.set("cart", [...])`
- Each cart item has: `product_id`, `name`, `price`, `quantity`

- Adding an existing product increments its quantity
- The cart page shows items, quantities, line totals, and a grand total
- Use flash messages for feedback ("Item added", "Cart cleared", etc.)

Use this product data for validation:

```
const PRODUCTS: Record<number, { name: string; price: number }> = {
    1: { name: "Wireless Keyboard", price: 79.99 },
    2: { name: "USB-C Hub", price: 49.99 },
    3: { name: "Monitor Stand", price: 129.99 },
    4: { name: "Mechanical Mouse", price: 59.99 },
    5: { name: "Desk Lamp", price: 39.99 }
};
```

Test with:

```
# Add items
curl -X POST http://localhost:7148/cart/add \
  -H "Content-Type: application/json" \
  -d '{"product_id": 1, "quantity": 2}' \
  -c cookies.txt -b cookies.txt

curl -X POST http://localhost:7148/cart/add \
  -H "Content-Type: application/json" \
  -d '{"product_id": 3, "quantity": 1}' \
  -c cookies.txt -b cookies.txt

# View cart
curl http://localhost:7148/cart/api -b cookies.txt

# Update quantity
curl -X POST http://localhost:7148/cart/update \
  -H "Content-Type: application/json" \
  -d '{"product_id": 1, "quantity": 3}' \
  -c cookies.txt -b cookies.txt

# Remove item
curl -X POST http://localhost:7148/cart/remove \
  -H "Content-Type: application/json" \
  -d '{"product_id": 3}' \
  -c cookies.txt -b cookies.txt

# Clear cart
curl -X POST http://localhost:7148/cart/clear -c cookies.txt -b cookies.txt
```

---

10. Solution

Create `src/routes/cart.ts`:

```
import { Router } from "tina4-nodejs";

const PRODUCTS: Record<number, { name: string; price: number }> = {
    1: { name: "Wireless Keyboard", price: 79.99 },
    2: { name: "USB-C Hub", price: 49.99 },
    3: { name: "Monitor Stand", price: 129.99 },
    4: { name: "Mechanical Mouse", price: 59.99 },
```

```
    5: { name: "Desk Lamp", price: 39.99 }
};


function getCart(session: any): any[] {
    return session.get("cart", []);
}


function saveCart(session: any, cart: any[]): void {
    session.set("cart", cart);
}


function calculateTotals(cart: any[]): { cart: any[]; grandTotal: number } {
    for (const item of cart) {
        item.line_total = Math.round(item.price * item.quantity * 100) / 100;
    }
    const grandTotal = Math.round(
        cart.reduce((sum, item) => sum + item.line_total, 0) * 100
    ) / 100;
    return { cart, grandTotal };
}


Router.get("/cart", async (req, res) => {
    const cart = getCart(req.session);
    const { grandTotal } = calculateTotals(cart);
    const flashMessage = req.session.getFlash("message");
    const flashType = req.session.getFlash("message_type") ?? "info";

    return res.html("cart.html", {
        cart,
        grand_total: grandTotal,
        item_count: cart.reduce((sum, item) => sum + item.quantity, 0),
        flash_message: flashMessage,
        flash_type: flashType
    });
});


Router.get("/cart/api", async (req, res) => {
    const cart = getCart(req.session);
    const { grandTotal } = calculateTotals(cart);

    return res.json({
        cart,
        grand_total: grandTotal,
        item_count: cart.reduce((sum, item) => sum + item.quantity, 0)
    });
});


Router.post("/cart/add", async (req, res) => {
    const body = req.body;
    const productId = parseInt(body.product_id, 10);
    const quantity = parseInt(body.quantity ?? "1", 10);

    if (!PRODUCTS[productId]) {
```

```
        return res.status(404).json({ error: "Product not found" });
    }

    if (quantity < 1) {
        return res.status(400).json({ error: "Quantity must be at least 1" });
    }

    const product = PRODUCTS[productId];
    const cart = getCart(req.session);

    // Check if product already in cart
    const existing = cart.find(item => item.product_id === productId);
    if (existing) {
        existing.quantity += quantity;
        saveCart(req.session, cart);
        req.session.flash("message", `Updated ${product.name} quantity`);
        req.session.flash("message_type", "success");
        return res.json({ message: `Updated ${product.name} quantity`, cart });
    }

    // Add new item
    cart.push({
        product_id: productId,
        name: product.name,
        price: product.price,
        quantity
    });
    saveCart(req.session, cart);

    req.session.flash("message", `Added ${product.name} to cart`);
    req.session.flash("message_type", "success");

    return res.status(201).json({ message: `Added ${product.name}`, cart });
});


Router.post("/cart/update", async (req, res) => {
    const body = req.body;
    const productId = parseInt(body.product_id, 10);
    const quantity = parseInt(body.quantity ?? "1", 10);

    const cart = getCart(req.session);
    const index = cart.findIndex(item => item.product_id === productId);

    if (index === -1) {
        return res.status(404).json({ error: "Product not in cart" });
    }

    if (quantity < 1) {
        const removed = cart[index];
        cart.splice(index, 1);
        req.session.flash("message", `Removed ${removed.name} from cart`);
    } else {
        cart[index].quantity = quantity;
        req.session.flash("message", `Updated ${cart[index].name} quantity to ${quantity}`);
    }

    req.session.flash("message_type", "success");
    saveCart(req.session, cart);
```

```
    return res.json({ message: "Cart updated", cart });
});


Router.post("/cart/remove", async (req, res) => {
    const productId = parseInt(req.body.product_id, 10);
    const cart = getCart(req.session);
    const index = cart.findIndex(item => item.product_id === productId);

    if (index === -1) {
        return res.status(404).json({ error: "Product not in cart" });
    }

    const removed = cart[index];
    cart.splice(index, 1);
    saveCart(req.session, cart);

    req.session.flash("message", `Removed ${removed.name} from cart`);
    req.session.flash("message_type", "success");

    return res.json({ message: `Removed ${removed.name}`, cart });
});


Router.post("/cart/clear", async (req, res) => {
    saveCart(req.session, []);
    req.session.flash("message", "Cart cleared");
    req.session.flash("message_type", "info");
    return res.json({ message: "Cart cleared", cart: [] });
});
```

Expected output for cart API after adding items:

```
{
  "cart": [
    {"product_id": 1, "name": "Wireless Keyboard", "price": 79.99, "quantity": 2, "line_total":
    {"product_id": 3, "name": "Monitor Stand", "price": 129.99, "quantity": 1, "line_total": 129
  ],
  "grand_total": 289.97,
  "item_count": 3
}
```

---

11. Gotchas

1. Session lost between requests

Problem: Data you stored in the session is gone on the next request.

Cause: The session cookie is not being sent back by the client. This happens when using curl without `-c cookies.txt -b cookies.txt`, or when the browser blocks cookies.

Fix: For curl testing, use `-c cookies.txt -b cookies.txt` to save and send cookies. In the browser, make sure cookies are not blocked for your site.

2. Session data is not JSON-serializable

Problem: Storing a JavaScript object with circular references or class instances in the session causes an error.

Cause: Sessions are serialized to JSON. Objects with circular references, `Map`, `Set`, `Date`, or custom class instances cannot be serialized directly.

Fix: Convert objects to plain values before storing. For dates: `req.session.set("login_time", new Date().toISOString())`. For maps: convert to a plain object with `Object.fromEntries(myMap)`.

3. Cookie not sent on cross-origin requests

Problem: Your frontend at `http://localhost:3000` calls your API at `http://localhost:7148`, but cookies are not included.

Cause: Browsers do not send cookies cross-origin by default. You need both CORS configuration and explicit `credentials: "include"` in fetch.

Fix: Set `CORS_CREDENTIALS=true` in `.env` and use `fetch(url, { credentials: "include" })` in your frontend JavaScript. Also set `CORS_ORIGINS` to the specific frontend origin (not `*` -- wildcard does not work with credentials).

4. File-based sessions on multi-server deployments

Problem: A user is logged in on server A but logged out on server B.

Cause: File-based sessions are stored on the local filesystem. Each server has its own session files.

Fix: Switch to Redis, Valkey, MongoDB, or database-backed sessions so all servers share the same session store.

5. Session cookie overwritten by another app

Problem: Your session keeps getting reset even though you set it correctly.

Cause: Another application on the same domain uses the same session cookie name.

Fix: The session cookie name (`tina4_session`) is managed by Tina4. If you need to change it, set `TINA4_SESSION_NAME=myapp_session` in `.env`.

6. Secure cookies on localhost

Problem: Setting `secure: true` on a cookie means it never gets sent during development.

Cause: Secure cookies are only sent over HTTPS. `http://localhost` is not HTTPS.

Fix: Do not set `secure: true` during development. Use environment-based configuration: `secure: process.env.TINA4_DEBUG !== "true"`.

7. Flash messages shown twice

Problem: A flash message appears on the page, but when you refresh, it shows again.

Cause: You read the flash message but did not remove it. Using `req.session.get()` reads without deleting.

Fix: Use `req.session.getFlash("message")` to read flash data. It reads and deletes in one step. Do not use `req.session.get()` for flash messages.

8. Session data disappears after server restart

Problem: All sessions vanish when you restart the Node.js process.

Cause: File-based sessions may be stored in a temporary directory that gets cleared on restart, or in-memory state was lost.

Fix: Use Redis or database sessions for production. For file sessions, set `TINA4_SESSION_PATH` to a persistent directory.

9. Large session data causes slow requests

Problem: Pages load slower than expected. Session reads and writes take visible time.

Cause: You stored large objects in the session -- full user profiles, query results, or file contents.

Fix: Keep session data small. Store IDs, not entire objects. Look up the full data from the database when you need it.

Middleware

1. The Pipeline Pattern

Every HTTP request passes through a series of gates before reaching your route handler. Rate limiter. Body parser. Auth check. Logger. These gates are middleware -- code that wraps your route handler and runs before, after, or both.

Picture a public API. Every request hits a rate limit check. Some endpoints require an API key. All responses need CORS headers. Errors need logging. Without middleware, that logic lives in every handler. Duplicated. Scattered. Fragile. With middleware, you write it once and attach it where it belongs.

Tina4 Node.js ships with built-in middleware (CORS, rate limiting, request logging, security headers) and lets you write your own. This chapter covers both.

---

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 Node.js supports two styles of middleware:

Function-based middleware receives `req`, `res`, and `next`. Call `next()` to continue. Skip it to short-circuit.

```
function passthrough(req, res, next) {
    next();
}
```

```
async function blockEverything(req, res, next) {
    return res.status(503).json({ error: "Service unavailable" });
}
```

Class-based middleware uses naming conventions. Static methods whose names start with `before` run before the handler (via `MiddlewareRunner.runBefore`). Methods starting with `after` run after it (via `MiddlewareRunner.runAfter`). Each method receives `(req, res)` and returns `[req, res]`.

```
class MyMiddleware {
    static beforeCheck(req, res) {
        // Runs before the route handler
        return [req, res];
    }

    static afterCleanup(req, res) {
        // Runs after the route handler
        return [req, res];
    }
}
```

If a `before*` method sets the response status to >= 400, the handler is skipped. This is short-circuiting.

Register class-based middleware globally with `Router.use()`:

```
import { Router, CorsMiddleware, RateLimiterMiddleware, RequestLogger } from "tina4-nodejs";

Router.use(CorsMiddleware);
Router.use(RateLimiterMiddleware);
Router.use(RequestLogger);
```

---

3. Built-in CorsMiddleware

CORS (Cross-Origin Resource Sharing) controls which domains can call your API from a browser. When React at `http://localhost:3000` calls your Tina4 API at `http://localhost:7148`, the browser sends a preflight `OPTIONS` request first. Wrong headers and the browser blocks everything.

Tina4 provides both a function-based `cors()` middleware and a class-based `CorsMiddleware`. Configure via `.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,X-Request-ID
TINA4_CORS_MAX_AGE=86400
```

With these settings, only `localhost:3000` and `myapp.com` can make cross-origin requests to your API. The browser handles preflight `OPTIONS` requests on its own.

For development, allow all origins:

```
TINA4_CORS_ORIGINS=*
```

Apply using the function-based form on a single route:

```
import { Router, cors } from "tina4-nodejs";

Router.get("/api/products", async (req, res) => {
    return res.json({ products: [] });
}, [cors()]);
```

Or apply the class-based form globally via `Router.use()`:

```
Router.use(CorsMiddleware);
```

The CORS middleware is active by default when registered. You do not need to handle preflight requests yourself.

You can also use the `cors()` function programmatically to inspect or apply CORS headers in your own middleware:

```
import { cors } from "tina4-nodejs";

function customCors(req, res, next) {
    const origin = req.headers["origin"] ?? "";
```

```
    // Only allow CORS for specific paths
    if (req.url.startsWith("/api/public")) {
        cors()(req, res, next);
        return;
    }

    next();
}
```

Preflight `OPTIONS` requests return `204 No Content` with the correct CORS headers. The browser caches the preflight based on `TINA4_CORS_MAX_AGE`.

---

4. Built-in RateLimiterMiddleware

The rate limiter prevents a single client from flooding your API. It uses a sliding-window algorithm that tracks requests per IP in memory. Configure via `.env`:

```
TINA4_RATE_LIMIT=60
TINA4_RATE_WINDOW=60
```

This allows 60 requests per 60-second window per IP address. Apply it:

```
Router.use(RateLimiterMiddleware);
```

When a client exceeds the limit, they receive a `429 Too Many Requests` response with a `Retry-After` header.

Rate limit headers are included in every response:

```
X-RateLimit-Limit: 60
X-RateLimit-Remaining: 57
X-RateLimit-Reset: 1711113060
Retry-After: 42
```

You can use the `RateLimiterMiddleware` class directly for custom rate limiting logic:

```
import { RateLimiterMiddleware } from "tina4-nodejs";

function customRateLimit(req, res, next) {
    const clientIp = req.socket?.remoteAddress ?? "unknown";
    const result = RateLimiterMiddleware.check(clientIp);

    if (!result.allowed) {
        res({
            error: "Too many requests",
            retry_after: result.reset
        }, 429);
        return;
    }

    res.header("X-RateLimit-Limit", String(result.limit));
    res.header("X-RateLimit-Remaining", String(result.remaining));
    res.header("X-RateLimit-Reset", String(result.reset));

    next();
}
```

Like CORS, the rate limiter is active by default based on your `.env` configuration.

---

5. Built-in RequestLogger

The `RequestLogger` middleware logs every request with timing and coloured status codes. It uses two hooks:

- `beforeLog` stamps the start time before the handler runs
- `afterLog` calculates elapsed time and prints a coloured log line

Register it globally:

```
import { Router, RequestLogger } from "tina4-nodejs";

Router.use(RequestLogger);
```

The console output looks like:

```
  200 GET /api/users 12ms
  201 POST /api/products 45ms
  404 GET /api/missing 2ms
```

Green for 2xx, yellow for 3xx, red for 4xx and 5xx.

Built-in SecurityHeadersMiddleware

The `SecurityHeadersMiddleware` adds standard security headers to every response. Register it globally:

```
import { Router, SecurityHeadersMiddleware } from "tina4-nodejs";

Router.use(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:

```
import { Router, CorsMiddleware, RateLimiterMiddleware, RequestLogger, SecurityHeadersMiddleware

Router.use(CorsMiddleware);
Router.use(RateLimiterMiddleware);
Router.use(RequestLogger);
Router.use(SecurityHeadersMiddleware);
```

Order matters. CORS handles `OPTIONS` preflight first. The rate limiter only counts real requests (not preflight). The logger measures total time including the other middleware. Security headers are added to every response.

---

6. Writing Custom Middleware

Function-Based Middleware

A middleware function takes three arguments: `req`, `res`, and `next`. It must call `next()` to continue the chain or return a response to short-circuit.

```
function myMiddleware(req, res, next) {
    // Code that runs BEFORE the route handler
    console.log("Before handler");

    // Call the next middleware or the route handler
    next();

    // Code that runs AFTER the route handler
    console.log("After handler");
}
```

Class-Based Middleware

Class-based middleware uses a naming convention: static methods prefixed with `before` run before the route handler, and methods prefixed with `after` run after it. Each method receives `(req, res)` and returns `[req, res]`.

```
class MyMiddleware {
    static beforeCheck(req, res) {
        // Runs before the route handler
        console.log("Before handler");
        return [req, res];
    }

    static afterCleanup(req, res) {
        // Runs after the route handler
        console.log("After handler");
        return [req, res];
    }
}
```

If a `before*` method sets the response status to >= 400, the route handler is skipped. This is short-circuiting.

Example: Request Timer (Class-Based)

```
class TimingMiddleware {
    static beforeStartTimer(req, res) {
        (req as any).startTime = Date.now();
        return [req, res];
    }

    static afterAddTiming(req, res) {
        const elapsed = Date.now() - ((req as any).startTime ?? Date.now());
        res.header("X-Response-Time", `${elapsed}ms`);
        return [req, res];
    }
}
```

Example: Request Logger (Function-Based)

```
function logMiddleware(req, res, next) {
    const timestamp = new Date().toISOString();
    console.log(`[${timestamp}] ${req.method} ${req.url}`);
    console.log(`  Headers: ${JSON.stringify(req.headers)}`);

    if (req.body) {
        console.log(`  Body: ${JSON.stringify(req.body)}`);
    }

    const start = Date.now();
    next();
    const duration = Date.now() - start;

    console.log(`  Completed in ${duration}ms`);
}
```

Example: Security Headers (Class-Based)

```
class CustomSecurityHeaders {
    static afterSecurity(req, res) {
        res.header("X-Content-Type-Options", "nosniff");
        res.header("X-Frame-Options", "DENY");
        res.header("X-XSS-Protection", "1; mode=block");
        res.header("Strict-Transport-Security", "max-age=31536000");
        return [req, res];
    }
}
```

Example: JSON Content-Type Enforcer (Function-Based)

```
function requireJson(req, res, next) {
    if (["POST", "PUT", "PATCH"].includes(req.method)) {
        const contentType = req.headers["content-type"] ?? "";

        if (!contentType.includes("application/json")) {
            res({
                error: "Content-Type must be application/json",
                received: contentType
            }, 415);
            return;
        }
    }

    next();
}
```

Example: Request ID (Class-Based)

```
import { randomUUID } from "crypto";

class RequestIdMiddleware {
    static beforeInjectId(req, res) {
        (req as any).requestId = randomUUID();
        return [req, res];
    }

    static afterAddHeader(req, res) {
        res.header("X-Request-ID", (req as any).requestId);
        return [req, res];
    }
}
```

Example: Input Sanitization (Class-Based)

```
class InputSanitizer {
    static beforeSanitize(req, res) {
        if (req.body && typeof req.body === "object") {
            req.body = InputSanitizer.sanitize(req.body);
        }
        return [req, res];
    }

    private static sanitize(data: Record<string, any>): Record<string, any> {
        const clean: Record<string, any> = {};
        for (const [key, value] of Object.entries(data)) {
            if (typeof value === "string") {
                clean[key] = value.replace(/[<>&"']/g, (c) =>
                    ({ "<": "&lt;", ">": "&gt;", "&": "&amp;", '"': "&quot;", "'": "&#39;" })[c]
                );
            } else if (typeof value === "object" && value !== null) {
                clean[key] = InputSanitizer.sanitize(value);
            } else {
                clean[key] = value;
            }
        }
        return clean;
    }
}
```

Example: IP Whitelist (Function-Based)

```
function ipWhitelist(req, res, next) {
    const allowedIps = (process.env.ALLOWED_IPS ?? "127.0.0.1").split(",");
    const clientIp = req.socket?.remoteAddress ?? "";

    if (!allowedIps.includes(clientIp)) {
        res({ error: "Access denied", your_ip: clientIp }, 403);
        return;
    }

    next();
}
```

---

7. Applying Middleware to Routes

Apply middleware to a single route by passing an array as the third argument. Both function-based and class-based middleware work:

```
import { Router } from "tina4-nodejs";

// Function-based middleware on a single route
Router.get("/api/data", async (req, res) => {
    return res.json({ data: [1, 2, 3] });
}, [logMiddleware]);

// Class-based middleware on a single route
Router.get("/api/stats", async (req, res) => {
    return res.json({ stats: {} });
}, [TimingMiddleware]);
```

Apply multiple middleware by listing them in the array:

```
Router.post("/api/items", async (req, res) => {
    return res.status(201).json({ item: req.body });
}, [RequestIdMiddleware, CustomSecurityHeaders, requireJson]);
```

You can mix function-based and class-based middleware freely. They run in the order you list them. In the example above: `RequestIdMiddleware` runs first (before and after hooks), then `CustomSecurityHeaders`, then `requireJson`, then the route handler.

---

8. Middleware on Route Groups

Apply middleware to all routes in a group:

```
import { Router } from "tina4-nodejs";

Router.group("/api/v1", (group) => {
    group.get("/users", async (req, res) => {
        return res.json({ users: [] });
    });

    group.post("/users", async (req, res) => {
        return res.status(201).json({ created: true });
```

```
    });

    group.get("/products", async (req, res) => {
        return res.json({ products: [] });
    });
}, [TimingMiddleware, CustomSecurityHeaders]);
```

Every route inside the group now has `TimingMiddleware` and `CustomSecurityHeaders` applied. You can still add route-specific middleware on top:

```
Router.group("/api/v1", (group) => {
    group.get("/public", async (req, res) => {
        // Only group middleware runs
        return res.json({ public: true });
    });

    group.post("/admin", async (req, res) => {
        // Group middleware + authMiddleware both run
        return res.json({ admin: true });
    }, [authMiddleware]);
}, [RequestIdMiddleware]);
```

Group middleware always runs before route-specific middleware. This means authentication checks at the group level cannot be bypassed by individual routes.

---

9. Execution Order

Stacked middleware forms a nested pipeline. Requests travel inward. Responses travel outward:

```
Request arrives
  → logMiddleware (before)
    → authMiddleware (before)
      → timerMiddleware (before)
        → route handler
      → timerMiddleware (after)
    → authMiddleware (after)
  → logMiddleware (after)
Response sent
```

Each middleware wraps around the next one. The outermost middleware runs first on the way in and last on the way out.

Here is a concrete example showing the order:

```
function middlewareA(req, res, next) {
    console.log("A: before");
    next();
    console.log("A: after");
}

function middlewareB(req, res, next) {
    console.log("B: before");
    next();
    console.log("B: after");
}

Router.get("/test", async (req, res) => {
```

```
    console.log("Handler");
    return res.json({ ok: true });
}, [middlewareA, middlewareB]);
```

When you request `/test`, the console shows:

```
A: before
B: before
Handler
B: after
A: after
```

---

10. Short-Circuiting

Skip `next()` and the chain stops cold. The route handler never runs. This is how blocking middleware works:

```
function maintenanceMode(req, res, next) {
    if (process.env.MAINTENANCE_MODE === "true") {
        // Allow health checks through
        if (req.url === "/health") {
            next();
            return;
        }
        res({
            error: "Service is under maintenance. Please try again later.",
            retry_after: 300
        }, 503);
        return;
    }

    next();
}
```

When `MAINTENANCE_MODE=true`, every request gets a 503 response without reaching any route handler. The health check endpoint still works -- a common pattern for load balancers.

Conditional Short-Circuit

```
function requireApiKey(req, res, next) {
    const apiKey = req.headers["x-api-key"] ?? "";

    if (!apiKey) {
        res({ error: "API key required" }, 401);
        return;
    }

    // Validate the key against a list or database
    const validKeys = ["key-abc-123", "key-def-456", "key-ghi-789"];
    if (!validKeys.includes(apiKey)) {
        res({ error: "Invalid API key" }, 403);
        return;
    }

    // Attach the key info to the request for the handler to use
    (req as any).apiKey = apiKey;

    next();
}
```

```
# No key -- 401
curl http://localhost:7148/api/data
```

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

```
# Invalid key -- 403
curl http://localhost:7148/api/data -H "X-API-Key: wrong-key"
```

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

```
# Valid key -- 200
curl http://localhost:7148/api/data -H "X-API-Key: key-abc-123"
```

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

---

11. Modifying Request and Response

Middleware can modify the request before it reaches the handler, and the response before it reaches the client.

Adding Data to the Request

```
function injectUserAgent(req, res, next) {
    const ua = req.headers["user-agent"] ?? "";

    (req as any).isMobile = ua.includes("Mobile") || ua.includes("Android") || ua.includes("iPho
    (req as any).isBot = ua.toLowerCase().includes("bot") || ua.toLowerCase().includes("spider")

    next();
}
```

Now the route handler can access `req.isMobile` and `req.isBot`:

```
Router.get("/api/content", async (req, res) => {
    const isMobile = (req as any).isMobile;
```

```
    if (isMobile) {
        return res.json({ layout: "compact", images: "low-res" });
    }

    return res.json({ layout: "full", images: "high-res" });
}, [injectUserAgent]);
```

Modifying the Response

```
function addSecurityHeaders(req, res, next) {
    next();

    // Add security headers to every response after the handler runs
    res.header("X-Content-Type-Options", "nosniff");
    res.header("X-Frame-Options", "DENY");
    res.header("X-XSS-Protection", "1; mode=block");
    res.header("Strict-Transport-Security", "max-age=31536000; includeSubDomains");
}
```

Adding a Request ID

```
function addRequestId(req, res, next) {
    const { randomUUID } = require("crypto");
    (req as any).requestId = randomUUID();
    res.header("X-Request-Id", (req as any).requestId);
    next();
}
```

The request ID follows the request through every layer. Log it in your handler, return it in error responses, and use it to trace issues across services.

---

12. Real-World Example: JWT Authentication Middleware

This class-based middleware verifies JWT tokens on protected routes. It uses the `before*` / `after*` convention.

```
import { Auth } from "tina4-nodejs";

class JwtAuthMiddleware {
    static beforeVerifyToken(req, res) {
        const authHeader = req.headers["authorization"] ?? "";

        if (!authHeader || !authHeader.startsWith("Bearer ")) {
            res(JSON.stringify({ error: "Authorization header required" }), 401);
            return [req, res];
        }

        const token = authHeader.slice(7);
        const secret = process.env.TINA4_SECRET || "tina4-default-secret";
        const payload = Auth.validToken(token, secret);

        if (!payload) {
            res(JSON.stringify({ error: "Invalid or expired token" }), 401);
            return [req, res];
        }

        // Attach the decoded payload to the request
```

```
        (req as any).user = payload;
        return [req, res];
    }
}
```

Apply it to a group of protected routes:

```
import { Router } from "tina4-nodejs";

Router.group("/api/protected", (group) => {
    group.get("/profile", async (req, res) => {
        return res.json({ user: (req as any).user });
    });

    group.post("/settings", async (req, res) => {
        const userId = (req as any).user.sub;
        return res.json({ updated: true, user_id: userId });
    });
}, [JwtAuthMiddleware]);
```

The middleware short-circuits with 401 if the token is missing or invalid. The route handler never runs. If the token is valid, the decoded payload is available as `req.user`.

---

13. Real-World Example: API Key Middleware with Database Lookup

```
import { Database } from "tina4-nodejs";

async function apiKeyMiddleware(req, res, next) {
    const apiKey = req.headers["x-api-key"] ?? "";

    if (!apiKey) {
        res({
            error: "API key required. Send it in the X-API-Key header."
        }, 401);
        return;
    }

    const db = new Database();
    const keyRecord = await db.fetchOne(
        "SELECT id, name, rate_limit, is_active FROM api_keys WHERE key_value = ?",
        [apiKey]
    );

    if (!keyRecord) {
        res({ error: "Invalid API key" }, 403);
        return;
    }

    if (!keyRecord.is_active) {
        res({ error: "API key has been deactivated" }, 403);
        return;
    }

    // Update last used timestamp
    await db.execute(
        "UPDATE api_keys SET last_used_at = ?, request_count = request_count + 1 WHERE id = ?",
        [new Date().toISOString(), keyRecord.id]
    );

    // Attach key info to request
    (req as any).apiKeyId = keyRecord.id;
    (req as any).apiKeyName = keyRecord.name;

    next();
}
```

The middleware checks the database on every request. Tina4's `Database` class uses connection pooling internally, so creating a new instance per request is safe. For high-traffic APIs, cache your key lookups in memory with a TTL instead of querying on every request.

---

14. Real-World Middleware Stack

```
import { Router } from "tina4-nodejs";

Router.group("/api/v1", (group) => {
    group.get("/products", async (req, res) => {
        return res.json({ products: [
            { id: 1, name: "Widget", price: 9.99 },
            { id: 2, name: "Gadget", price: 19.99 }
        ]});
    });

    group.post("/products", async (req, res) => {
        return res.status(201).json({
            id: 3,
            name: req.body.name ?? "Unknown",
            price: parseFloat(req.body.price ?? 0)
        });
    });
}, [addRequestId, logMiddleware, cors(), requireApiKey]);
```

Four middleware layers. Each does one job. The request ID comes first so every log line and error response includes it. The logger records the request. CORS handles browser preflight. The API key check gates access. The route handler never sees any of that work.

---

15. Exercise: Build an API Key Middleware System

Build a complete API key system with key management and usage tracking.

Requirements

- Create a migration for an `api_keys` table: `id`, `name`, `key_value` (unique), `is_active` (default true), `rate_limit` (default 100), `request_count` (default 0), `last_used_at`, `created_at`
- Build these endpoints:

Method	Path	Middleware	Description
`POST`	`/admin/api-keys`	Auth	Create a new API key (generate random key)
`GET`	`/admin/api-keys`	Auth	List all API keys with usage stats
`DELETE`	`/admin/api-keys/:id`	Auth	Deactivate an API key
`GET`	`/api/data`	API Key	Protected endpoint -- requires valid API key

`GET`	`/api/status`	API Key	Another protected endpoint

- The API key middleware should:
- Check `X-API-Key` header
- Validate against the database
- Reject deactivated keys
- Track usage (increment count, update last*used*at)
- Attach key info to the request

Test with:

```
# Create an API key (requires auth token from Chapter 8)
curl -X POST http://localhost:7148/admin/api-keys \
  -H "Authorization: Bearer YOUR_AUTH_TOKEN" \
  -H "Content-Type: application/json" \
  -d '{"name": "Mobile App"}'

# Use the API key
curl http://localhost:7148/api/data \
  -H "X-API-Key: THE_GENERATED_KEY"

# List keys with stats
curl http://localhost:7148/admin/api-keys \
  -H "Authorization: Bearer YOUR_AUTH_TOKEN"
```

---

16. Solution

Migration

Create `migrations/20260322170000_create_api_keys_table.sql`:

```
-- UP
CREATE TABLE api_keys (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    name TEXT NOT NULL,
    key_value TEXT NOT NULL UNIQUE,
    is_active INTEGER NOT NULL DEFAULT 1,
    rate_limit INTEGER NOT NULL DEFAULT 100,
    request_count INTEGER NOT NULL DEFAULT 0,
    last_used_at TEXT,
    created_at TEXT DEFAULT CURRENT_TIMESTAMP
);

CREATE UNIQUE INDEX idx_api_keys_key ON api_keys (key_value);

-- DOWN
DROP INDEX IF EXISTS idx_api_keys_key;
DROP TABLE IF EXISTS api_keys;
```

Routes

Create `src/routes/apiKeys.ts`:

```
import { Router, Auth, Database } from "tina4-nodejs";
import { randomBytes } from "crypto";

async function authMiddleware(req, res, next) {
    const authHeader = req.headers["authorization"] ?? "";
    if (!authHeader || !authHeader.startsWith("Bearer ")) {
        res({ error: "Authorization required" }, 401);
        return;
    }

    const token = authHeader.slice(7);
    const secret = process.env.TINA4_SECRET || "tina4-default-secret";
    const payload = Auth.validToken(token, secret);

    if (!payload) {
        res({ error: "Invalid or expired token" }, 401);
        return;
    }

    (req as any).user = payload;
    next();
}

async function apiKeyMiddleware(req, res, next) {
    const apiKey = req.headers["x-api-key"] ?? "";

    if (!apiKey) {
        res({ error: "API key required. Send in X-API-Key header." }, 401);
        return;
    }

    const db = new Database();
    const keyRecord = await db.fetchOne(
        "SELECT id, name, is_active FROM api_keys WHERE key_value = ?",
        [apiKey]
    );

    if (!keyRecord) {
        res({ error: "Invalid API key" }, 403);
        return;
    }

    if (!keyRecord.is_active) {
        res({ error: "API key has been deactivated" }, 403);
        return;
    }

    await db.execute(
        "UPDATE api_keys SET last_used_at = ?, request_count = request_count + 1 WHERE id = ?",
        [new Date().toISOString(), keyRecord.id]
    );

    (req as any).apiKeyId = keyRecord.id;
    (req as any).apiKeyName = keyRecord.name;

    next();
```

```
}

// Admin routes -- require auth token
Router.group("/admin", (group) => {
    group.post("/api-keys", async (req, res) => {
        const db = new Database();
        const name = req.body?.name ?? "Unnamed Key";
        const keyValue = `tk_${randomBytes(24).toString("hex")}`;

        await db.execute(
            "INSERT INTO api_keys (name, key_value) VALUES (?, ?)",
            [name, keyValue]
        );

        const key = await db.fetchOne(
            "SELECT * FROM api_keys WHERE key_value = ?",
            [keyValue]
        );

        return res.status(201).json({ message: "API key created", key });
    });

    group.get("/api-keys", async (req, res) => {
        const db = new Database();
        const keys = await db.fetch(
            "SELECT id, name, key_value, is_active, rate_limit, request_count, last_used_at, cre
        );
        return res.json({ keys, count: keys.length });
    });

    group.delete("/api-keys/:id", async (req, res) => {
        const db = new Database();
        const keyId = req.params.id;

        const existing = await db.fetchOne(
            "SELECT id FROM api_keys WHERE id = ?",
            [keyId]
        );

        if (!existing) {
            return res.status(404).json({ error: "API key not found" });
        }

        await db.execute(
            "UPDATE api_keys SET is_active = 0 WHERE id = ?",
            [keyId]
        );

        return res.json({ message: "API key deactivated" });
    });
}, [authMiddleware]);

// Public API routes -- require API key
Router.group("/api", (group) => {
    group.get("/data", async (req, res) => {
        return res.json({
            data: [1, 2, 3, 4, 5],
            api_key: (req as any).apiKeyName
        });
```

```
    });

    group.get("/status", async (req, res) => {
        return res.json({
            status: "operational",
            api_key: (req as any).apiKeyName
        });
    });
}, [apiKeyMiddleware]);
```

---

17. Gotchas

1. Forgetting to Call next()

Problem: The route handler never runs. The request hangs or times out.

Cause: You forgot to call `next()` after your middleware logic. Without `next()`, the chain stops and the handler never executes.

Fix: Always call `next()` to continue the chain. The `next()` function takes no arguments. If you intend to block the request, return a response instead of leaving the request hanging.

2. Middleware Modifies Response After It Is Sent

Problem: Headers or cookies you add in the "after" phase of middleware do not appear in the response.

Cause: The response was already finalized by the route handler. In Node.js, once the response body is sent, headers cannot be modified.

Fix: In the "after" phase, modify the result before it reaches the client. Some modifications need to happen in the "before" phase instead. For class-based middleware, the `after*` methods run before the response is finalized, so they can still add headers.

3. Middleware Applied to Wrong Routes

Problem: Your API key middleware runs on public routes that should not require a key.

Cause: The middleware is applied to the group, and the public route is inside that group.

Fix: Move public routes outside the group. Or create two groups -- one for public endpoints with no auth middleware, one for protected endpoints with it. For custom middleware, you can also check the path inside the middleware and skip the check for specific routes.

4. Middleware Execution Order Surprises

Problem: Your auth check runs after your logging middleware, but you wanted it to run first.

Cause: Middleware in the array `[a, b, c]` runs in left-to-right order: `a` wraps `b` wraps `c` wraps handler.

Fix: Put the middleware you want to run first at the leftmost position: `[authMiddleware, logMiddleware]`.

5. Error in Middleware Breaks the Chain

Problem: An unhandled exception in middleware causes a 500 error without reaching the route handler.

Cause: If middleware throws an exception before calling `next()`, no subsequent middleware or the handler runs.

Fix: Wrap risky middleware code in try/catch and return an appropriate error response instead of letting the exception propagate:

```
function safeMiddleware(req, res, next) {
    try {
        // risky operation
        const result = somethingThatMightFail();
        (req as any).result = result;
        next();
    } catch (err) {
        res({ error: "Internal middleware error" }, 500);
    }
}
```

6. Database Connections in Middleware

Problem: Opening a database connection in middleware that runs on every request causes connection pool exhaustion.

Cause: Each middleware call creates a new `Database()` instance.

Fix: Tina4's `Database` class uses connection pooling internally, so this is usually safe. But if you see issues under high traffic, cache your database lookups (like API keys) in memory with a TTL instead of querying on every request.

7. Modifying Request in Middleware Does Not Persist

Problem: You set `req.customField = "value"` in middleware, but the route handler does not see it.

Cause: TypeScript's type system does not know about your custom property. In some cases, the request object may be copied between middleware stages.

Fix: Use type assertion `(req as any).customField` consistently. If the property is not persisting, check that you are modifying the same request object that is passed to `next()`. Do not create a new request object.

8. CORS Preflight Returns 404

Problem: The browser sends an `OPTIONS` request and gets a 404, blocking the actual request.

Cause: No middleware handles the preflight `OPTIONS` request for that route.

Fix: Apply `CorsMiddleware` to the group or globally with `Router.use()`. It handles `OPTIONS` on its own. Make sure CORS middleware is registered before the rate limiter so preflight requests do not count against the rate limit.

9. Rate Limiter Counts Preflight Requests

Problem: Browsers burn through the rate limit with `OPTIONS` requests before making actual API calls.

Cause: The rate limiter runs before the CORS middleware, so it counts preflight requests.

Fix: Put `CorsMiddleware` before `RateLimiterMiddleware` in the registration order. CORS handles the preflight and returns early. The rate limiter only sees real requests.

10. Middleware File Not Auto-Loaded

Problem: You defined middleware in a file, but Tina4 does not recognize it.

Cause: The file is not inside the auto-loaded directory.

Fix: Put middleware files inside `src/routes/`. Tina4 auto-loads all files in that directory. If you place middleware in a separate folder, import it in a file inside `src/routes/`.

11. Short-Circuiting Skips Cleanup

Problem: Your timing middleware never logs the "after" phase because an earlier middleware short-circuited the request.

Cause: When middleware returns a response without calling `next()`, downstream middleware never runs -- including the "after" hooks of outer middleware.

Fix: Put timing and logging middleware at the outermost layer. They wrap everything else, so their "after" code runs regardless of what happens inside.

12. Middleware Must Be Passed as Function References

Problem: You passed a middleware name as a string and nothing happened.

Cause: The middleware array expects function or class references, not strings.

Fix: Pass middleware as references in an array: `[myMiddleware]`, not `["myMiddleware"]`. If you use a factory function like `cors()`, call it to get the actual middleware function.

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 export to remember. The framework enforces this before your handler runs.

```
// src/routes/api/orders/post.ts
import type { Tina4Request, Tina4Response } from "tina4-nodejs";

export default async function (req: Tina4Request, res: Tina4Response) {
    // This handler ONLY runs if the request carries a valid Bearer token.
    // Without one, the framework returns 401 before your code executes.
    return res.status(201).json({ created: true });
}
```

Test it without a token:

```
curl -X POST http://localhost:7148/api/orders \
  -H "Content-Type: application/json" \
  -d '{"product": "widget"}'
# 401 Unauthorized
```

Test it with a valid token:

```
curl -X POST http://localhost:7148/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. Export `meta` with `noAuth: true`:

```
// src/routes/api/webhooks/stripe/post.ts
import type { Tina4Request, Tina4Response } from "tina4-nodejs";

export const meta = { noAuth: true };

export default async function (req: Tina4Request, res: Tina4Response) {
    // No token required. Stripe can POST here freely.
    return res.json({ received: true });
}
```

Protecting a GET Route

Admin dashboards, user profiles, account settings — some pages need protection even though they only read data. Export `meta` with `secured: true`:

```
// src/routes/api/admin/users/get.ts
import type { Tina4Request, Tina4Response } from "tina4-nodejs";

export const meta = { secured: true };

export default async function (req: Tina4Request, res: Tina4Response) {
    // Requires a valid Bearer token, even though it's a GET.
```

```
    return res.json({ users: [] });
}
```

The Rule

Method	Default	Override
GET, HEAD, OPTIONS	Public	`{ secured: true }` to protect
POST, PUT, PATCH, DELETE	Auth required	`{ noAuth: true }` to open

One export. 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** — `req.body.formToken`
- **Request header** — `X-Form-Token`

If the token is missing or invalid, the middleware returns 403 before your handler runs.

AJAX Requests

For JavaScript-driven forms, send the token as a header:

```
// frond.min.js handles this automatically via saveForm()
// For manual AJAX, extract the token from the hidden field:
const token = document.querySelector('input[name="formToken"]').value;

fetch("/api/profile", {
    method: "POST",
    headers: {
        "Content-Type": "application/json",
        "X-Form-Token": token
    },
    body: JSON.stringify({ name: "Alice" })
});
```

Tokens in Query Strings — Blocked

Tokens must never appear in URLs. Query strings are logged in server access logs, browser history, and referrer headers. A token in the URL is a token anyone can steal.

Tina4 rejects any request that carries `formToken` in the query string and logs a warning:

```
CSRF token found in query string — rejected for security.
Use POST body or X-Form-Token header instead.
```

Skipping CSRF Validation

Three scenarios skip CSRF checks automatically:

- **GET, HEAD, OPTIONS** — Safe methods don't modify state.
- **Routes with** `noAuth: true` — 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

```
// src/routes/api/login/post.ts
import type { Tina4Request, Tina4Response } from "tina4-nodejs";
import { getToken, checkPassword } from "tina4-nodejs";

export const meta = { noAuth: true };

export default async function (req: Tina4Request, res: Tina4Response) {
    const email: string = req.body.email || "";
    const password: string = req.body.password || "";

    if (!email || !password) {
        return res.status(400).json({ error: "Email and password required" });
    }

    // Look up user (replace with your database query)
    const user = await db.fetchOne(
        "SELECT id, email, password_hash, role FROM users WHERE email = ?",
        [email]
    );

    if (!user) {
        return res.status(401).json({ error: "Invalid credentials" });
    }

    // Verify password
    if (!checkPassword(password, user.password_hash)) {
        return res.status(401).json({ error: "Invalid credentials" });
    }

    // Generate token with user claims
    const secret = process.env.TINA4_SECRET || "tina4-default-secret";
    const token: string = getToken({
        sub: user.id,
        email: user.email,
        role: user.role,
    }, secret);

    // Store user in session
    req.session.set("user_id", user.id);
    req.session.set("email", user.email);
    req.session.set("role", user.role);

    return res.json({
        token,
        user: { id: user.id, email: user.email },
    });
}
```

The `meta.noAuth` flag 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

```
// src/routes/dashboard/get.ts
import type { Tina4Request, Tina4Response } from "tina4-nodejs";

export default async function (req: Tina4Request, res: Tina4Response) {
    const userId: string | undefined = req.session.get("user_id");

    if (!userId) {
        return res.redirect("/login");
    }

    return res.render("dashboard.twig", {
        email: req.session.get("email"),
        role: req.session.get("role"),
    });
}
```

Logout — Destroying the Session

```
// src/routes/api/logout/post.ts
import type { Tina4Request, Tina4Response } from "tina4-nodejs";

export const meta = { noAuth: true };

export default async function (req: Tina4Request, res: Tina4Response) {
    req.session.destroy();
    return res.json({ logged_out: true });
}
```

---

7. Handling Expired Sessions

Sessions expire. Tokens expire. The user clicks a link and finds themselves staring at a broken page or a cryptic error. A good security implementation handles expiry gracefully.

The Pattern: Redirect to Login, Then Back

When a session expires mid-use, the user should:

- See a login page — not an error.
- Log in again.
- Land on the page they were trying to reach — not the home page.

```
// src/routes/account/settings/get.ts
import type { Tina4Request, Tina4Response } from "tina4-nodejs";

export default async function (req: Tina4Request, res: Tina4Response) {
    const userId: string | undefined = req.session.get("user_id");

    if (!userId) {
        // Remember where they wanted to go
        const returnUrl: string = encodeURIComponent(req.url);
        return res.redirect(`/login?redirect=${returnUrl}`);
    }

    return res.render("settings.twig", { user_id: userId });
}
```

The login handler reads the `redirect` parameter after successful authentication:

```
// src/routes/api/login/post.ts
import type { Tina4Request, Tina4Response } from "tina4-nodejs";
import { getToken, checkPassword } from "tina4-nodejs";

export const meta = { noAuth: true };

export default async function (req: Tina4Request, res: Tina4Response) {
    // ... validate credentials ...

    const redirectUrl: string = req.params.redirect || "/dashboard";

    return res.json({
        token,
        redirect: redirectUrl,
```

```
    });
}
```

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:

```
// src/routes/api/login/post.ts
import type { Tina4Request, Tina4Response } from "tina4-nodejs";
import { getToken, checkPassword } from "tina4-nodejs";

export const meta = { noAuth: true };

// Simple in-memory rate limiter for login
const loginAttempts: Map<string, { count: number; resetAt: number }> = new Map();
```

```
export default async function (req: Tina4Request, res: Tina4Response) {
    const ip: string = req.ip;
    const now: number = Date.now();
    const windowMs: number = 60_000; // 60 seconds
    const maxAttempts: number = 5;

    // Check rate limit
    const entry = loginAttempts.get(ip);
    if (entry && now < entry.resetAt) {
        if (entry.count >= maxAttempts) {
            const retryAfter: number = Math.ceil((entry.resetAt - now) / 1000);
            return res.status(429).json({
                error: "Too many login attempts",
                retry_after: retryAfter,
            });
        }
        entry.count++;
    } else {
        loginAttempts.set(ip, { count: 1, resetAt: now + windowMs });
    }

    // ... login logic ...
}
```

---

9. CORS and Credentials

When your frontend runs on a different origin than your API (common in development), CORS controls whether the browser sends cookies and auth headers.

Tina4 handles CORS automatically. The relevant security settings:

```
TINA4_CORS_ORIGINS=*
TINA4_CORS_CREDENTIALS=true
```

Two rules to remember:

- `TINA4_CORS_ORIGINS=*` **with** `TINA4_CORS_CREDENTIALS=true` is invalid per the CORS spec. Tina4 handles this — when origin is `*`, the credentials header is not sent. But in production, list your actual origins.
- **Cookies need** `SameSite=None; Secure` for true cross-origin requests. If your API is on `api.example.com` and your frontend is on `app.example.com`, the default `Lax` cookie works because they share the same registrable domain. Different domains need `SameSite=None`.

Production CORS:

```
TINA4_CORS_ORIGINS=https://app.example.com,https://admin.example.com
TINA4_CORS_CREDENTIALS=true
```

---

10. Security Checklist

Before you deploy, verify:

- [ ] `TINA4_SECRET` is set to a long, random string — not the default.
- [ ] `TINA4_DEBUG=false` in production.
- [ ] `TINA4_HSTS=31536000` if serving over HTTPS.
- [ ] `TINA4_CORS_ORIGINS` lists your actual domains — not `*`.
- [ ] `TINA4_CSRF=true` (the default) for any browser-facing application.
- [ ] Login route uses `noAuth: true` and validates credentials before issuing tokens.
- [ ] Session is regenerated after login (prevents session fixation).
- [ ] Passwords are hashed with `hashPassword()` — 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: Export `meta` with `noAuth: true` 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 export `meta` with `noAuth: true` on the route.

4. "My Content-Security-Policy blocks inline scripts"

Cause: The default CSP is `default-src 'self'`, which blocks inline `<script>` tags and `onclick` handlers.

Fix: Move scripts to external `.js` files (the right approach) or relax the CSP:

```
TINA4_CSP=default-src 'self'; script-src 'self' 'unsafe-inline'
```

Prefer external scripts. Inline scripts are an XSS vector.

5. "User stays logged in after session expires"

Cause: The frontend stores a JWT in localStorage. The token is still valid even after the session is destroyed server-side.

Fix: Check the session on every page load. If the session is gone, redirect to login regardless of the token. Tokens authenticate API calls; sessions track server-side state. Both must be valid.

Exercise: Secure Contact Form

Build a public contact form that:

- Does not require login (`noAuth: true`).
- 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/get.ts
import type { Tina4Request, Tina4Response } from "tina4-nodejs";

export default async function (req: Tina4Request, res: Tina4Response) {
    return res.render("contact.twig", { title: "Contact Us" });
}
```

```
// src/routes/api/contact/post.ts
import type { Tina4Request, Tina4Response } from "tina4-nodejs";

export const meta = { noAuth: true };

// Simple in-memory rate limiter for contact submissions
const submissions: Map<string, { count: number; resetAt: number }> = new Map();

export default async function (req: Tina4Request, res: Tina4Response) {
    // Rate limit: 3 submissions per 60 seconds per IP
    const ip: string = req.ip;
    const now: number = Date.now();
    const windowMs: number = 60_000;
    const maxSubmissions: number = 3;

    const entry = submissions.get(ip);
    if (entry && now < entry.resetAt) {
        if (entry.count >= maxSubmissions) {
            return res.status(429).json({ error: "Too many submissions. Try again later." });
        }
        entry.count++;
    } else {
        submissions.set(ip, { count: 1, resetAt: now + windowMs });
    }

    const name: string = (req.body.name || "").trim();
```

```
    const email: string = (req.body.email || "").trim();
    const message: string = (req.body.message || "").trim();

    if (!name || !email || !message) {
        return res.status(400).json({ error: "All fields are required" });
    }

    await db.insert("contact_messages", {
        name,
        email,
        message,
    });
    await db.commit();

    return res.json({ success: true, message: "Thank you for your message" });
}
```

```
{# src/templates/contact.twig #}
{% extends "base.twig" %}
{% block title %}Contact Us{% endblock %}
{% block content %}
<div class="container mt-5" style="max-width: 500px;">
    <h2>{{ title }}</h2>
    <form id="contactForm">
        {{ form_token() }}
        <div class="mb-3">
            <label for="name">Name</label>
            <input type="text" name="name" id="name" class="form-control"
                   placeholder="Jane Smith" required>
        </div>
        <div class="mb-3">
            <label for="email">Email</label>
            <input type="email" name="email" id="email" class="form-control"
                   placeholder="jane@example.com" required>
        </div>
        <div class="mb-3">
            <label for="message">Message</label>
            <textarea name="message" id="message" class="form-control" rows="4"
                      placeholder="How can we help?" required></textarea>
        </div>
        <button type="button" class="btn btn-primary"
                onclick="saveForm('contactForm', '/api/contact', 'contactMsg')">
            Send Message
        </button>
        <div id="contactMsg" class="mt-3"></div>
    </form>
</div>
{% endblock %}
```

The form is public. The CSRF token is present. The `noAuth: true` export opens the route. The middleware validates the token. The database stores the message. The user sees confirmation.

Five moving parts. Zero security holes. The framework handles the rest.

Caching

1. From 800ms to 3ms

Your product catalog page runs 12 database queries. 800 milliseconds to render. Every visitor triggers the same queries, the same template rendering, the same JSON serialization -- for data that changes once a day.

Add caching. The first request takes 800ms. The next 10,000 take 3ms each. A 266x improvement from one line of configuration.

Caching stores the result of expensive operations for reuse. Tina4 provides three levels: response caching (entire HTTP responses), database query caching, and a direct cache API for custom use cases.

---

2. Response Caching with ResponseCache Middleware

The fastest way to cache is at the HTTP response level. The `ResponseCache` middleware stores the complete response (headers and body) and serves it on subsequent requests without calling your route handler at all.

```
import { Router } from "tina4-nodejs";

Router.get("/api/products", async (req, res) => {
    // This handler runs 12 database queries and takes 800ms
    // With ResponseCache, it runs once every 5 minutes

    console.log("Handler called -- should only appear once every 5 minutes");

    const products = [
        { id: 1, name: "Wireless Keyboard", price: 79.99 },
        { id: 2, name: "USB-C Hub", price: 49.99 },
        { id: 3, name: "Monitor Stand", price: 129.99 }
    ];

    return res.json({ products, generated_at: new Date().toISOString() });
}, "ResponseCache:300");
```

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:7148/api/products
```

```
{
  "products": [
    {"id": 1, "name": "Wireless Keyboard", "price": 79.99},
    {"id": 2, "name": "USB-C Hub", "price": 49.99},
    {"id": 3, "name": "Monitor Stand", "price": 129.99}
```

```
  ],
  "generated_at": "2026-03-22T14:30:00.000Z"
}
```

Call it again within 5 minutes:

```
curl http://localhost:7148/api/products
```

```
{
  "products": [
    {"id": 1, "name": "Wireless Keyboard", "price": 79.99},
    {"id": 2, "name": "USB-C Hub", "price": 49.99},
    {"id": 3, "name": "Monitor Stand", "price": 129.99}
  ],
  "generated_at": "2026-03-22T14:30:00.000Z"
}
```

Notice the `generated_at` timestamp is the same. The handler did not run -- the response came from cache.

Cache Headers

The `ResponseCache` middleware sets cache-related headers on every response:

```
X-Cache: HIT
X-Cache-TTL: 247
Cache-Control: public, max-age=300
```

- `X-Cache: HIT` or `X-Cache: MISS` tells you whether the response came from cache
- `X-Cache-TTL` shows the remaining time-to-live in seconds
- `Cache-Control` enables browser and CDN caching

Caching with Query Parameters

By default, the cache key includes the full URL with query parameters. `/api/products?page=1` and `/api/products?page=2` are cached separately:

```
Router.get("/api/products", async (req, res) => {
    const page = parseInt(req.query.page ?? "1", 10);
    const limit = 20;
    const offset = (page - 1) * limit;

    return res.json({
        page,
        products: [],
        generated_at: new Date().toISOString()
    });
}, "ResponseCache:300");
```

```
curl "http://localhost:7148/api/products?page=1"  # Cache MISS, stores for page=1
curl "http://localhost:7148/api/products?page=2"  # Cache MISS, stores for page=2
curl "http://localhost:7148/api/products?page=1"  # Cache HIT
```

What Not to Cache

Do not use `ResponseCache` on:

- **POST, PUT, PATCH, DELETE routes**: Only GET responses should be cached

- **User-specific endpoints**: `/api/profile` returns different data for each user
- **Real-time data**: Stock prices, live scores, chat messages
- **Authenticated endpoints**: Unless the cache is scoped per user

```
// GOOD: Public, rarely changing data
Router.get("/api/categories", async (req, res) => {
    return res.json({ categories: [] });
}, "ResponseCache:3600");

// BAD: User-specific data -- do NOT cache
Router.get("/api/profile", async (req, res) => {
    return res.json(req.user);
}, "auth_middleware");
```

---

3. Memory Cache (Default)

Tina4's cache system stores data in memory by default. No configuration needed.

```
# This is the default -- you do not need to set it explicitly
TINA4_CACHE_BACKEND=memory
```

Memory cache is the fastest option (no disk I/O, no network calls) but it resets when the server restarts. It is ideal for development and single-server deployments where losing the cache on restart is acceptable.

---

4. Redis Cache

For production deployments where you want cache persistence across server restarts and shared cache across multiple server instances, use Redis:

```
TINA4_CACHE_BACKEND=redis
TINA4_CACHE_URL=localhost
TINA4_CACHE_URL=6379
TINA4_CACHE_URL=your-redis-password
```

Your code does not change. The `cacheGet`, `cacheSet`, and `ResponseCache` middleware all work the same way. Only the storage backend changes.

Why Redis?

- Cache survives server restarts
- Shared across multiple server instances (behind a load balancer)
- Sub-millisecond reads and writes
- Built-in key expiry (TTL cleanup is automatic)
- Same Redis instance can serve sessions, cache, and queues

---

5. File Cache

If you want cache persistence but do not have Redis, use file-based caching:

```
TINA4_CACHE_BACKEND=file
TINA4_CACHE_DIR=/path/to/cache/directory
```

File cache stores each cache entry as a file on disk. It is slower than memory or Redis but survives server restarts without extra infrastructure.

When to Use File Cache

- You need cache persistence but cannot run Redis
- Your hosting environment is limited (shared hosting, no external services)
- Cache entries are large and you do not want them in memory

---

6. Direct Cache API

For custom caching logic, use the `cacheGet`, `cacheSet`, and `cacheDelete` functions:

cacheSet

```
import { cacheSet } from "tina4-nodejs";

// Cache a value for 300 seconds
await cacheSet("product:42", {
    id: 42,
    name: "Wireless Keyboard",
    price: 79.99,
    inStock: true
}, 300);

// Cache a string
await cacheSet("exchange_rate:USD_EUR", "0.92", 3600);

// Cache indefinitely (no TTL)
await cacheSet("app:config", { theme: "dark", lang: "en" });
```

cacheGet

```
import { cacheGet, cacheSet } from "tina4-nodejs";

const product = await cacheGet("product:42");
// Returns the cached value, or null if not found or expired

if (product === null) {
    // Cache miss -- fetch from database
    const freshProduct = await fetchProductFromDatabase(42);
    await cacheSet("product:42", freshProduct, 300);
}

return res.json(product);
```

cacheDelete

```
import { cacheDelete } from "tina4-nodejs";

// Delete a specific key
await cacheDelete("product:42");

// Delete multiple keys
await cacheDelete("product:42");
await cacheDelete("product:43");
await cacheDelete("product:44");
```

Real-World Pattern: Cache-Aside

The most common caching pattern is cache-aside (also called lazy loading):

```
import { Router, cacheGet, cacheSet } from "tina4-nodejs";
import { Database } from "tina4-nodejs/orm";

Router.get("/api/products/{id:int}", async (req, res) => {
    const id = req.params.id;
    const cacheKey = `product:${id}`;

    // 1. Try the cache first
    let product = await cacheGet(cacheKey);

    if (product !== null) {
        // Cache hit -- return immediately
        return res.json({ ...product, source: "cache" });
    }

    // 2. Cache miss -- fetch from database
    const db = Database.getConnection();
    product = await db.fetchOne(
        "SELECT id, name, category, price, in_stock FROM products WHERE id = :id",
        { id }
    );

    if (product === null) {
        return res.status(404).json({ error: "Product not found" });
    }

    // 3. Store in cache for next time
    await cacheSet(cacheKey, product, 600);  // Cache for 10 minutes

    return res.json({ ...product, source: "database" });
});
```

```
curl http://localhost:7148/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. 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:

```
import { Router } from "tina4-nodejs";
import { Database } from "tina4-nodejs/orm";

Router.get("/api/categories", async (req, res) => {
    const db = Database.getConnection();

    // First call: executes the query (20ms)
    // Subsequent calls within 300 seconds: returns cached result (0.1ms)
    const categories = await db.fetchAll("SELECT * FROM categories ORDER BY name");

    return res.json({ categories });
});
```

The cache key is derived from the SQL query and its parameters. Different queries or different parameters produce different cache keys:

```
// These are cached separately:
await db.fetchAll("SELECT * FROM products WHERE category = :cat", { cat: "Electronics" });
await db.fetchAll("SELECT * FROM products WHERE category = :cat", { cat: "Fitness" });
```

When to Use DB Cache

- Read-heavy applications where the same queries run over and over
- Reference data that changes seldom (categories, countries, settings)
- Dashboard queries that aggregate large datasets

When Not to Use DB Cache

- Write-heavy applications where data changes on every request
- Queries with real-time requirements (inventory counts, live prices)
- Queries that must return the latest data at all times

Skipping Cache for Specific Queries

```
// Force a fresh query, bypassing the cache
const freshData = await db.fetchAll("SELECT * FROM products", { noCache: true });
```

---

8. Cache Invalidation Strategies

Cache invalidation is the hard problem. Stale cache serves outdated data. Premature invalidation throws away performance gains. Three strategies handle this.

Strategy 1: Time-Based Expiry (TTL)

The simplest strategy. Set a TTL and let the cache expire on its own:

```
await cacheSet("products:featured", featuredProducts, 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 fine for most storefronts.

Strategy 2: Event-Based Invalidation

Clear the cache when the underlying data changes:

```
import { Router, cacheSet, cacheDelete } from "tina4-nodejs";
import { Database } from "tina4-nodejs/orm";

Router.put("/api/products/{id:int}", async (req, res) => {
    const productId = req.params.id;
    const body = req.body;
    const db = Database.getConnection();

    await db.execute(
        "UPDATE products SET name = :name, price = :price WHERE id = :id",
        { name: body.name, price: body.price, id: productId }
    );

    // Invalidate the cache for this product
    await cacheDelete(`product:${productId}`);

    // Also invalidate any list caches that might include this product
    await cacheDelete("products:all");
    await cacheDelete("products:featured");

    const updated = await db.fetchOne("SELECT * FROM products WHERE id = :id", { id: productId });

    return res.json(updated);
});
```

This is the most accurate strategy -- the cache is fresh after every write. The downside: you must remember to invalidate every key that holds the affected data.

Strategy 3: Write-Through Cache

Update the cache at the same time as the database:

```
Router.put("/api/products/{id:int}", async (req, res) => {
    const productId = req.params.id;
```

```
    const body = req.body;
    const db = Database.getConnection();

    await db.execute(
        "UPDATE products SET name = :name, price = :price WHERE id = :id",
        { name: body.name, price: body.price, id: productId }
    );

    const updated = await db.fetchOne("SELECT * FROM products WHERE id = :id", { id: productId }

    // Write the new data to cache (instead of deleting)
    await cacheSet(`product:${productId}`, updated, 600);

    return res.json(updated);
});
```

This ensures the cache holds the latest data at all times. No cache miss after an update -- the next read comes from the already-warm cache.

Choosing a Strategy

Strategy	Best For	Tradeoff
TTL	Read-heavy, tolerates staleness	Stale data for up to TTL duration
Event-based	Consistency matters	Every write must invalidate
Write-through	High traffic, frequent updates	Every write does cache work

Combine them. Use TTL as a safety net (entries expire even if invalidation misses). Use event-based invalidation for immediate consistency on critical data.

---

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 fast
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 often, slight delay is acceptable
Shopping cart	0 (no cache)	Must reflect current state at all times

Dynamic TTL

Adjust TTL based on data characteristics:

```
import { Router, cacheGet, cacheSet } from "tina4-nodejs";

async function getCachedProduct(productId: number) {
    const cacheKey = `product:${productId}`;
    let product = await cacheGet(cacheKey);

    if (product !== null) {
        return product;
    }

    product = await fetchProductFromDatabase(productId);

    // Popular products: shorter TTL (more likely to change)
    // Inactive products: longer TTL (rarely change)
    const ttl = product.viewCount > 1000 ? 60 : 3600;

    await cacheSet(cacheKey, product, ttl);

    return product;
}
```

---

10. Cache Statistics

Monitor cache performance to verify that caching helps:

```
import { Router, cacheStats } from "tina4-nodejs";

Router.get("/api/cache/stats", async (req, res) => {
    const stats = await cacheStats();
    return res.json(stats);
});
```

```
curl http://localhost:7148/api/cache/stats
```

```
{
  "backend": "memory",
  "entries": 42,
  "hits": 15234,
  "misses": 891,
  "hit_rate": "94.5%",
  "memory_bytes": 524288,
  "oldest_entry_age": 3542
}
```

Hit rate above 90%: your caching strategy works. Below 80%: TTLs are too short, the cache is too small, or you are caching data that is not accessed often enough to benefit.

The dev dashboard at `/__dev` shows cache statistics too -- per-key hit counts and miss counts. You see which keys earn their keep.

---

11. Combining Cache Layers

For maximum performance, layer multiple cache strategies:

```
import { Router, cacheGet, cacheSet } from "tina4-nodejs";
import { Database } from "tina4-nodejs/orm";

Router.get("/api/catalog", async (req, res) => {
    const page = parseInt(req.query.page ?? "1", 10);
    const cacheKey = `catalog:page:${page}`;

    // Layer 1: Check application cache
    const cached = await cacheGet(cacheKey);

    if (cached !== null) {
        return res.json({ ...cached, cache: "application" });
    }

    // Layer 2: Database query (with DB-level caching if TINA4_DB_CACHE=true)
    const db = Database.getConnection();
    const limit = 20;
    const offset = (page - 1) * limit;

    const products = await db.fetchAll(
        `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`,
        { limit, offset }
    );

    const total = await db.fetchOne("SELECT COUNT(*) as count FROM products WHERE active = 1");

    const catalog = {
        products,
        page,
        total: total.count,
        pages: Math.ceil(total.count / limit),
        generated_at: new Date().toISOString()
    };

    // Store in application cache
    await cacheSet(cacheKey, catalog, 300);

    return res.json({ ...catalog, cache: "none" });
}, "ResponseCache:60");
```

This creates three cache layers:

• **ResponseCache** (60 seconds): The entire HTTP response is cached. No JavaScript 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 (`cacheGet`/`cacheSet`) 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:7148/api/store/products?category=Electronics&page=1"

# Second call -- cache hit, fast
curl "http://localhost:7148/api/store/products?category=Electronics&page=1"

# Different category -- cache miss
curl "http://localhost:7148/api/store/products?category=Fitness&page=1"

# Create a product -- should invalidate cache
curl -X POST http://localhost:7148/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:7148/api/store/products?category=Electronics&page=1"

# Check cache stats
curl http://localhost:7148/api/store/cache-stats
```

---

13. Solution

Create `src/routes/storeCached.ts`:

```
import { Router, cacheGet, cacheSet, cacheDelete, cacheStats } from "tina4-nodejs";
import { createHash } from "crypto";

function getProductStore() {
    return [
        { id: 1, name: "Wireless Keyboard", category: "Electronics", price: 79.99, inStock: true
        { id: 2, name: "Yoga Mat", category: "Fitness", price: 29.99, inStock: true },
        { id: 3, name: "Coffee Grinder", category: "Kitchen", price: 49.99, inStock: false },
        { id: 4, name: "Standing Desk", category: "Electronics", price: 549.99, inStock: true },
        { id: 5, name: "Running Shoes", category: "Fitness", price: 119.99, inStock: true },
        { id: 6, name: "Bluetooth Speaker", category: "Electronics", price: 39.99, inStock: true
        { id: 7, name: "Resistance Bands", category: "Fitness", price: 14.99, inStock: true },
        { id: 8, name: "French Press", category: "Kitchen", price: 34.99, inStock: true },
    ];
}


Router.get("/api/store/products", async (req, res) => {
    const category = req.query.category ?? null;
    const page = parseInt(req.query.page ?? "1", 10);
    const limit = parseInt(req.query.limit ?? "20", 10);

    // Build cache key from query parameters
    const keyData = JSON.stringify({ category, page, limit });
    const cacheKey = `store:products:${createHash("md5").update(keyData).digest("hex")}`;

    // Try cache first
    const cached = await cacheGet(cacheKey);

    if (cached !== null) {
        return res.json({ ...cached, source: "cache" });
    }

    // Simulate expensive database query
    await new Promise(resolve => setTimeout(resolve, 100));  // 100ms delay

    let products = getProductStore();

    // Filter by category
    if (category !== null) {
        products = products.filter(
            p => p.category.toLowerCase() === String(category).toLowerCase()
        );
    }

    const total = products.length;
    const offset = (page - 1) * limit;
    products = products.slice(offset, offset + limit);

    const result = {
        products,
        page,
```

```
        limit,
        total,
        pages: Math.ceil(total / limit),
        generated_at: new Date().toISOString()
    };

    // Cache for 5 minutes
    await cacheSet(cacheKey, result, 300);

    return res.json({ ...result, source: "database" });
});


Router.post("/api/store/products", async (req, res) => {
    const body = req.body;

    if (!body.name) {
        return res.status(400).json({ error: "Name is required" });
    }

    const product = {
        id: Math.floor(Math.random() * 9000) + 100,
        name: body.name,
        category: body.category ?? "General",
        price: parseFloat(body.price ?? "0"),
        inStock: true
    };

    // Invalidate all product list caches
    const categories = ["Electronics", "Fitness", "Kitchen", "General"];
    for (const cat of categories) {
        for (let p = 1; p <= 5; p++) {
            const keyData = JSON.stringify({ category: cat, page: p, limit: 20 });
            await cacheDelete(
                `store:products:${createHash("md5").update(keyData).digest("hex")}`
            );
        }
    }

    // Also invalidate the unfiltered list
    for (let p = 1; p <= 5; p++) {
        const keyData = JSON.stringify({ category: null, page: p, limit: 20 });
        await cacheDelete(
            `store:products:${createHash("md5").update(keyData).digest("hex")}`
        );
    }

    return res.status(201).json({
        message: "Product created",
        product,
        cache_invalidated: true
    });
});


Router.get("/api/store/cache-stats", async (req, res) => {
    return res.json(await cacheStats());
});
```

Expected output -- first call (cache miss):

```
curl "http://localhost:7148/api/store/products?category=Electronics&page=1"
```


```
{
  "products": [
    {"id": 1, "name": "Wireless Keyboard", "category": "Electronics", "price": 79.99, "inStock":
    {"id": 4, "name": "Standing Desk", "category": "Electronics", "price": 549.99, "inStock": tr
    {"id": 6, "name": "Bluetooth Speaker", "category": "Electronics", "price": 39.99, "inStock":
  ],
  "page": 1,
  "limit": 20,
  "total": 3,
  "pages": 1,
  "generated_at": "2026-03-22T14:30:00.000Z",
  "source": "database"
}
```


Expected output -- second call (cache hit):


```
{
  "products": [
    {"id": 1, "name": "Wireless Keyboard", "category": "Electronics", "price": 79.99, "inStock":
    {"id": 4, "name": "Standing Desk", "category": "Electronics", "price": 549.99, "inStock": tr
    {"id": 6, "name": "Bluetooth Speaker", "category": "Electronics", "price": 39.99, "inStock":
  ],
  "page": 1,
  "limit": 20,
  "total": 3,
  "pages": 1,
  "generated_at": "2026-03-22T14:30:00.000Z",
  "source": "cache"
}
```


Notice: same `generated_at`, but `source` changed from `"database"` to `"cache"`. The handler did not run.

---

14. Gotchas

1. Caching Authenticated Responses

Problem: User A's profile is served to User B because the response was cached.

Cause: `ResponseCache` caches by URL alone. If `/api/profile` returns different data per user but all requests hit the same URL, the first user's response is served to everyone.

Fix: Do not use `ResponseCache` on user-specific endpoints. Use the direct cache API with user-specific keys instead: `await cacheSet(`profile:${userId}`, data, 300)`.

2. Cache Stampede

Problem: When a popular cache key expires, hundreds of requests hit the database at the same moment (all experiencing a cache miss together).

Cause: All requests see the cache miss and all try to rebuild the cache on their own.

Fix: Use cache locking or "stale-while-revalidate." One request rebuilds the cache while others serve the stale value. Tina4's `ResponseCache` handles this by serving the expired response to concurrent requests while one request refreshes it.

3. Memory Cache Lost on Restart

Problem: After restarting the server, performance drops until the cache warms up.

Cause: Memory cache is lost when the process restarts. Every request is a cache miss until data is cached again.

Fix: For production, use Redis cache (`TINA4_CACHE_BACKEND=redis`). It persists across server restarts. You can also implement a cache warmup script that pre-populates the cache with data your application accesses most.

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: Invalidate (or update) the cache when you modify the underlying data. Use `await cacheDelete(product:${productId})` after an update, or use write-through caching with `cacheSet()` 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: ` products:${createHash("md5").update(JSON.stringify(params)).digest("hex")} `.

6. Serialization Overhead

Problem: Caching makes certain requests slower, not faster.

Cause: The cached object is large. Serializing and deserializing it takes more time than re-computing it.

Fix: Only cache data that is expensive to compute. If the original operation takes 5ms and cache serialization takes 10ms, caching is counterproductive. Profile before and after caching to verify the improvement.

7. Forgetting to Set TTL

Problem: Cache entries never expire and the server's memory grows until it crashes.

Cause: You called `await cacheSet("key", value)` without a TTL. The entry lives forever (or until the server restarts).

Fix: Set a TTL on every entry: `await cacheSet("key", value, 300)`. Even for data that "never changes," set a long TTL like 86400 (24 hours). This provides a safety net against stale data and prevents unbounded memory growth.

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.

Queues move slow work to a background process. The user gets a response in milliseconds. The work still happens -- just not during the request.

Tina4 has a built-in queue system. Works out of the box with a file-based backend. No Redis. No RabbitMQ. No external services.

---

2. Why Queues Matter

Without queues: 6530ms response time. With queues: 33ms. Same work done. Different timing.

Queues also deliver retry logic, rate limiting, fault isolation, and horizontal scaling.

---

3. File Queue (Default)

The file-based backend is the default. No configuration needed.

Creating a Queue and Pushing a Job

```
import { Queue } from "tina4-nodejs";

const queue = new Queue({ 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:

```
const queue = new Queue({ 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
```

```
// push(payload, delay, priority) -- delay 0, priority 10
queue.push({ to: "alice@example.com", subject: "Password Reset" }, 0, 10);
```

Queue Size

Check how many pending messages are in the queue:

```
const count = queue.size();
```

Pass the topic and a status string to count jobs in a specific state:

```
const failed = queue.size(undefined, "failed");
```

---

4. Pushing from Route Handlers

```
import { Router, Queue } from "tina4-nodejs";

const queue = new Queue({ topic: "emails" });

Router.post("/api/register", async (req, res) => {
    const body = req.body;
    const userId = 42;

    queue.push({
        user_id: userId,
        to: body.email,
        name: body.name,
        subject: "Welcome!"
    });

    return res.status(201).json({
        message: "Registration successful. Welcome email will arrive shortly.",
        user_id: userId
    });
});
```

---

5. Consuming Jobs

The `consume` method is a generator that yields jobs one at a time. Each job must be explicitly completed or failed:

```
import { Queue } from "tina4-nodejs";

const queue = new Queue({ topic: "emails" });

for (const job of queue.consume("emails")) {
    try {
        await sendEmail(job.payload.to, job.payload.subject, job.payload.body);
        job.complete();
    } catch (e) {
        job.fail(e.message);
    }
}
```

Retry with Delay

If a job fails but you want to retry it after a cooldown instead of marking it as failed:

```
for (const job of queue.consume("emails")) {
    try {
        await sendEmail(job.payload.to, job.payload.subject, job.payload.body);
        job.complete();
    } catch (e) {
        // Retry after 30 seconds instead of failing immediately
        job.retry(30);
    }
}
```

Manual Pop

For more control, pop a single message:

```
const job = queue.pop();

if (job !== null) {
    try {
        await sendEmail(job.payload.to, job.payload.subject);
        job.complete();
    } catch (e) {
        job.fail(e.message);
    }
}
```

---

6. Job Lifecycle

```
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(delaySeconds)` -- re-queue the job after a delay (in seconds). The job goes back to PENDING after the delay elapses.

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`, `retryFailed()` skips it.

Retrying Failed Jobs

```
// Retry a specific job by ID
queue.retry(jobId);

// Retry all failed jobs (skips those that exceeded max_retries)
queue.retryFailed();
```

Dead Letters

Jobs that have exceeded `max_retries` are dead letters. There is no magic dead letter queue -- you retrieve and handle them yourself:

```
const deadJobs = queue.deadLetters();

for (const job of deadJobs) {
    console.log(`Dead job: ${job.id}`);
    console.log(`  Payload: ${JSON.stringify(job.payload)}`);
    console.log(`  Error: ${job.error}`);
}
```

Purging Jobs

Remove jobs by status:

```
queue.purge("completed");
queue.purge("failed");
```

---

8. 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:

```
npm install mongodb
```

Your code stays identical. Same `queue.push()` and `queue.consume()` calls. The backend is an implementation detail.

---

9. Multiple Jobs from One Action

```
import { Router, Queue } from "tina4-nodejs";

const queue = new Queue({ topic: "emails" });

Router.post("/api/orders", async (req, res) => {
    const orderId = 101;

    queue.push({ order_id: orderId, to: req.body.email, subject: "Order Confirmation" });
    queue.produce("invoices", { order_id: orderId, format: "pdf" });
    queue.produce("inventory", { items: req.body.items });
    queue.produce("warehouse", { order_id: orderId, shipping_address: req.body.shipping_address

    return res.status(201).json({ message: "Order placed successfully", order_id: orderId });
});
```

---

10. 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:7148/api/emails/send \
  -H "Content-Type: application/json" \
  -d '{"to": "alice@example.com", "subject": "Welcome!", "body": "Thanks for signing up."}'

curl http://localhost:7148/api/emails/queue

curl http://localhost:7148/api/emails/dead

curl -X POST http://localhost:7148/api/emails/retry
```

---

11. Solution

Create `src/routes/emailQueue.ts`:

```
import { Router, Queue } from "tina4-nodejs";

const queue = new Queue({ topic: "emails" });

/**
 * @noauth
 */
Router.post("/api/emails/send", async (req, res) => {
    const body = req.body;

    const errors: string[] = [];
    if (!body.to) errors.push("'to' is required");
    if (!body.subject) errors.push("'subject' is required");
    if (!body.body) errors.push("'body' is required");

    if (errors.length > 0) {
        return res.status(400).json({ errors });
    }

    const messageId = queue.push({
        to: body.to,
        subject: body.subject,
        body: body.body
    });

    return res.status(201).json({
        message: "Email queued for sending",
        message_id: messageId
    });
});

Router.get("/api/emails/queue", async (req, res) => {
    const count = queue.size();
    return res.json({ pending: count });
});

Router.get("/api/emails/dead", async (req, res) => {
    const deadJobs = queue.deadLetters();
    const items = deadJobs.map((job) => ({
```

```
        id: job.id,
        payload: job.payload,
        error: job.error
    }));
    return res.json({ dead_letters: items, count: items.length });
});

Router.post("/api/emails/retry", async (req, res) => {
    queue.retryFailed();
    return res.json({ message: "Failed emails re-queued for retry" });
});
```

Create a separate consumer file `src/workers/emailWorker.ts`:

```
import { Queue } from "tina4-nodejs";

const queue = new Queue({ topic: "emails" });

for (const job of queue.consume("emails")) {
    const payload = job.payload;

    console.log(`Sending email to ${payload.to}...`);
    console.log(`  Subject: ${payload.subject}`);

    try {
        // Simulate sending (replace with real email logic)
        await new Promise((resolve) => setTimeout(resolve, 1000));

        // Simulate failure for a specific address
        if (payload.to === "bad@example.com") {
            throw new Error("SMTP connection refused");
        }

        console.log(`  Email sent to ${payload.to} successfully!`);
        job.complete();

    } catch (e) {
        console.log(`  Failed: ${e.message}`);
        job.fail(e.message);
    }
}
```

After the consumer has retried a job to `bad@example.com` three times, `queue.deadLetters()` returns that job. The `/api/emails/dead` endpoint shows it. You investigate, fix the address, and call `/api/emails/retry` to re-queue.

---

12. Gotchas

1. Always call complete or fail

Fix: Always call `job.complete()` on success and `job.fail(reason)` on failure. If you forget, the job stays reserved forever.

2. Worker not picking up messages

Fix: Make sure the consumer is running. Check that the topic in `queue.push()` matches the topic in `queue.consume()`.

3. Payload must be JSON-serializable

Fix: Only pass simple data types. Pass IDs, not full objects.

4. Dead letters pile up

Fix: Monitor `queue.deadLetters()` and set up alerts. Investigate root causes, then call `queue.retryFailed()` or `queue.purge("failed")`.

5. File backend for production

Fix: For multiple workers, switch to RabbitMQ, Kafka, or MongoDB via `TINA4_QUEUE_BACKEND`.

6. Consumer returns nothing

Fix: The consume pattern requires explicit `job.complete()` / `job.fail()` calls. Do not rely on return values.

Events System

1. Decouple Without Wiring Everything Together

Your user registers. You need to send a welcome email, create a default profile, log the signup, and update analytics. You could call all four functions from the signup handler. But now the signup handler knows about email, profiles, logging, and analytics. Change any of them and you're touching the handler.

Events decouple the action from the reaction. The signup handler emits `user.registered`. Anything that cares about that event subscribes to it. The handler knows nothing about what happens next.

Tina4 ships a built-in event bus. Zero external dependencies. Synchronous listeners, priority ordering, one-shot listeners, and cleanup.

---

2. Listening for Events

Register a listener with `Events.on()`:

```
import { Events } from "tina4-nodejs";

Events.on("user.registered", (payload) => {
    console.log(`New user: ${payload.name} (${payload.email})`);
});
```

The listener runs every time `user.registered` is emitted. The `payload` is whatever object was passed to `emit()`.

---

3. Emitting Events

Trigger all listeners for an event with `Events.emit()`:

```
import { Router, Events } from "tina4-nodejs";

Router.post("/api/register", async (req, res) => {
    const body = req.body;

    // ... create the user in the database ...
    const userId = 42;

    Events.emit("user.registered", {
        id: userId,
        name: body.name,
        email: body.email,
        registeredAt: new Date().toISOString()
    });

    return res.status(201).json({ message: "Registration complete", user_id: userId });
});
```

All registered listeners for `user.registered` run immediately, in priority order, before `emit()` returns.

---

4. One-Shot Listeners with once()

A listener registered with `Events.once()` fires exactly one time and then removes itself:

```
import { Events } from "tina4-nodejs";

// Fires the first time "server.ready" is emitted, then never again
Events.once("server.ready", () => {
    console.log("Server is up -- warming the cache");
    warmCache();
});
```

Useful for initialisation tasks, first-run setup, or integration tests that need to wait for a single event.

---

5. Priority

When multiple listeners are registered for the same event, priority controls the order they run. Higher priority runs first. Default priority is `0`.

```
import { Events } from "tina4-nodejs";

// Runs third (lowest priority)
Events.on("order.placed", (payload) => {
    console.log("Analytics: recording order");
}, 0);

// Runs first (highest priority)
Events.on("order.placed", (payload) => {
    console.log("Inventory: reserving stock");
}, 20);

// Runs second
Events.on("order.placed", (payload) => {
    console.log("Email: sending confirmation");
}, 10);

Events.emit("order.placed", { orderId: 101, total: 249.99 });
// Output:
// Inventory: reserving stock
// Email: sending confirmation
// Analytics: recording order
```

Priority is specified as the third argument to `on()`.

---

6. Removing Listeners with off()

Remove a specific listener by passing the same function reference used in `on()`:

```
import { Events } from "tina4-nodejs";

function onUserLogin(payload: { userId: number }) {
    console.log(`User ${payload.userId} logged in`);
}

// Register
Events.on("user.login", onUserLogin);

// Later, remove it
Events.off("user.login", onUserLogin);

// This emit does nothing -- the listener was removed
Events.emit("user.login", { userId: 99 });
```

Note: arrow functions assigned to variables work the same way as long as you pass the same reference.

---

7. Clearing All Listeners with clear()

Remove every listener for a given event (or all events):

```
import { Events } from "tina4-nodejs";

// Clear listeners for a specific event
Events.clear("user.registered");

// Clear all listeners for all events
Events.clear();
```

Useful in tests to reset state between test cases, or in long-running processes to tear down a subsystem cleanly.

---

8. Listing Active Listeners

Inspect registered listeners for debugging:

```
import { Events } from "tina4-nodejs";

const listeners = Events.listeners("order.placed");
console.log(`${listeners.length} listeners for order.placed`);
```

---

9. Organising Events in a Real Application

Keep all event subscriptions in one place so they are easy to audit. Create `src/events/index.ts`:

```
import { Events } from "tina4-nodejs";
import { sendWelcomeEmail } from "../services/email";
import { createDefaultProfile } from "../services/profile";
import { trackSignup } from "../services/analytics";
import { logUserEvent } from "../services/logging";

// user.registered
Events.on("user.registered", async (payload) => {
    await sendWelcomeEmail(payload.email, payload.name);
}, 10);

Events.on("user.registered", async (payload) => {
    await createDefaultProfile(payload.id);
}, 5);

Events.on("user.registered", (payload) => {
    trackSignup(payload.id, payload.registeredAt);
}, 1);

Events.on("user.registered", (payload) => {
    logUserEvent("signup", payload.id);
}, 0);
```

Then import this file in `src/index.ts` so listeners are registered before any requests arrive:

```
import "./events/index";
import { Tina4 } from "tina4-nodejs";

const app = new Tina4();
app.start();
```

---

10. Exercise: Order Processing Pipeline

Build an order processing pipeline using events.

Requirements

- Create a `POST /api/orders` endpoint that emits `order.placed` with order data
- Register three listeners for `order.placed`:
- Priority 20: validate and reserve inventory
- Priority 10: send confirmation email (simulated)
- Priority 0: record in analytics (simulated)
- Create a `GET /api/orders/events` endpoint that returns the current listener count for `order.placed`

Test with:

```
curl -X POST http://localhost:7148/api/orders \
  -H "Content-Type: application/json" \
  -d '{"items": [{"sku": "KB-001", "qty": 2}], "email": "alice@example.com"}'

curl http://localhost:7148/api/orders/events
```

---

11. Solution

Create `src/events/orders.ts`:

```
import { Events } from "tina4-nodejs";

Events.on("order.placed", (payload) => {
    console.log(`[Inventory] Reserving stock for order ${payload.orderId}`);
    for (const item of payload.items) {
        console.log(`  - SKU ${item.sku}: qty ${item.qty}`);
    }
}, 20);

Events.on("order.placed", (payload) => {
    console.log(`[Email] Sending confirmation to ${payload.email} for order ${payload.orderId}`)
}, 10);

Events.on("order.placed", (payload) => {
    console.log(`[Analytics] Recording order ${payload.orderId} — total $${payload.total}`);
}, 0);
```

Create `src/routes/orders.ts`:

```
import { Router, Events } from "tina4-nodejs";

let orderCounter = 1000;

Router.post("/api/orders", async (req, res) => {
    const body = req.body;

    if (!body.items || body.items.length === 0) {
        return res.status(400).json({ error: "Order must contain at least one item" });
    }

    const orderId = ++orderCounter;
    const total = body.items.reduce((sum: number, item: { qty: number; price?: number }) => {
        return sum + (item.qty * (item.price ?? 9.99));
    }, 0);

    Events.emit("order.placed", {
        orderId,
        items: body.items,
        email: body.email,
        total: parseFloat(total.toFixed(2)),
        placedAt: new Date().toISOString()
    });

    return res.status(201).json({
        message: "Order placed",
        order_id: orderId,
        total
    });
});

Router.get("/api/orders/events", async (req, res) => {
    const count = Events.listeners("order.placed").length;
    return res.json({ event: "order.placed", listener_count: count });
```

```
});
```

In `src/index.ts`, import the events file before starting the server:

```
import "./events/orders";
import { Tina4 } from "tina4-nodejs";

const app = new Tina4();
app.start();
```

---

12. Gotchas

1. Async listeners are not awaited

`Events.emit()` is synchronous. If a listener is `async`, the promise is not awaited. Long async work belongs in a queue, not an event listener.

Fix: For truly async fire-and-forget tasks, use `Queue`. For synchronous side effects, use events. If you must use async listeners, handle errors inside them -- unhandled rejections in event listeners are silent.

2. Same function reference required for off()

`Events.off("event", fn)` only works if `fn` is the exact same reference registered with `on()`.

Fix: Store named functions in variables. Arrow functions defined inline cannot be removed because each definition creates a new reference.

3. Listeners persist across requests

Events registered at module load time persist for the lifetime of the process. Registering listeners inside a route handler creates duplicates on every request.

Fix: Register listeners once, at startup, in a dedicated events file.

4. Forgetting to import the events file

Listeners registered in a file that is never imported do not exist.

Fix: Explicitly import your events file in `src/index.ts`. A missing import is a silent failure -- no error, no listeners, no side effects.

Localization

1. Your App Speaks One Language. Your Users Speak Many.

An e-commerce store. Half your traffic is German. Error messages in English. Product descriptions in English. Currency formatted American-style. Visitors leave.

Localization means serving content in the user's language and format. Tina4 provides an `I18n` class that loads JSON locale files, interpolates variables, falls back to a default locale when a key is missing, and switches locale per request.

---

2. Locale Files

Locale files are JSON files stored in a `locales/` directory in your project root. One file per language:

```
locales/
  en.json
  de.json
  fr.json
  es.json
```

`locales/en.json`:

```
{
  "welcome": "Welcome, {{name}}!",
  "errors": {
    "not_found": "The page you requested was not found.",
    "unauthorized": "You must be logged in to access this page.",
    "validation": "{{field}} is required."
  },
  "orders": {
    "placed": "Your order #{{orderId}} has been placed.",
    "total": "Total: {{currency}}{{amount}}"
  },
  "nav": {
    "home": "Home",
    "products": "Products",
    "cart": "Cart",
    "account": "Account"
  }
}
```

`locales/de.json`:

```
{
  "welcome": "Willkommen, {{name}}!",
  "errors": {
    "not_found": "Die angeforderte Seite wurde nicht gefunden.",
    "unauthorized": "Sie müssen angemeldet sein, um auf diese Seite zuzugreifen.",
    "validation": "{{field}} ist erforderlich."
  },
  "orders": {
    "placed": "Ihre Bestellung #{{orderId}} wurde aufgegeben.",
```

```
    "total": "Gesamt: {{currency}}{{amount}}"
  },
  "nav": {
    "home": "Startseite",
    "products": "Produkte",
    "cart": "Warenkorb",
    "account": "Konto"
  }
}
```

---

3. Creating an I18n Instance

```
import { I18n } from "tina4-nodejs";

// Loads all JSON files from ./locales/
// Default locale is "en"
const i18n = new I18n({
    localesPath: "./locales",
    defaultLocale: "en"
});
```

Option	Default	Description
`localesPath`	`"./locales"`	Directory containing locale JSON files
`defaultLocale`	`"en"`	Fallback locale when a key is missing
`fallbackLocale`	same as `defaultLocale`	Explicit fallback, if different from default

---

4. Translating Keys with t()

The `t()` method looks up a translation key in the current locale:

```
const i18n = new I18n({ localesPath: "./locales", defaultLocale: "en" });

i18n.setLocale("en");
console.log(i18n.t("nav.home"));         // "Home"
console.log(i18n.t("nav.products"));     // "Products"

i18n.setLocale("de");
console.log(i18n.t("nav.home"));         // "Startseite"
console.log(i18n.t("nav.products"));     // "Produkte"
```

Keys use dot notation to access nested values. `"errors.not_found"` reaches `errors -> not_found` in the JSON file.

---

5. Interpolation

Pass variables to replace `{{placeholder}}` tokens:

```
i18n.setLocale("en");
console.log(i18n.t("welcome", { name: "Alice" }));
// "Welcome, Alice!"

console.log(i18n.t("errors.validation", { field: "Email" }));
// "Email is required."

console.log(i18n.t("orders.placed", { orderId: "10042" }));
// "Your order #10042 has been placed."

i18n.setLocale("de");
console.log(i18n.t("welcome", { name: "Alice" }));
// "Willkommen, Alice!"
```

Any key not found in the variables object is left as-is in the output string.

---

6. Locale Switching Per Request

Detect the user's preferred locale from the `Accept-Language` header, a query parameter, or a session value:

```
import { Router, I18n } from "tina4-nodejs";

const i18n = new I18n({ localesPath: "./locales", defaultLocale: "en" });

Router.get("/api/greeting", async (req, res) => {
    // Priority: ?lang query param > Accept-Language header > default
    const lang = req.query.lang
        ?? (req.headers["accept-language"]?.split(",")[0]?.split("-")[0])
        ?? "en";

    i18n.setLocale(lang);

    return res.json({
        locale: lang,
        message: i18n.t("welcome", { name: req.query.name ?? "guest" })
    });
});
```

```
curl "http://localhost:7148/api/greeting?name=Alice&lang=de"
```

```
{
  "locale": "de",
  "message": "Willkommen, Alice!"
}
```

```
curl "http://localhost:7148/api/greeting?name=Alice&lang=en"
```

```
{
  "locale": "en",
  "message": "Welcome, Alice!"
}
```

---

7. Fallback Locale

When a key exists in the default locale but is missing from the requested locale, `I18n` falls back automatically rather than returning an empty string or throwing:

```
// locales/fr.json is missing the "orders" section
i18n.setLocale("fr");
console.log(i18n.t("orders.placed", { orderId: "202" }));
// Falls back to "en": "Your order #202 has been placed."
```

The fallback locale is configured once and applied globally. You never need to check whether a translation exists before calling `t()`.

---

8. Listing Available Locales

```
const locales = i18n.availableLocales();
console.log(locales);
// ["en", "de", "fr", "es"]
```

Use this to build a language picker in your UI or to validate the `lang` parameter on requests.

---

9. Using I18n in Templates

Pass the translation function to Frond templates:

```
import { Router, I18n } from "tina4-nodejs";

const i18n = new I18n({ localesPath: "./locales", defaultLocale: "en" });

Router.get("/store", async (req, res) => {
    const lang = req.query.lang ?? "en";
    i18n.setLocale(lang as string);

    return res.render("store.frond", {
        t: (key: string, vars?: Record<string, string>) => i18n.t(key, vars),
        locale: lang,
        locales: i18n.availableLocales()
    });
});
```

In `templates/store.frond`:

```
<nav>
  <a href="/">{{t("nav.home")}}</a>
  <a href="/products">{{t("nav.products")}}</a>
  <a href="/cart">{{t("nav.cart")}}</a>
</nav>
```

---

10. Exercise: Multilingual API Errors

Build an API that returns validation error messages in the user's language.

Requirements

- Create locale files for `en` and `de` with an `errors` section covering `required`, `too_short`, and `invalid_email`
- Create a `POST /api/contact` endpoint that validates a `name` (required, min 2 chars) and `email` (required, valid format) field
- Return error messages in the locale specified by the `lang` query parameter (default: `en`)

Test with:

```
# English errors
curl -X POST "http://localhost:7148/api/contact?lang=en" \
  -H "Content-Type: application/json" \
  -d '{"name": "A", "email": "not-an-email"}'

# German errors
curl -X POST "http://localhost:7148/api/contact?lang=de" \
  -H "Content-Type: application/json" \
  -d '{"name": "A", "email": "not-an-email"}'
```

---

11. Solution

`locales/en.json`:

```
{
  "errors": {
    "required": "{{field}} is required.",
    "too_short": "{{field}} must be at least {{min}} characters.",
    "invalid_email": "{{field}} must be a valid email address."
  },
  "contact": {
    "success": "Thank you, {{name}}. We will be in touch."
  }
}
```

`locales/de.json`:

```
{
  "errors": {
    "required": "{{field}} ist erforderlich.",
    "too_short": "{{field}} muss mindestens {{min}} Zeichen lang sein.",
    "invalid_email": "{{field}} muss eine gültige E-Mail-Adresse sein."
  },
  "contact": {
    "success": "Danke, {{name}}. Wir werden uns bei Ihnen melden."
  }
}
```

`src/routes/contact.ts`:

```
import { Router, I18n } from "tina4-nodejs";
```

```
const i18n = new I18n({ localesPath: "./locales", defaultLocale: "en" });

const EMAIL_RE = /^[^\s@]+@[^\s@]+\.[^\s@]+$/;

Router.post("/api/contact", async (req, res) => {
    const lang = (req.query.lang ?? "en") as string;
    i18n.setLocale(lang);

    const body = req.body;
    const errors: string[] = [];

    if (!body.name) {
        errors.push(i18n.t("errors.required", { field: "Name" }));
    } else if (String(body.name).length < 2) {
        errors.push(i18n.t("errors.too_short", { field: "Name", min: "2" }));
    }

    if (!body.email) {
        errors.push(i18n.t("errors.required", { field: "Email" }));
    } else if (!EMAIL_RE.test(String(body.email))) {
        errors.push(i18n.t("errors.invalid_email", { field: "Email" }));
    }

    if (errors.length > 0) {
        return res.status(422).json({ locale: lang, errors });
    }

    return res.json({
        locale: lang,
        message: i18n.t("contact.success", { name: body.name })
    });
});
```

English error response:

```
{
  "locale": "en",
  "errors": [
    "Name must be at least 2 characters.",
    "Email must be a valid email address."
  ]
}
```

German error response:

```
{
  "locale": "de",
  "errors": [
    "Name muss mindestens 2 Zeichen lang sein.",
    "Email muss eine gültige E-Mail-Adresse sein."
  ]
}
```

---

12. Gotchas

1. Locale files must be valid JSON

A trailing comma or missing quote breaks the entire locale file silently. `I18n` will log a parse error and skip the file, falling back to the default locale for all keys.

Fix: Use a JSON linter or your editor's JSON validation before deploying. CI pipelines should run `JSON.parse()` against all locale files.

2. Missing keys return the key string

If a key does not exist in any locale, `t()` returns the key itself (e.g., `"orders.shipped"`). This makes missing translations visible in production.

Fix: Maintain the English locale as the complete reference. Other locales can be partial -- they fall back to English. Audit periodically with a script that compares all locale files against the English baseline.

3. setLocale is not thread-safe for concurrent requests

If multiple concurrent requests call `i18n.setLocale()` on the same instance, they interfere with each other.

Fix: Create a new `I18n` instance per request, or use `i18n.translate(key, vars, locale)` which accepts the locale as a parameter without mutating instance state.

4. Placeholder names are case-sensitive

`{{Name}}` and `{{name}}` are different placeholders. A mismatch silently leaves the placeholder unreplaced.

Fix: Use consistent lowercase placeholder names in all locale files.

Structured Logging

1. console.log Is Not a Logging Strategy

`console.log("something happened")` tells you nothing. No timestamp. No severity. No context. No way to filter by level in production. No way to ship to a log aggregator. When an incident happens at 2am, you need to know exactly what happened and when.

Structured logging emits machine-readable JSON. Every entry has a timestamp, log level, message, and optional context fields. You can filter, search, and aggregate without grep.

Tina4 provides a `Log` singleton with four severity levels. Zero configuration required. Control verbosity with one environment variable.

---

2. The Four Log Levels

```
import { Log } from "tina4-nodejs";

Log.debug("Cache lookup", { key: "product:42", hit: false });
Log.info("User registered", { userId: 99, email: "alice@example.com" });
Log.warn("Rate limit approaching", { ip: "203.0.113.5", requests: 95, limit: 100 });
Log.error("Payment failed", { orderId: 1042, reason: "Card declined" });
```

Each call emits a structured log line:

```
{"timestamp":"2026-04-02T08:12:01.234Z","level":"DEBUG","message":"Cache lookup","key":"product:
{"timestamp":"2026-04-02T08:12:01.235Z","level":"INFO","message":"User registered","userId":99,"
{"timestamp":"2026-04-02T08:12:01.236Z","level":"WARN","message":"Rate limit approaching","ip":"
{"timestamp":"2026-04-02T08:12:01.237Z","level":"ERROR","message":"Payment failed","orderId":104
```

Level	Use for
`debug`	Detailed diagnostic info, cache hits/misses, query plans
`info`	Normal application events: logins, signups, orders placed
`warn`	Unexpected but recoverable situations: retries, slow queries
`error`	Failures that need attention: payment errors, crashed workers

---

3. Controlling Verbosity with TINA4_LOG_LEVEL

Set the minimum level in `.env`:

```
TINA4_LOG_LEVEL=info
```

Only entries at or above the configured level are emitted:

	debug	info	warn	error
`debug`	shown	shown	shown	shown
`info`	silent	shown	shown	shown
`warn`	silent	silent	shown	shown
`error`	silent	silent	silent	shown

In development, use `debug`. In production, use `info` or `warn` to reduce log volume.

```
# .env.development
TINA4_LOG_LEVEL=debug

# .env.production
TINA4_LOG_LEVEL=warn
```

---

4. Adding Context Fields

The second argument to any log method is a plain object. Its fields are merged into the log entry:

```
import { Log } from "tina4-nodejs";

Log.info("HTTP request", {
    method: "POST",
    path: "/api/orders",
    status: 201,
    duration_ms: 47,
    user_id: 15
});
```

```
{"timestamp":"2026-04-02T08:15:00.000Z","level":"INFO","message":"HTTP request","method":"POST",
```

Any JSON-serializable value is valid: strings, numbers, booleans, arrays, nested objects.

---

5. Logging Errors

Pass an `Error` object alongside context:

```
import { Log } from "tina4-nodejs";

try {
    await processPayment(orderId, amount);
} catch (err) {
    Log.error("Payment processing failed", {
        orderId,
        amount,
        error: err instanceof Error ? err.message : String(err),
        stack: err instanceof Error ? err.stack : undefined
```

```
    });
}
```

```
{
  "timestamp": "2026-04-02T08:16:00.000Z",
  "level": "ERROR",
  "message": "Payment processing failed",
  "orderId": 1042,
  "amount": 249.99,
  "error": "Connection timeout after 5000ms",
  "stack": "Error: Connection timeout...\n    at PaymentGateway.charge ..."
}
```

---

6. Request-Scoped Logging

Add a request ID to every log entry in a request handler so you can trace all log lines from a single request:

```
import { Router, Log } from "tina4-nodejs";
import { randomUUID } from "crypto";

Router.post("/api/checkout", async (req, res) => {
    const requestId = randomUUID();
    const start = Date.now();

    Log.info("Checkout started", { requestId, user: req.user?.id });

    try {
        const orderId = await placeOrder(req.body, requestId);

        Log.info("Order placed", {
            requestId,
            orderId,
            duration_ms: Date.now() - start
        });

        return res.status(201).json({ order_id: orderId });

    } catch (err) {
        Log.error("Checkout failed", {
            requestId,
            error: err instanceof Error ? err.message : String(err),
            duration_ms: Date.now() - start
        });

        return res.status(500).json({ error: "Checkout failed" });
    }
});
```

Search your log aggregator for `requestId` to see every log line from that request, in order, with timings.

---

7. Performance Logging

Log slow operations to identify bottlenecks:

```
import { Log } from "tina4-nodejs";
import { Database } from "tina4-nodejs/orm";

async function fetchDashboardData(userId: number) {
    const t0 = Date.now();
    const db = Database.getConnection();

    const data = await db.fetchAll(
        `SELECT o.id, o.total, o.status, COUNT(oi.id) as item_count
         FROM orders o
         JOIN order_items oi ON oi.order_id = o.id
         WHERE o.user_id = :userId
         GROUP BY o.id
         ORDER BY o.created_at DESC
         LIMIT 10`,
        { userId }
    );

    const duration = Date.now() - t0;

    if (duration > 500) {
        Log.warn("Slow dashboard query", { userId, duration_ms: duration, rows: data.length });
    } else {
        Log.debug("Dashboard query", { userId, duration_ms: duration, rows: data.length });
    }

    return data;
}
```

---

8. Exercise: Add Logging to an Existing API

Take the product listing endpoint from Chapter 11 and add structured logging at every meaningful point.

Requirements

- Log `info` when a request is received, including the route and query parameters
- Log `debug` for cache hits and misses with the cache key
- Log `warn` when a query takes longer than 200ms
- Log `error` when an exception is caught, with the error message and stack

Expected log output (debug level):

```
{"timestamp":"...","level":"INFO","message":"Products request","category":"Electronics","page":1
{"timestamp":"...","level":"DEBUG","message":"Cache miss","key":"store:products:a3f2..."}
{"timestamp":"...","level":"WARN","message":"Slow product query","duration_ms":312,"rows":3}
{"timestamp":"...","level":"INFO","message":"Products served","source":"database","count":3,"dur
```

---

9. Solution

```
import { Router, Log, cacheGet, cacheSet } from "tina4-nodejs";
import { createHash } from "crypto";

const PRODUCTS = [
    { id: 1, name: "Wireless Keyboard", category: "Electronics", price: 79.99 },
    { id: 2, name: "Yoga Mat", category: "Fitness", price: 29.99 },
    { id: 3, name: "Coffee Grinder", category: "Kitchen", price: 49.99 },
    { id: 4, name: "Standing Desk", category: "Electronics", price: 549.99 },
];

Router.get("/api/products/logged", async (req, res) => {
    const category = req.query.category ?? null;
    const page = parseInt(req.query.page ?? "1", 10);
    const start = Date.now();

    Log.info("Products request", { category, page });

    const keyData = JSON.stringify({ category, page });
    const cacheKey = `products:${createHash("md5").update(keyData).digest("hex")}`;

    try {
        const cached = await cacheGet(cacheKey);

        if (cached !== null) {
            Log.debug("Cache hit", { key: cacheKey });
            return res.json({ ...cached, source: "cache" });
        }

        Log.debug("Cache miss", { key: cacheKey });

        // Simulate database work
        const t0 = Date.now();
        await new Promise(resolve => setTimeout(resolve, 50));
        let products = PRODUCTS;

        if (category) {
            products = products.filter(
                p => p.category.toLowerCase() === String(category).toLowerCase()
            );
        }

        const queryDuration = Date.now() - t0;

        if (queryDuration > 200) {
            Log.warn("Slow product query", { duration_ms: queryDuration, rows: products.length }
        }

        const result = { products, page, total: products.length };
        await cacheSet(cacheKey, result, 300);

        Log.info("Products served", {
            source: "database",
            count: products.length,
            duration_ms: Date.now() - start
        });

        return res.json({ ...result, source: "database" });
```

```
    } catch (err) {
        Log.error("Products endpoint failed", {
            error: err instanceof Error ? err.message : String(err),
            stack: err instanceof Error ? err.stack : undefined,
            duration_ms: Date.now() - start
        });
        return res.status(500).json({ error: "Internal server error" });
    }
});
```

---

10. Gotchas

1. Logging sensitive data

`Log.info("Login", { email, password })` ships the password to your log aggregator.

Fix: Never log passwords, tokens, credit card numbers, or PII. Log user IDs and request IDs instead. Before shipping a log call to production, check every field.

2. Logging in hot paths adds latency

Calling `Log.debug()` on every database row in a loop adds up.

Fix: Log aggregates, not individual items. `Log.debug("Fetched rows", { count: rows.length })` is better than logging each row.

3. Circular references in context objects

`Log.info("Data", { obj })` throws if `obj` contains circular references, because JSON serialization fails.

Fix: Pass primitive values and simple objects. Use `JSON.stringify` with a replacer to handle circular references if you must log complex objects.

4. TINA4*LOG*LEVEL defaults to info

In development, you might expect to see debug output and see nothing.

Fix: Set `TINA4_LOG_LEVEL=debug` in your `.env` file. Check the value with `Log.debug("Log level check", {})` -- if it appears, debug logging is active.

Email with Messenger

1. Every App Sends Email

Signup confirmations. Password resets. Weekly digests. Invoices with PDF attachments. Every application needs email. Nobody enjoys building it.

SMTP configuration. Plain text fallbacks. Attachment encoding. Connection timeouts. The details pile up fast. Tina4's `Messenger` class absorbs all of it. Configure through `.env`. Create an instance. Send. In development mode, Messenger intercepts every outgoing email and displays it in the dev dashboard -- no real SMTP server required.

---

2. Messenger Configuration via .env

All email configuration lives in `.env`:

```
TINA4_MAIL_HOST=smtp.example.com
TINA4_MAIL_PORT=587
TINA4_MAIL_USERNAME=your-email@example.com
TINA4_MAIL_PASSWORD=your-app-password
TINA4_MAIL_ENCRYPTION=tls
TINA4_MAIL_FROM=noreply@example.com
TINA4_MAIL_FROM_NAME=My Store
```

Variable	Description	Common Values
`TINA4_MAIL_HOST`	SMTP server hostname	`smtp.gmail.com`, `smtp.mailgun.org`, `smtp.sendgrid.net`
`TINA4_MAIL_PORT`	SMTP port	`587` (TLS), `465` (SSL), `25` (unencrypted)
`TINA4_MAIL_USERNAME`	Login username	Usually your email address
`TINA4_MAIL_PASSWORD`	Login password or app-specific password	App passwords for Gmail
`TINA4_MAIL_ENCRYPTION`	Encryption method	`tls` (recommended), `ssl`, `none`
`TINA4_MAIL_FROM`	Default "From" address	`noreply@yourdomain.com`
`TINA4_MAIL_FROM_NAME`	Default "From" display name	`My Store`, `Acme Corp`

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:

```
import { Messenger } from "tina4-nodejs";

// Uses .env defaults
const mailer = new Messenger();

// Override specific settings
const marketingMailer = new Messenger({
    host: "smtp.mailgun.org",
    port: 587,
    username: "marketing@mg.yourdomain.com",
    password: "marketing-smtp-password",
    encryption: "tls",
    fromAddress: "newsletter@yourdomain.com",
    fromName: "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:

```
import { Router, Messenger } from "tina4-nodejs";

Router.post("/api/contact", async (req, res) => {
```

```
    const body = req.body;
    const mailer = new Messenger();

    const result = await mailer.send({
        to: body.email,
        subject: "Contact Form Submission",
        body: `Name: ${body.name}\nEmail: ${body.email}\nMessage:\n${body.message}`
    });

    if (result.success) {
        return res.json({ message: "Email sent successfully" });
    }
    return res.status(500).json({ error: "Failed to send email", details: result.error });
});
```

```
curl -X POST http://localhost:7148/api/contact \
  -H "Content-Type: application/json" \
  -d '{"name": "Alice", "email": "alice@example.com", "message": "I love your products!"}'
```

```
{"message":"Email sent successfully"}
```

The `send()` method returns an object with three keys: `success` (boolean), `error` (string or null), and `messageId` (string or null).

The send() Method Signature

```
await mailer.send({
    to,                 // Recipient(s) -- string or array of strings
    subject,            // Email subject line
    body,               // Email body (plain text or HTML)
    html,               // If true, body is treated as HTML (default: false)
    text,               // Plain text alternative (when body is HTML)
    cc,                 // CC recipient(s) -- string or array
    bcc,                // BCC recipient(s) -- string or array
    replyTo,            // Reply-To address
    attachments,        // Array of file paths or objects
    headers             // Additional email headers (object)
});
```

---

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:

```
import { Messenger } from "tina4-nodejs";

const mailer = new Messenger();

const htmlBody = `
<html>
<body style="font-family: Arial, sans-serif; max-width: 600px; margin: 0 auto;">
    <div style="background: #1a1a2e; color: white; padding: 20px; text-align: center;">
        <h1 style="margin: 0;">Welcome to My Store!</h1>
    </div>
    <div style="padding: 20px;">
        <p>Hi Alice,</p>
        <p>Thank you for creating your account. We are excited to have you!</p>
```

```
        <p>Here is what you can do next:</p>
        <ul>
            <li>Browse our <a href="https://mystore.com/products">product catalog</a></li>
            <li>Set up your <a href="https://mystore.com/profile">profile</a></li>
            <li>Check out our <a href="https://mystore.com/deals">current deals</a></li>
        </ul>
        <p>If you have any questions, reply to this email and we will get back to you within 24
        <p>Cheers,<br>The My Store Team</p>
    </div>
    <div style="background: #f5f5f5; padding: 12px; text-align: center; font-size: 12px; color:
        <p>You received this because you signed up at mystore.com</p>
    </div>
</body>
</html>
`;

const textBody = [
    "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",
    "",
    "If you have any questions, reply to this email.",
    "",
    "Cheers,",
    "The My Store Team"
].join("\n");

const result = await mailer.send({
    to: "alice@example.com",
    subject: "Welcome to My Store!",
    body: htmlBody,
    html: true,
    text: textBody
});
```

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:

```
import { Messenger } from "tina4-nodejs";

const mailer = new Messenger();

const result = await mailer.send({
    to: "accounting@example.com",
    subject: "Monthly Invoice #1042",
    body: "<h2>Invoice #1042</h2><p>Please find the invoice attached.</p>",
```

```
    html: true,
    attachments: [
        "/path/to/invoices/invoice-1042.pdf",
        "/path/to/reports/monthly-summary.csv"
    ]
});
```

Messenger reads each file, determines its MIME type, and encodes it for email transmission. Each entry can be a file path string or an object for more control.

Attachments with Custom Names and Binary Content

```
const result = await mailer.send({
    to: "alice@example.com",
    subject: "Your Export",
    body: "<p>Here is your data export.</p>",
    html: true,
    attachments: [
        {
            filename: "my-store-export.csv",
            content: csvBuffer,
            mime: "text/csv"
        }
    ]
});
```

The object format accepts `filename` (display name), `content` (Buffer or raw bytes), and `mime` (MIME type string). The recipient sees the file named `my-store-export.csv` regardless of how it was generated.

---

7. CC, BCC, and Reply-To

```
import { Messenger } from "tina4-nodejs";

const mailer = new Messenger();

const result = await mailer.send({
    to: "alice@example.com",
    subject: "Team Meeting Notes",
    body: "<p>Here are the notes from today's meeting.</p>",
    html: true,
    cc: ["bob@example.com", "charlie@example.com"],
    bcc: ["manager@example.com"],
    replyTo: "alice@example.com"
});
```

- **cc**: Array of email addresses to carbon copy. All recipients see CC addresses.
- **bcc**: Array of email addresses to blind carbon copy. Recipients cannot see BCC addresses.
- **replyTo**: 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 two methods for adding headers. The `addHeader()` method sets a default header on all emails sent by that instance. The `headers` property on `send()` sets headers for a single email.

Default Headers on an Instance

```
import { Messenger } from "tina4-nodejs";

const mailer = new Messenger();
mailer.addHeader("X-App-Name", "My Store");
mailer.addHeader("X-Environment", "production");

// Every email from this instance carries both headers
await mailer.send({ to: "alice@example.com", subject: "Test", body: "Hello" });
```

Per-Email Headers

```
const result = await mailer.send({
    to: "customer@example.com",
    subject: "Your Support Ticket #123",
    body: "We are looking into your issue.",
    replyTo: "support@mystore.com",
    headers: {
        "X-Ticket-Id": "123",
        "X-Priority": "1",
        "X-Mailer": "Tina4 Messenger"
    }
});
```

Per-email headers merge with default headers. If both define the same key, the per-email value wins.

Custom headers serve several purposes. Tracking headers like `X-Ticket-Id` let you correlate emails with support tickets. Priority headers influence some email clients' display. Bulk-sending headers like `Precedence: bulk` help mail servers classify newsletters.

Header	Purpose
`X-Priority`	Email priority (1=high, 3=normal, 5=low)
`X-Mailer`	Identifies the sending application
`List-Unsubscribe`	One-click unsubscribe link (required for bulk email)
`X-Entity-Ref-ID`	Prevents threading in some email clients
`References`	Controls email threading

---

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
TINA4_MAIL_IMAP_USERNAME=support@example.com
TINA4_MAIL_IMAP_PASSWORD=your-imap-password
TINA4_MAIL_IMAP_ENCRYPTION=ssl
```

You can also override the IMAP host and port in the constructor:

```
const mailer = new Messenger({ imapHost: "imap.gmail.com", imapPort: 993 });
```

Listing Inbox Messages

The `getInbox()` method fetches message headers from the mailbox:

```
import { Router, Messenger } from "tina4-nodejs";

Router.get("/api/inbox", async (req, res) => {
    const mailer = new Messenger();

    const emails = await mailer.getInbox({ limit: 20, offset: 0 });

    const messages = emails.map(email => ({
        uid: email.uid,
        from: email.from,
        subject: email.subject,
        date: email.date,
        snippet: email.snippet,
        seen: email.seen
    }));

    return res.json({ messages, count: messages.length });
});
```

```
curl http://localhost:7148/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 `getInbox()` 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

```
Router.get("/api/inbox/:uid", async (req, res) => {
    const mailer = new Messenger();
    const uid = req.params.uid;

    const email = await mailer.read(uid, { markRead: true });

    if (!email) {
        return res.status(404).json({ error: "Email not found" });
    }

    return res.json({
        uid: email.uid,
        from: email.from,
        to: email.to,
        cc: email.cc,
        subject: email.subject,
        date: email.date,
        bodyHtml: email.bodyHtml,
        bodyText: email.bodyText,
        attachments: (email.attachments ?? []).map(a => ({
            filename: a.filename,
            size: a.size,
            contentType: a.contentType
        }))
    });
});
```

The `read()` method fetches the full message including body and attachments. Pass `{ markRead: false }` to leave the message unread.

Searching Messages

```
Router.get("/api/inbox/search", async (req, res) => {
    const mailer = new Messenger();

    const results = await mailer.search({
        subject: req.query.q as string,
        sender: req.query.from as string,
        unseenOnly: req.query.unread === "true",
        limit: 20
    });

    return res.json({ messages: results, count: results.length });
});
```

The `search()` method accepts `subject`, `sender`, `since` (date string), `before`, and `unseenOnly` as filters. All filters combine with AND logic.

Other IMAP Operations

```
const mailer = new Messenger();

// Count unread messages
const count = await mailer.getUnreadCount();

// Mark a message as read
await mailer.markRead("12345");

// Mark a message as unread
await mailer.markUnread("12345");

// Delete a message
await mailer.deleteEmail("12345");

// List all mailbox folders
const folders = await mailer.getFolders();
// ["INBOX", "Sent", "Drafts", "Trash", "Spam"]
```

Read from a Specific Folder

```
const sent = await mailer.getInbox({ folder: "Sent", limit: 10 });
```

Testing IMAP Connectivity

```
const mailer = new Messenger();
const result = await mailer.testImapConnection();
if (result.success) {
    console.log("IMAP connection works");
} else {
    console.log(`IMAP failed: ${result.error}`);
}
```

---

10. Dev Mode: Email Interception

When `TINA4_DEBUG=true`, Tina4 intercepts all outgoing emails and stores them locally. No real recipients receive anything. No accidental emails during development.

Navigate to `/__dev` and look for the "Mail" section. You see:

- Every email "sent" during the current session
- The To, CC, and BCC addresses
- The subject and body (both HTML and plain text)
- Attachments (viewable inline)
- The timestamp

This is invaluable for testing email functionality without configuring a real SMTP server. Use `createMessenger()` instead of `new Messenger()` to get automatic dev-mode interception:

```
import { createMessenger } from "tina4-nodejs";

const mailer = createMessenger();
// In dev mode: captures locally
```

```
// In production: sends via SMTP
```

Disabling Interception

If you need to test real email delivery during development, override the interception:

```
TINA4_MAILBOX_DIR=false
```

With this set, emails reach real recipients even when `TINA4_DEBUG=true`. Use with caution -- you do not want to accidentally email your entire user base from a dev machine.

---

11. Using Templates for Email Content

Hardcoded HTML in string literals is ugly and hard to maintain. Templates fix this.

Create `src/templates/emails/welcome.html`:

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8">
</head>
<body style="font-family: Arial, sans-serif; max-width: 600px; margin: 0 auto; background: #f5f5
    <div style="background: #1a1a2e; color: white; padding: 24px; text-align: center; border-rad
        <h1 style="margin: 0; font-size: 24px;">Welcome, {{ name }}!</h1>
    </div>
    <div style="background: white; padding: 24px; border-radius: 0 0 8px 8px;">
        <p>Hi {{ name }},</p>
        <p>Your account has been created successfully. Here are your details:</p>

        <table style="width: 100%; border-collapse: collapse; margin: 16px 0;">
            <tr>
                <td style="padding: 8px; border-bottom: 1px solid #eee; font-weight: bold;">Emai
                <td style="padding: 8px; border-bottom: 1px solid #eee;">{{ email }}</td>
            </tr>
            <tr>
                <td style="padding: 8px; border-bottom: 1px solid #eee; font-weight: bold;">Acco
                <td style="padding: 8px; border-bottom: 1px solid #eee;">#{{ user_id }}</td>
            </tr>
            <tr>
                <td style="padding: 8px; border-bottom: 1px solid #eee; font-weight: bold;">Sign
                <td style="padding: 8px; border-bottom: 1px solid #eee;">{{ signed_up_at }}</td>
            </tr>
        </table>

        <p>Get started by exploring:</p>
        <ul>
            <li><a href="{{ base_url }}/products" style="color: #1a1a2e;">Our product catalog</a
            <li><a href="{{ base_url }}/profile" style="color: #1a1a2e;">Your profile settings</
        </ul>

        {% if promo_code %}
            <div style="background: #d4edda; padding: 16px; border-radius: 4px; margin: 16px 0;"
                <strong>Special offer!</strong> Use code <code>{{ promo_code }}</code> for 10% o
            </div>
        {% endif %}
```

```
        <p>Cheers,<br>The {{ app_name }} Team</p>
    </div>

    <div style="text-align: center; padding: 12px; color: #888; font-size: 12px;">
        <p>You received this because you signed up at {{ app_name }}.</p>
        <p><a href="{{ base_url }}/unsubscribe?token={{ unsubscribe_token }}" style="color: #888
    </div>
</body>
</html>
```

Rendering and Sending

```
import { Router, Messenger } from "tina4-nodejs";
import { Frond } from "tina4-nodejs/frond";
import crypto from "crypto";

Router.post("/api/register", async (req, res) => {
    const body = req.body;

    // Create user (database logic)
    const userId = 42;

    // Render the email template
    const emailData = {
        name: body.name,
        email: body.email,
        user_id: userId,
        signed_up_at: new Date().toLocaleDateString("en-US", { year: "numeric", month: "long", d
        base_url: process.env.APP_URL ?? "http://localhost:7148",
        app_name: "My Store",
        promo_code: "WELCOME10",
        unsubscribe_token: crypto.randomBytes(16).toString("hex")
    };

    const htmlBody = await Frond.render("emails/welcome.html", emailData);

    // Send the email
    const mailer = new Messenger();
    const result = await mailer.send({
        to: body.email,
        subject: `Welcome to My Store, ${body.name}!`,
        body: htmlBody,
        html: true,
        text: `Hi ${body.name},\n\nWelcome to My Store! Your account (#${userId}) has been creat
    });

    return res.status(201).json({
        message: "Registration successful",
        email_sent: result.success,
        user_id: userId
    });
});
```

```
curl -X POST http://localhost:7148/api/register \
  -H "Content-Type: application/json" \
  -d '{"name": "Alice", "email": "alice@example.com", "password": "securePass123"}'
```

```
{
  "message": "Registration successful",
  "email_sent": true,
```

```
  "user_id": 42
}
```

With `TINA4_DEBUG=true`, the email appears in the dev dashboard instead of reaching a real inbox. Inspect the rendered HTML, check that template variables substituted, and verify the layout.

---

12. Sending Email via Queues

In production, never send email inside a route handler. The SMTP call blocks the response. Use the queue system from Chapter 12:

```
import { Router, Queue, Messenger } from "tina4-nodejs";
import { Frond } from "tina4-nodejs/frond";

const emailQueue = new Queue({ topic: "emails" });

// In the route handler, just queue the email
Router.post("/api/register", async (req, res) => {
    const userId = 42; // Simulated

    emailQueue.push({
        template: "emails/welcome.html",
        to: req.body.email,
        subject: `Welcome to My Store, ${req.body.name}!`,
        data: {
            name: req.body.name,
            email: req.body.email,
            user_id: userId,
            signed_up_at: "March 22, 2026",
            base_url: "http://localhost:7148",
            app_name: "My Store",
            promo_code: "WELCOME10"
        }
    });

    return res.status(201).json({ message: "Registration successful", user_id: userId });
});

// The consumer sends the actual email
emailQueue.process(async (job) => {
    const { template, to, subject, data } = job.payload as any;

    const frond = new Frond("src/templates");
    const htmlBody = frond.render(template, data);

    const mailer = new Messenger();
    const result = await mailer.send({ to, subject, body: htmlBody, html: true });

    if (!result.success) {
        console.log(`Email failed: ${result.message}`);
        job.fail(result.message);
        return;
    }

    console.log(`Email sent to ${to}`);
```

```
    job.complete();
});
```

The route handler returns in under 50 milliseconds. The queue worker sends the email on its own timeline. If the SMTP server is down, retries happen automatically.

---

13. Exercise: Build a Contact Form with Email Notification

Build a contact form that sends an email notification when submitted.

Requirements

- Create a `GET /contact` page that renders a contact form with fields: name, email, subject, and message
- Create a `POST /contact` endpoint that:
- Validates all fields are present
- Sends an email notification to the site admin (`admin@example.com`)
- The email should include all form fields, formatted in HTML
- Shows a flash message on success
- Redirects back to the contact page
- Create an email template at `src/templates/emails/contact-notification.html` that formats the contact submission

Test with:

```
# View the form
curl http://localhost:7148/contact

# Submit the form
curl -X POST http://localhost:7148/contact \
  -H "Content-Type: application/json" \
  -d '{"name": "Bob", "email": "bob@example.com", "subject": "Product inquiry", "message": "Do y

# Check the dev dashboard for the intercepted email
# Navigate to http://localhost:7148/__dev
```

---

14. Solution

Create `src/templates/emails/contact-notification.html`:

```
<!DOCTYPE html>
<html>
<head><meta charset="UTF-8"></head>
<body style="font-family: Arial, sans-serif; max-width: 600px; margin: 0 auto;">
    <div style="background: #1a1a2e; color: white; padding: 16px; text-align: center;">
        <h2 style="margin: 0;">New Contact Form Submission</h2>
    </div>
    <div style="padding: 20px; background: white;">
```

```
        <table style="width: 100%; border-collapse: collapse;">
            <tr>
                <td style="padding: 10px; border-bottom: 1px solid #eee; font-weight: bold; widt
                <td style="padding: 10px; border-bottom: 1px solid #eee;">{{ name }} ({{ email }
            </tr>
            <tr>
                <td style="padding: 10px; border-bottom: 1px solid #eee; font-weight: bold;">Sub
                <td style="padding: 10px; border-bottom: 1px solid #eee;">{{ subject }}</td>
            </tr>
            <tr>
                <td style="padding: 10px; border-bottom: 1px solid #eee; font-weight: bold;">Dat
                <td style="padding: 10px; border-bottom: 1px solid #eee;">{{ submitted_at }}</td
            </tr>
        </table>

        <div style="margin-top: 16px; padding: 16px; background: #f8f9fa; border-radius: 4px;">
            <h3 style="margin-top: 0;">Message</h3>
            <p style="white-space: pre-wrap;">{{ message }}</p>
        </div>

        <p style="margin-top: 16px; color: #888; font-size: 12px;">
            Reply directly to this email to respond to {{ name }} at {{ email }}.
        </p>
    </div>
</body>
</html>
```

Create `src/templates/contact.html`:

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

{% block title %}Contact Us{% endblock %}

{% block content %}
    <h1>Contact Us</h1>

    {% if flash %}
        <div style="padding: 12px; border-radius: 4px; margin-bottom: 16px;
            {% if flash.type == 'success' %}background: #d4edda; color: #155724;{% endif %}
            {% if flash.type == 'error' %}background: #f8d7da; color: #721c24;{% endif %}">
            {{ flash.message }}
        </div>
    {% endif %}

    <form method="POST" action="/contact" style="max-width: 500px;">
        <div style="margin-bottom: 12px;">
            <label for="name" style="display: block; margin-bottom: 4px; font-weight: bold;">Nam
            <input type="text" name="name" id="name" required
                   style="width: 100%; padding: 8px; border: 1px solid #ddd; border-radius: 4px;
        </div>

        <div style="margin-bottom: 12px;">
            <label for="email" style="display: block; margin-bottom: 4px; font-weight: bold;">Em
            <input type="email" name="email" id="email" required
                   style="width: 100%; padding: 8px; border: 1px solid #ddd; border-radius: 4px;
        </div>

        <div style="margin-bottom: 12px;">
            <label for="subject" style="display: block; margin-bottom: 4px; font-weight: bold;">
```

```
            <input type="text" name="subject" id="subject" required
                   style="width: 100%; padding: 8px; border: 1px solid #ddd; border-radius: 4px;
        </div>

        <div style="margin-bottom: 12px;">
            <label for="message" style="display: block; margin-bottom: 4px; font-weight: bold;">
            <textarea name="message" id="message" rows="6" required
                      style="width: 100%; padding: 8px; border: 1px solid #ddd; border-radius: 4
        </div>

        <button type="submit"
                style="padding: 10px 20px; background: #1a1a2e; color: white; border: none; bord
            Send Message
        </button>
    </form>
{% endblock %}
```

Create `src/routes/contact.ts`:

```
import { Router, Messenger } from "tina4-nodejs";
import { Frond } from "tina4-nodejs/frond";

Router.get("/contact", async (req, res) => {
    const flash = req.session._flash ?? null;
    delete req.session._flash;
    return res.html("contact.html", { flash });
});

Router.post("/contact", async (req, res) => {
    const body = req.body;

    // Validate
    const errors: string[] = [];
    if (!body.name) errors.push("Name is required");
    if (!body.email) errors.push("Email is required");
    if (!body.subject) errors.push("Subject is required");
    if (!body.message) errors.push("Message is required");

    if (errors.length > 0) {
        req.session._flash = { type: "error", message: `Please fill in all fields: ${errors.join
        return res.redirect("/contact");
    }

    // Render the email template
    const htmlBody = await Frond.render("emails/contact-notification.html", {
        name: body.name,
        email: body.email,
        subject: body.subject,
        message: body.message,
        submitted_at: new Date().toLocaleString()
    });

    // Send the email
    const mailer = new Messenger();
    const adminEmail = process.env.ADMIN_EMAIL ?? "admin@example.com";

    const result = await mailer.send({
        to: adminEmail,
        subject: `Contact Form: ${body.subject}`,
```

```
        body: htmlBody,
        html: true,
        replyTo: body.email,
        text: `Contact form submission from ${body.name} (${body.email}):\n\nSubject: ${body.sub
    });

    if (result.success) {
        req.session._flash = {
            type: "success",
            message: "Thank you for your message! We will get back to you shortly."
        };
    } else {
        req.session._flash = {
            type: "error",
            message: "Sorry, there was a problem sending your message. Please try again later."
        };
    }

    return res.redirect("/contact");
});
```

Testing:

- Open `http://localhost:7148/contact` in your browser
- Fill in the form and submit
- You should see a green "Thank you" flash message
- Open `http://localhost:7148/__dev` to see the intercepted email
- The email shows the sender details, subject, message, and formatted HTML

API test:

```
curl -X POST http://localhost:7148/contact \
  -H "Content-Type: application/json" \
  -d '{"name": "Bob", "email": "bob@example.com", "subject": "Product inquiry", "message": "Do y
  -c cookies.txt -b cookies.txt
```

The response is a `302` redirect to `/contact`. Follow the redirect to see the flash message:

```
curl http://localhost:7148/contact -b cookies.txt
```

The HTML response includes the success flash message.

---

15. Gotchas

1. Gmail Blocks "Less Secure" Apps

Problem: Sending via Gmail fails with "Authentication failed" or "Username and Password not accepted".

Cause: Gmail blocks SMTP access from apps that do not use OAuth2 by default. Your regular password will not work when two-factor authentication is enabled.

Fix: Generate an "App Password" in your Google Account settings (Security > 2-Step Verification > App Passwords). Use this 16-character password as `TINA4_MAIL_PASSWORD`. This is

separate from your regular Google password.

2. Emails Go to Spam

Problem: Emails land in the recipient's spam folder.

Cause: Your sending domain lacks proper DNS records (SPF, DKIM, DMARC) or you send from a free email provider (Gmail, Yahoo).

Fix: Use a dedicated sending domain with proper DNS records. Set up SPF, DKIM, and DMARC records. Use a transactional email service -- Mailgun, SendGrid, or Amazon SES -- that handles email reputation for you.

3. HTML Email Looks Broken

Problem: The email looks fine in Gmail but broken in Outlook or Apple Mail.

Cause: Email clients have different HTML/CSS support. CSS flexbox, grid, and many modern properties do not work in email.

Fix: Use inline styles (not external stylesheets or `<style>` blocks). Use table-based layouts for complex designs. Test with an email preview tool. Keep it simple -- most transactional emails do not need elaborate designs.

4. Attachment File Not Found

Problem: `mailer.send()` returns an error about a missing file.

Cause: The attachment path is relative or incorrect. The file does not exist at the specified location.

Fix: Use absolute paths for attachments. Verify the file exists before calling `send()`: `if (!fs.existsSync(path)) { ... }`. If the file is generated dynamically (a PDF, for example), make sure the generation completes before sending.

5. Dev Mode Silently Intercepts Emails

Problem: You configured SMTP but no emails arrive. No errors either.

Cause: `TINA4_DEBUG=true` intercepts all emails and stores them in the dev dashboard. The email never reaches the SMTP server.

Fix: Check the dev dashboard at `/__dev` for intercepted emails. If you want to send real emails during development, set `TINA4_MAILBOX_DIR=false`. Remove this setting before committing.

6. Email Template Variables Not Substituted

Problem: The email body shows `{{ name }}` literally instead of the user's name.

Cause: You passed the raw template file content instead of rendering it through the template engine.

Fix: Use `Frond.render("emails/template.html", data)` to render the template with variables substituted. Do not read the file with `fs.readFileSync()` -- that gives you the raw template source.

7. Connection Timeout on Send

Problem: `mailer.send()` hangs for 30 seconds and then fails with a timeout error.

Cause: The SMTP server is unreachable from your network. The port may be blocked by a firewall, or the hostname may be wrong.

Fix: Test SMTP connectivity with `mailer.testConnection()`. Verify the hostname, port, and encryption settings. Check that your firewall allows outbound connections on the SMTP port. If you sit behind a corporate firewall, port 587 or 465 might be blocked -- ask your network administrator.

8. IMAP Host Not Configured

Problem: Calling `mailer.getInbox()` throws `MessengerError: IMAP host not configured`.

Cause: You did not set `TINA4_MAIL_IMAP_HOST` in `.env` or pass `imapHost` to the constructor.

Fix: Add `TINA4_MAIL_IMAP_HOST=imap.example.com` and `TINA4_MAIL_IMAP_PORT=993` to `.env`. The IMAP host is separate from the SMTP host -- many providers use different hostnames for sending and reading.

Frontend Integration

1. Beyond JSON APIs

Your API returns perfect JSON. Now someone has to build the interface.

Tina4 ships two frontend tools: **tina4css** (a utility CSS framework) and **frond.js** (a reactive JavaScript library). Both arrive with every project. No npm installs. No build tools. No webpack.

This chapter ends with a complete admin dashboard. Sidebar. Navigation. Cards. Tables. Modals. Dark mode. Progress bars. AJAX-driven user management. Zero npm dependencies.

---

2. What Ships with Tina4

Run `tina4 init nodejs`. Several files appear in your project:

```
src/public/
■■■ css/
■   ■■■ tina4.css        # The CSS framework
■■■ js/
■   ■■■ tina4.min.js     # Core AJAX utilities
■   ■■■ frond.min.js     # Template engine client-side helpers
■   ■■■ tina4js.min.js   # Reactive frontend framework (tina4-js)
■■■ scss/
    ■■■ tina4.scss       # SCSS source (optional, for customization)
```

Include them in any template:

```
<link rel="stylesheet" href="/css/tina4.css">
<script src="/js/tina4.min.js"></script>
<script src="/js/frond.min.js"></script>
```

Include only what you need. See section 7 for the full JavaScript API reference. No CDN. No package manager. No version conflicts.

---

3. The Grid System

tina4css uses a 12-column responsive grid. The class names match Bootstrap. Know Bootstrap? You know tina4css.

```
<div class="container">
    <div class="row">
        <div class="col-md-4">
            <p>One third</p>
        </div>
        <div class="col-md-4">
            <p>One third</p>
        </div>
        <div class="col-md-4">
            <p>One third</p>
        </div>
```

```
    </div>
</div>
```

Breakpoints:

Class Prefix	Screen Width	Typical Device
`col-`	All sizes	Phones and up
`col-sm-`	>= 576px	Large phones
`col-md-`	>= 768px	Tablets
`col-lg-`	>= 992px	Desktops
`col-xl-`	>= 1200px	Large desktops

Columns stack vertically on screens smaller than their breakpoint. A `col-md-6` element takes half the row on tablets and up, full width on phones.

---

4. Components

Navbar

```
<nav class="navbar navbar-dark bg-dark">
    <div class="container">
        <a class="navbar-brand" href="/">My Dashboard</a>
        <ul class="navbar-nav">
            <li class="nav-item"><a class="nav-link" href="/dashboard">Dashboard</a></li>
            <li class="nav-item"><a class="nav-link" href="/products">Products</a></li>
            <li class="nav-item"><a class="nav-link" href="/settings">Settings</a></li>
        </ul>
    </div>
</nav>
```

Use `navbar-light bg-light` for a light theme, or `navbar-dark bg-primary` for a colored background.

Cards

```
<div class="card">
    <div class="card-header">Monthly Revenue</div>
    <div class="card-body">
        <h2 class="card-title">$12,450</h2>
        <p class="card-text">Up 12% from last month</p>
    </div>
    <div class="card-footer text-muted">Updated 5 minutes ago</div>
</div>
```

Buttons

```
<button class="btn btn-primary">Save</button>
<button class="btn btn-secondary">Cancel</button>
<button class="btn btn-danger">Delete</button>
<button class="btn btn-success">Publish</button>
<button class="btn btn-outline-primary">Outlined</button>
<button class="btn btn-sm btn-primary">Small</button>
<button class="btn btn-lg btn-primary">Large</button>
```

Tables

```
<table class="table table-striped table-hover">
    <thead>
        <tr>
            <th>Name</th>
            <th>Category</th>
            <th>Price</th>
            <th>Status</th>
        </tr>
    </thead>
    <tbody>
        <tr>
            <td>Wireless Keyboard</td>
            <td>Electronics</td>
            <td>$79.99</td>
            <td><span class="badge bg-success">In Stock</span></td>
        </tr>
        <tr>
            <td>Standing Desk</td>
            <td>Furniture</td>
            <td>$549.99</td>
            <td><span class="badge bg-danger">Out of Stock</span></td>
        </tr>
    </tbody>
</table>
```

Table variants: `table-bordered`, `table-striped`, `table-hover`, `table-sm` (compact), `table-responsive` (wraps in a scrollable container on small screens). Mix and match.

Alerts

```
<div class="alert alert-success">Product created.</div>
<div class="alert alert-danger">Failed to save changes.</div>
<div class="alert alert-warning alert-dismissible">
    Your trial expires in 3 days.
    <button type="button" class="close" data-dismiss="alert">&times;</button>
</div>
<div class="alert alert-info">A new version is available.</div>
```

Modals

```
<button class="btn btn-primary" data-toggle="modal" data-target="#confirmModal">Delete Product</

<div class="modal" id="confirmModal">
    <div class="modal-dialog">
        <div class="modal-content">
            <div class="modal-header">
                <h5 class="modal-title">Confirm Delete</h5>
                <button class="btn-close" data-dismiss="modal"></button>
            </div>
            <div class="modal-body">
                <p>Are you sure you want to delete this product? This action cannot be undone.</
            </div>
            <div class="modal-footer">
                <button class="btn btn-secondary" data-dismiss="modal">Cancel</button>
                <button class="btn btn-danger">Delete</button>
            </div>
        </div>
    </div>
</div>
```

tina4css includes the JavaScript for modal toggling. No jQuery required.

Progress Bars

Progress bars visualize completion, upload status, or system health. The outer `progress` container holds a `progress-bar` fill element.

```
<div class="progress">
    <div class="progress-bar bg-success" style="width: 75%">75%</div>
</div>

<div class="progress">
    <div class="progress-bar bg-warning progress-bar-striped" style="width: 45%">
        45% - Uploading...
    </div>
</div>
```

Set the width with inline `style`. Add `bg-success`, `bg-info`, `bg-warning`, or `bg-danger` for color. Add `progress-bar-striped` for animated stripes.

Badges

```
<span class="badge bg-primary">Primary</span>
<span class="badge bg-success">Active</span>
<span class="badge bg-warning">Pending</span>
<span class="badge bg-danger">Overdue</span>
<span class="badge bg-info">Info</span>
<span class="badge bg-dark">Dark</span>
```

Forms

```
<form>
    <div class="form-group">
        <label for="name" class="form-label">Product Name</label>
        <input type="text" class="form-control" id="name" placeholder="Enter product name">
    </div>

    <div class="form-group">
        <label for="category" class="form-label">Category</label>
        <select class="form-control" id="category">
            <option value="">Select a category</option>
            <option value="electronics">Electronics</option>
            <option value="furniture">Furniture</option>
        </select>
    </div>

    <div class="form-group">
        <label for="description" class="form-label">Description</label>
        <textarea class="form-control" id="description" rows="4"></textarea>
    </div>

    <div class="form-check">
        <input type="checkbox" class="form-check-input" id="featured">
        <label class="form-check-label" for="featured">Featured product</label>
    </div>

    <button type="submit" class="btn btn-primary mt-3">Save Product</button>
</form>
```

Utility Classes

```
<p class="text-center">Centered</p>
<div class="mt-4">Margin top</div>
<div class="p-3">Padding</div>
<span class="text-muted">Gray text</span>
```

---

5. SCSS Customization

The default tina4css works out of the box. To customize colors, fonts, or spacing, edit the SCSS source.

Edit `src/public/scss/tina4.scss`:

```
// Override variables before importing the framework
$primary: #2d6a4f;
$secondary: #52b788;
$dark: #1b4332;
$font-family-base: 'Inter', sans-serif;
$border-radius: 8px;

// Import the framework
@import 'tina4-base';
```

Compile SCSS to CSS:

```
tina4 scss
```

```
Compiling SCSS...
  src/public/scss/tina4.scss -> src/public/css/tina4.css
Done (0.12s)
```

The compiled CSS replaces the default `tina4.css`. Your custom colors and fonts take effect across the entire application.

Live SCSS Compilation

During development, run SCSS compilation in watch mode:

```
tina4 scss --watch
```

Every save to a `.scss` file triggers a recompile. Combined with Tina4's live reload, changes appear in the browser within a second.

---

6. frond.js -- The JavaScript Helper

`frond.js` is a lightweight JavaScript library that ships with Tina4. It handles AJAX requests, form submission, JWT token management, and loading indicators. No jQuery. No Axios. No other dependency.

AJAX Requests

```
// GET request
frond.get("/api/products", function (data) {
    console.log("Products:", data);
});

// GET with error handling
frond.get("/api/products", function (data) {
    console.log(data);
}, function (error) {
    console.error("Failed:", error);
});

// POST request
frond.post("/api/products", {
    name: "New Product",
    price: 29.99
}, function (data) {
    console.log("Created:", data);
});

// PUT request
frond.put("/api/products/1", {
    name: "Updated Product"
}, function (data) {
    console.log("Updated:", data);
});

// DELETE request
frond.delete("/api/products/1", function (data) {
    console.log("Deleted:", data);
});
```

Form Submission via AJAX

```
<form id="product-form" data-frond-submit="/api/products" data-frond-method="POST">
    <input type="text" name="name" placeholder="Product name">
    <input type="number" name="price" placeholder="Price">
    <button type="submit">Create</button>
</form>

<script src="/js/frond.min.js"></script>
<script>
    frond.onFormSuccess("product-form", function (data) {
        alert("Product created: " + data.name);
    });

    frond.onFormError("product-form", function (error) {
        alert("Error: " + error.message);
    });
</script>
```

The `data-frond-submit` attribute tells frond.js to intercept the form submission and send it as an AJAX request. No page reload. frond.js serializes all form fields as JSON.

Token Management

frond.js manages JWT tokens. Store a token after login, and frond.js attaches it to every request.

```
// Store the token (usually after login)
frond.setToken("eyJhbGciOiJIUzI1NiIs...");

// All subsequent requests include:
// Authorization: Bearer eyJhbGciOiJIUzI1NiIs...
frond.get("/api/profile", function (data) {
    console.log("Profile:", data);
});

// Clear the token (logout)
frond.clearToken();
```

frond.js stores the token in `localStorage` and includes it as a `Bearer` token in the `Authorization` header on every request.

Loading Indicators

frond.js can show and hide a loading element during AJAX requests. Pass a CSS selector in the options object.

```
// Show a loading state while fetching
frond.get("/api/products", function (data) {
    renderProducts(data.products);
}, null, {
    loading: "#loadingSpinner"  // CSS selector for loading element
});
```

The element with id `loadingSpinner` appears while the request flies and disappears when it completes. Pair it with a spinner or "Loading..." text in your HTML:

```
<div id="loadingSpinner" class="text-center p-4" style="display: none;">
    Loading...
</div>
```

frond.js toggles `display: block` and `display: none` on the element. No extra CSS needed.

WebSocket (Covered in Chapter 23)

```
const ws = frond.ws("/ws/chat/general");
ws.on("message", function (data) {
    console.log("Message:", JSON.parse(data));
});
```

Connections drop. frond.js reconnects with exponential backoff. If the server restarts or the network blips, the client reconnects without intervention.

---

7. JavaScript API Reference

Tina4 ships three JavaScript files. Each serves a different purpose. Use them independently or together.

Including the Scripts

```
<script src="/js/tina4.min.js"></script>
<script src="/js/frond.min.js"></script>
<script src="/js/tina4js.min.js"></script>
```

All three live in `src/public/js/` and are served from `/js/`. Include only what you need.

7.1 tina4.min.js -- Core Utilities

Low-level helpers for AJAX page loading and form submission. Use this when you want simple dynamic page updates without a full framework.

`loadPage(url, targetId)`

Fetches HTML from `url` and injects it into the element with the given `id`.

```
<nav>
    <a href="#" onclick="loadPage('/dashboard', 'content')">Dashboard</a>
    <a href="#" onclick="loadPage('/settings', 'content')">Settings</a>
</nav>
<div id="content"><!-- pages load here --></div>

<script src="/js/tina4.min.js"></script>
```

`saveForm(formId, url, method)`

Serializes a form and submits it via AJAX. Prevents the default page reload.

```
<form id="product-form">
    <input type="text" name="name" placeholder="Product name">
    <input type="number" name="price" placeholder="Price">
    <button type="button" onclick="saveForm('product-form', '/api/products', 'POST')">
        Save
    </button>
</form>
```

`sendRequest(url, method, data, callback)`

Generic AJAX helper for any HTTP method. Returns the response to a callback function.

```
sendRequest("/api/products", "GET", null, function (response) {
    console.log("Products:", JSON.parse(response));
});

sendRequest("/api/products", "POST", { name: "Widget", price: 9.99 }, function (response) {
    console.log("Created:", JSON.parse(response));
});
```

7.2 frond.min.js -- Template Engine Client-Side Helpers

A companion to the Frond template engine. Handles AJAX form interception, WebSocket connections with auto-reconnect, JWT token refresh, and dynamic template loading.

AJAX Form Handling

Forms with `data-frond-submit` are intercepted. No page reload. No boilerplate.

```
<form id="login-form" data-frond-submit="/api/login" data-frond-method="POST">
    <input type="text" name="username" placeholder="Username">
    <input type="password" name="password" placeholder="Password">
    <button type="submit">Log In</button>
</form>

<script src="/js/frond.min.js"></script>
<script>
    frond.onFormSuccess("login-form", function (data) {
        frond.setToken(data.token);
        window.location.href = "/dashboard";
    });
</script>
```

WebSocket Auto-Reconnect

```
const ws = frond.ws("/ws/notifications");
ws.on("message", function (data) {
    const notification = JSON.parse(data);
    alert(notification.text);
});
// If the server restarts or the network blips, frond.js reconnects.
```

Token Refresh

JWT tokens stored via `frond.setToken()` are attached to every request. When a token expires, frond.js triggers a refresh before retrying the request.

```
frond.setToken("eyJhbGciOiJIUzI1NiIs...");
// All subsequent frond.get/post/put/delete calls include the token.
// When the token expires, frond.js calls the refresh endpoint.
```

Dynamic Template Loading

Load server-rendered Frond templates into any element without a full page reload.

```
frond.loadTemplate("/templates/user-card", { userId: 42 }, "user-panel");
```

```
// Fetches the rendered template and injects it into #user-panel.
```

7.3 tina4js.min.js -- Reactive Frontend Framework

A standalone reactive framework for building interactive client-side applications. Provides signals, computed values, effects, Web Components, client-side routing, and built-in fetch and WebSocket wrappers. This is the **tina4-js** project.

Reactive State: `signal()`, `computed()`, `effect()`

```
import { signal, computed, effect } from "/js/tina4js.min.js";

const count = signal(0);
const doubled = computed(() => count.value * 2);

effect(() => {
    console.log(`Count: ${count.value}, Doubled: ${doubled.value}`);
});

count.value = 5; // logs "Count: 5, Doubled: 10"
```

DOM Rendering: `html` Tagged Template

```
import { signal, html } from "/js/tina4js.min.js";

const name = signal("World");

const app = html`
    <div>
        <h1>Hello, ${name}!</h1>
        <input value="${name}" oninput="${(e) => name.value = e.target.value}" />
    </div>
`;

document.getElementById("app").append(app);
```

Web Components: `Tina4Element`

```
import { Tina4Element, signal, html } from "/js/tina4js.min.js";

class CounterButton extends Tina4Element {
    setup() {
        this.count = signal(0);
    }
    render() {
        return html`
            <button onclick="${() => this.count.value++}">
                Clicked ${this.count} times
            </button>
        `;
    }
}
customElements.define("counter-button", CounterButton);
```

Use it in HTML:

```
<counter-button></counter-button>
<script type="module" src="/js/counter-button.js"></script>
```

Fetch Wrapper: `api()`

```
import { api } from "/js/tina4js.min.js";

const products = await api("/api/products");            // GET
await api("/api/products", { method: "POST", body: { name: "Widget" } });
```

WebSocket Client: `ws()`

```
import { ws } from "/js/tina4js.min.js";

const socket = ws("/ws/chat");
socket.on("message", (data) => console.log(data));
socket.send({ text: "Hello" });
```

Client-Side Routing: `route()`, `navigate()`

```
import { route, navigate, html } from "/js/tina4js.min.js";

route("/", () => html`<h1>Home</h1>`);
route("/about", () => html`<h1>About</h1>`);

// Navigate programmatically
navigate("/about");
```

---

8. Building an Admin Dashboard

A complete admin dashboard. The kind of page that powers the backend of every web application.

Base Template

Create `src/templates/base.html`:

```
<!DOCTYPE html>
<html lang="en" data-theme="light">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>{% block title %}Dashboard{% endblock %}</title>
    <link rel="stylesheet" href="/css/tina4.css">
    <script>
        var t = localStorage.getItem("theme");
        if (t) document.documentElement.setAttribute("data-theme", t);
    </script>
</head>
<body>
    <nav class="navbar navbar-dark bg-dark">
        <div class="container-fluid">
            <a class="navbar-brand" href="/admin">TaskFlow Admin</a>
            <ul class="navbar-nav">
                <li class="nav-item"><a class="nav-link" href="/admin">Dashboard</a></li>
                <li class="nav-item"><a class="nav-link" href="/admin/products">Products</a></li
                <li class="nav-item"><a class="nav-link" href="/admin/users">Users</a></li>
                <li class="nav-item">
                    <button class="btn btn-sm btn-outline-light" onclick="toggleDarkMode()">Dark
                </li>
```

```
            </ul>
        </div>
    </nav>

    <div class="container-fluid mt-4">
        <div class="row">
            <div class="col-md-2">
                {% block sidebar %}
                <div class="list-group">
                    <a href="/admin" class="list-group-item list-group-item-action">Overview</a>
                    <a href="/admin/products" class="list-group-item list-group-item-action">Pro
                    <a href="/admin/orders" class="list-group-item list-group-item-action">Order
                    <a href="/admin/customers" class="list-group-item list-group-item-action">Cu
                    <a href="/admin/reports" class="list-group-item list-group-item-action">Repo
                    <a href="/admin/settings" class="list-group-item list-group-item-action">Set
                </div>
                {% endblock %}
            </div>
            <div class="col-md-10">
                {% block content %}{% endblock %}
            </div>
        </div>
    </div>

    <script src="/js/frond.min.js"></script>
    <script>
        function toggleDarkMode() {
            var html = document.documentElement;
            var current = html.getAttribute("data-theme");
            html.setAttribute("data-theme", current === "dark" ? "light" : "dark");
            localStorage.setItem("theme", current === "dark" ? "light" : "dark");
        }
    </script>
    {% block scripts %}{% endblock %}
</body>
</html>
```

Dashboard Page

Create `src/templates/dashboard.html`:

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

{% block title %}Dashboard - Admin{% endblock %}

{% block content %}
    <h1>Dashboard</h1>

    <div class="row mb-4">
        <div class="col-md-3">
            <div class="card">
                <div class="card-body">
                    <h6 class="card-subtitle text-muted">Total Products</h6>
                    <h2 class="card-title">{{ stats.total_products }}</h2>
                </div>
            </div>
        </div>
        <div class="col-md-3">
            <div class="card">
```

```
            <div class="card-body">
                <h6 class="card-subtitle text-muted">Total Orders</h6>
                <h2 class="card-title">{{ stats.total_orders }}</h2>
            </div>
        </div>
    </div>
    <div class="col-md-3">
        <div class="card">
            <div class="card-body">
                <h6 class="card-subtitle text-muted">Revenue</h6>
                <h2 class="card-title">${{ stats.revenue }}</h2>
            </div>
        </div>
    </div>
    <div class="col-md-3">
        <div class="card">
            <div class="card-body">
                <h6 class="card-subtitle text-muted">Active Users</h6>
                <h2 class="card-title">{{ stats.active_users }}</h2>
            </div>
        </div>
    </div>
</div>

<div class="row">
    <div class="col-md-8">
        <div class="card">
            <div class="card-header">Recent Orders</div>
            <div class="card-body">
                <table class="table table-striped">
                    <thead>
                        <tr>
                            <th>Order</th>
                            <th>Customer</th>
                            <th>Amount</th>
                            <th>Status</th>
                        </tr>
                    </thead>
                    <tbody>
                        {% for order in recent_orders %}
                        <tr>
                            <td>#{{ order.id }}</td>
                            <td>{{ order.customer }}</td>
                            <td>${{ order.amount }}</td>
                            <td>
                                <span class="badge bg-{{ order.badge }}">{{ order.status }}<
                            </td>
                        </tr>
                        {% endfor %}
                    </tbody>
                </table>
            </div>
        </div>
    </div>
    <div class="col-md-4">
        <div class="card">
            <div class="card-header">Quick Actions</div>
            <div class="card-body">
                <a href="/admin/products/new" class="btn btn-primary btn-block mb-2">Add Pro
```

```
                    <a href="/admin/orders" class="btn btn-outline-primary btn-block mb-2">View
                    <a href="/admin/reports" class="btn btn-outline-secondary btn-block">Generat
                </div>
            </div>

            <div class="card mt-3">
                <div class="card-header">System Health</div>
                <div class="card-body">
                    <p><strong>CPU:</strong></p>
                    <div class="progress mb-3">
                        <div class="progress-bar bg-success" style="width: {{ stats.cpu_usage }}
                            {{ stats.cpu_usage }}%
                        </div>
                    </div>
                    <p><strong>Memory:</strong></p>
                    <div class="progress mb-3">
                        <div class="progress-bar bg-info" style="width: {{ stats.memory_usage }}
                            {{ stats.memory_usage }}%
                        </div>
                    </div>
                    <p><strong>Disk:</strong></p>
                    <div class="progress">
                        <div class="progress-bar bg-warning" style="width: {{ stats.disk_usage }
                            {{ stats.disk_usage }}%
                        </div>
                    </div>
                </div>
            </div>
        </div>
    </div>
{% endblock %}
```

Dashboard Route

Create `src/routes/admin.ts`:

```
import { Router } from "tina4-nodejs";

Router.get("/admin", async (req, res) => {
    const stats = {
        total_products: 156,
        total_orders: 1243,
        revenue: "24,580",
        active_users: 89,
        cpu_usage: 42,
        memory_usage: 68,
        disk_usage: 55
    };

    const recent_orders = [
        { id: 1042, customer: "Alice Johnson", amount: "129.99", status: "Shipped", badge: "succ
        { id: 1041, customer: "Bob Smith", amount: "549.99", status: "Processing", badge: "warni
        { id: 1040, customer: "Carol White", amount: "79.99", status: "Delivered", badge: "info"
        { id: 1039, customer: "Dave Brown", amount: "34.99", status: "Cancelled", badge: "danger
    ];

    return res.html("dashboard.html", { stats, recent_orders });
});
```

Start the server and visit `http://localhost:7148/admin`. You see a sidebar, stat cards, a data table, quick action buttons, and system health progress bars. Zero npm dependencies.

---

9. Dark Mode

tina4css supports dark mode through a single attribute on the `<html>` element:

```
<!-- Light mode (default) -->
<html data-theme="light">

<!-- Dark mode -->
<html data-theme="dark">
```

The toggle function from the base template handles switching:

```
function toggleDarkMode() {
    var html = document.documentElement;
    var current = html.getAttribute("data-theme");
    html.setAttribute("data-theme", current === "dark" ? "light" : "dark");
    localStorage.setItem("theme", current === "dark" ? "light" : "dark");
}
```

Dark mode transforms every surface. Backgrounds darken. Text colors invert. Borders shift. Cards, tables, buttons -- all adjust contrast. Zero CSS rules required from you.

Respecting System Preference

Match the user's operating system dark mode setting:

```
var saved = localStorage.getItem("theme");
if (saved) {
    document.documentElement.setAttribute("data-theme", saved);
} else if (window.matchMedia("(prefers-color-scheme: dark)").matches) {
    document.documentElement.setAttribute("data-theme", "dark");
}
```

---

10. Responsive Design

tina4css is mobile-first. Components stack on small screens and expand on larger ones.

Responsive Sidebar

On mobile, the sidebar collapses into a toggle:

```
<button class="btn btn-dark d-md-none" onclick="toggleSidebar()">Menu</button>

<div id="sidebar" class="d-none d-md-block col-md-2">
    <!-- sidebar content -->
</div>

<script>
function toggleSidebar() {
    var sidebar = document.getElementById("sidebar");
    sidebar.classList.toggle("d-none");
```

```
}
</script>
```

Responsive Tables

Tables scroll horizontally on small screens:

```
<div class="table-responsive">
    <table class="table">
        <!-- table content -->
    </table>
</div>
```

Hiding Elements by Screen Size

```
<div class="d-none d-md-block">Only visible on tablet and up</div>
<div class="d-md-none">Only visible on mobile</div>
```

---

11. Building a Users Page with AJAX

A user management page. Data loads via AJAX using frond.js. No full page reloads. The table populates from the API. Users create, edit, and delete records through modals and inline actions.

The Template

Create `src/templates/admin/users.html`:

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

{% block title %}Users - Admin{% endblock %}

{% block content %}
    <div class="card">
        <div class="card-header d-flex justify-content-between align-items-center">
            <span>All Users</span>
            <button class="btn btn-sm btn-primary" data-toggle="modal" data-target="#addUserModa
                Add User
            </button>
        </div>
        <div class="card-body p-0">
            <div id="loadingSpinner" class="text-center p-4" style="display: none;">
                Loading...
            </div>
            <table class="table table-hover m-0" id="usersTable">
                <thead>
                    <tr>
                        <th>ID</th>
                        <th>Name</th>
                        <th>Email</th>
                        <th>Created</th>
                        <th>Actions</th>
                    </tr>
                </thead>
                <tbody id="usersBody">
                    <!-- Populated by JavaScript -->
                </tbody>
            </table>
```

```
        </div>
    </div>

    <!-- Add User Modal -->
    <div class="modal" id="addUserModal">
        <div class="modal-dialog">
            <div class="modal-content">
                <div class="modal-header">
                    <h5 class="modal-title">Add New User</h5>
                    <button type="button" class="btn-close" data-dismiss="modal"></button>
                </div>
                <div class="modal-body">
                    <form id="addUserForm">
                        <div class="form-group">
                            <label for="userName">Name</label>
                            <input type="text" class="form-control" name="name" id="userName" re
                        </div>
                        <div class="form-group">
                            <label for="userEmail">Email</label>
                            <input type="email" class="form-control" name="email" id="userEmail"
                        </div>
                    </form>
                </div>
                <div class="modal-footer">
                    <button class="btn btn-secondary" data-dismiss="modal">Cancel</button>
                    <button class="btn btn-primary" onclick="createUser()">Create User</button>
                </div>
            </div>
        </div>
    </div>

    <div id="alertArea" class="mt-3"></div>
{% endblock %}

{% block scripts %}
<script>
    function loadUsers() {
        frond.get("/api/users", function (data) {
            var tbody = document.getElementById("usersBody");
            tbody.innerHTML = "";

            if (data.data && data.data.length > 0) {
                data.data.forEach(function (user) {
                    var row = '<tr>'
                        + '<td>' + user.id + '</td>'
                        + '<td>' + user.name + '</td>'
                        + '<td>' + user.email + '</td>'
                        + '<td>' + user.created_at + '</td>'
                        + '<td>'
                        + '<button class="btn btn-sm btn-outline-primary" onclick="editUser(' +
                        + '<button class="btn btn-sm btn-outline-danger" onclick="deleteUser(' +
                        + '</td>'
                        + '</tr>';
                    tbody.innerHTML += row;
                });
            } else {
                tbody.innerHTML = '<tr><td colspan="5" class="text-center p-4">No users found</t
            }
        }, null, { loading: "#loadingSpinner" });
```

```
    }

    function createUser() {
        var name = document.getElementById("userName").value;
        var email = document.getElementById("userEmail").value;

        frond.post("/api/users", { name: name, email: email }, function (data) {
            document.getElementById("alertArea").innerHTML =
                '<div class="alert alert-success">User "' + data.name + '" created.</div>';
            loadUsers();
        }, function (error) {
            document.getElementById("alertArea").innerHTML =
                '<div class="alert alert-danger">Error creating user.</div>';
        });
    }

    function deleteUser(id) {
        if (confirm("Are you sure you want to delete this user?")) {
            frond.delete("/api/users/" + id, function () {
                loadUsers();
            });
        }
    }

    // Load users on page load
    loadUsers();
</script>
{% endblock %}
```

The Route

```
import { Router } from "tina4-nodejs";

Router.get("/admin/users", async (req, res) => {
    return res.html("admin/users.html", {});
});
```

This page loads users via an AJAX call to `/api/users` (provided by auto-CRUD on the User model from Chapter 6). The loading indicator appears while the request flies. The table populates when data arrives. Users add records through a modal form and delete with confirmation -- all without full page reloads.

---

12. Building a CRUD Interface

Create `src/templates/product-manager.html`:

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

{% block title %}Product Manager{% endblock %}

{% block content %}
    <h1>Product Manager</h1>

    <div id="app">
        <form id="add-form" class="card" style="padding: 16px; margin-bottom: 16px;">
            <h3>Add Product</h3>
```

```
            <div class="form-group">
                <label for="name">Name</label>
                <input type="text" id="name" class="form-control" required>
            </div>
            <div class="form-group">
                <label for="price">Price</label>
                <input type="number" id="price" class="form-control" step="0.01" required>
            </div>
            <button type="submit" class="btn btn-primary">Add Product</button>
        </form>

        <div id="product-list"></div>
    </div>

    <script src="/js/frond.js"></script>
    <script>
        async function loadProducts() {
            const data = await frond.get("/api/products");
            const list = document.getElementById("product-list");
            list.innerHTML = data.products.map(p =>
                `<div class="card" style="padding: 12px; margin-bottom: 8px;">
                    <strong>${p.name}</strong> - $${p.price.toFixed(2)}
                    <button class="btn btn-danger" style="float:right" onclick="deleteProduct(${
                </div>`
            ).join("");
        }

        document.getElementById("add-form").addEventListener("submit", async (e) => {
            e.preventDefault();
            const name = document.getElementById("name").value;
            const price = document.getElementById("price").value;
            await frond.post("/api/products", { name, price: parseFloat(price) });
            document.getElementById("name").value = "";
            document.getElementById("price").value = "";
            loadProducts();
        });

        async function deleteProduct(id) {
            await frond.del("/api/products/" + id);
            loadProducts();
        }

        loadProducts();
    </script>
{% endblock %}
```

---

13. Static File Serving

Files in `src/public/` are served directly:

```
src/public/images/logo.png → /images/logo.png
src/public/js/app.js       → /js/app.js
src/public/css/custom.css  → /css/custom.css
```

---

14. Integrating with React, Vue, or Svelte

Tina4 is the API backend. Point your frontend build tool's output to `src/public/`:

```
# Vue
VITE_OUTPUT_DIR=../my-tina4-project/src/public

# React (Create React App)
BUILD_PATH=../my-tina4-project/src/public
```

Or use CORS to run them on separate ports during development:

```
CORS_ORIGINS=http://localhost:3000,http://localhost:5173
```

---

15. Exercise: Build an Admin Dashboard

Build an admin dashboard for a product management system.

Requirements

- Create a base template with:
- A dark navbar with the app name and navigation links
- A sidebar with menu items (Dashboard, Products, Orders, Settings)
- A main content area
- Dark mode toggle that persists across page loads
- Create a dashboard page at `GET /admin` with:
- Four stat cards (Products, Orders, Revenue, Users)
- A table showing recent orders with status badges
- Progress bars showing system health (CPU, Memory, Disk)
- Create a users page at `GET /admin/users` with:
- A table of users loaded via AJAX using frond.js
- An "Add User" button that opens a modal with a form
- AJAX form submission that refreshes the table without a page reload
- A loading indicator while data fetches
- Use tina4css classes throughout (no custom CSS needed)

Test by:

- Visit `http://localhost:7148/admin` -- see the dashboard with stats, orders, and progress bars
- Click "Dark Mode" -- the entire page switches to dark theme
- Refresh the page -- dark mode persists
- Resize the browser to mobile width -- the sidebar collapses

- Visit `http://localhost:7148/admin/users` -- see the user table loaded via AJAX

---

16. Solution

The base template, dashboard page, and users page are shown in sections 8 and 11. The product list page follows the same AJAX pattern from section 11, but for products instead of users.

For the users page, use auto-CRUD on the User model (`autoCrud = true`) so the API endpoints exist at `/api/users`. Load the table with `frond.get("/api/users", ...)` and handle form submission with `frond.post("/api/users", ...)`.

Add the product list route:

```
import { Router } from "tina4-nodejs";

Router.get("/admin/products", async (req, res) => {
    const search = req.query.search ?? "";
    const selectedCategory = req.query.category ?? "";

    let products = [
        { id: 1, name: "Wireless Keyboard", category: "Electronics", price: "79.99", inStock: tr
        { id: 2, name: "Standing Desk", category: "Furniture", price: "549.99", inStock: true },
        { id: 3, name: "Coffee Grinder", category: "Kitchen", price: "49.99", inStock: false },
        { id: 4, name: "Yoga Mat", category: "Fitness", price: "29.99", inStock: true },
        { id: 5, name: "USB-C Hub", category: "Electronics", price: "49.99", inStock: true },
    ];

    if (selectedCategory) {
        products = products.filter(p => p.category === selectedCategory);
    }

    if (search) {
        products = products.filter(p => p.name.toLowerCase().includes(search.toLowerCase()));
    }

    const categories = ["Electronics", "Furniture", "Kitchen", "Fitness"];

    return res.html("products.html", { products, categories, search, selected_category: selected
});
```

---

17. Gotchas

1. CSS Not Loading

Problem: The page looks unstyled -- no colors, no layout, plain text.

Cause: The path to `tina4.css` is wrong. The file lives in `src/public/css/` but the link points elsewhere.

Fix: Tina4 serves everything in `src/public/` from the root path. Link to `/css/tina4.css` (not `/src/public/css/tina4.css`). Verify the file exists at `src/public/css/tina4.css`.

2. frond.js Functions Not Found

Problem: `frond.get is not a function` or `frond is not defined` in the browser console.

Cause: The `frond.min.js` script tag is missing or placed after the code that uses it.

Fix: Include `<script src="/js/frond.min.js"></script>` before any script that calls `frond.*`. Put it at the bottom of the body, before your custom scripts.

3. Dark Mode Flickers on Page Load

Problem: The page loads in light mode and then flashes to dark mode.

Cause: The dark mode JavaScript runs after the page renders. The browser paints the light theme first, then switches.

Fix: Add the theme detection script in the `<head>` (before the body renders):

```
<head>
    <script>
        var t = localStorage.getItem("theme");
        if (t) document.documentElement.setAttribute("data-theme", t);
    </script>
</head>
```

4. SCSS Not Compiling

Problem: You edited `tina4.scss` but the CSS did not change.

Cause: SCSS does not compile on its own. You need to run `tina4 scss` or use `--watch` mode.

Fix: Run `tina4 scss --watch` during development. For production builds, run `tina4 scss` as part of your build process.

5. Modal Does Not Open

Problem: Clicking the button does nothing -- the modal stays hidden.

Cause: The `data-toggle` and `data-target` attributes require frond.js to be loaded. Without it, no JavaScript handles the modal toggle.

Fix: Ensure `frond.min.js` is loaded. Verify the `data-target` matches the modal's `id` exactly (including the `#` prefix).

6. Grid Columns Do Not Stack on Mobile

Problem: Columns stay side-by-side on phone screens instead of stacking vertically.

Cause: You used `col-4` instead of `col-md-4`. The `col-4` class applies at all screen sizes.

Fix: Use responsive prefixes: `col-md-4` means "one-third on medium screens and up, full width on small screens."

7. Static Files Return 404

Problem: CSS, JS, or image files return 404 Not Found.

Cause: The files are not in the `src/public/` directory.

Fix: Static files must be in `src/public/`. The URL path maps to the file path within that directory. `/css/tina4.css` maps to `src/public/css/tina4.css`.

8. Form Data Not Reaching the Server

Problem: `frond.post()` sends the request but `req.body` is empty on the server.

Cause: frond.js sends JSON by default. Passing a `FormData` object or building the data object wrong prevents correct parsing.

Fix: Pass a plain JavaScript object to `frond.post()`. Do not use `new FormData()` -- frond.js handles serialization. Make sure your object keys match what the server expects.

9. Loading Indicator Never Disappears

Problem: The loading spinner shows but never hides after the AJAX request completes.

Cause: The CSS selector passed to the `loading` option does not match any element, or the element uses a CSS class to toggle visibility instead of inline `display`.

Fix: frond.js toggles `display: block` and `display: none`. Make sure the element exists and its initial style is `display: none`. Use an `id` selector: `{ loading: "#loadingSpinner" }`.

10. CORS Errors with Separate Frontend

Fix: Set `CORS_ORIGINS` in `.env`.

11. Large Bundle Sizes

Fix: Use code splitting. Serve static assets from a CDN.

12. React Router Conflicts

Fix: Add a catch-all route that serves `index.html` for client-side routing.

---

18. HtmlElement — Programmatic HTML Builder

Build HTML in TypeScript without string concatenation:

```
import { HtmlElement, htmlElement, addHtmlHelpers } from "@tina4/core";

const el = new HtmlElement("div", { class: "card" }, ["Hello"]);
el.toString(); // '<div class="card">Hello</div>'

// Nesting
const card = new HtmlElement("div", { class: "card" }, [
    new HtmlElement("h2", {}, ["Title"]),
    new HtmlElement("p", {}, ["Content"]),
]);

// Helper functions
const h: Record<string, any> = {};
addHtmlHelpers(h);
```

```
const html = h._div({ class: "card" }, h._p("Hello"), h._a({ href: "/" }, "Home"));
```

Void tags (
, <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. But did that fix break something else? 47 routes. 12 ORM models. 3 middleware functions. Clicking through every page takes an hour. Running the test suite takes 2 seconds.

```
npm test

Running tests...

  add
    + add([[5, 3]]) == 8
    + add([[null]]) raises Error

  isEven
    + isEven([[4]]) is truthy
    + isEven([[3]]) is falsy

  4 tests: 4 passed, 0 failed, 0 errors
```

Everything still works. Deploy with confidence. Weekend intact.

Tina4 ships an inline testing framework. No external packages. No Jest configuration. No setup ceremony.

---

2. Your First Test

Tina4's testing framework uses a decorator-style pattern. Attach test assertions directly to functions using `tests()`, `assertEqual()`, `assertThrows()`, `assertTrue()`, and `assertFalse()`. Then call `runAllTests()` to execute them.

Create `tests/basic.ts`:

```
import { tests, assertEqual, assertThrows, assertTrue, assertFalse, runAllTests } from "tina4-no

const add = tests(
    assertEqual([5, 3], 8),
    assertEqual([0, 0], 0),
    assertThrows(Error, [null]),
)(function add(a: number, b: number | null = null): number {
    if (b === null) throw new Error("b required");
    return a + b;
});

const isEven = tests(
    assertTrue([4]),
    assertFalse([3]),
)(function isEven(n: number): boolean {
    return n % 2 === 0;
});

runAllTests();
```

Run it:

```
tina4 test

  add
    + add([[5, 3]]) == 8
    + add([[0, 0]]) == 0
    + add([[null]]) raises Error

  isEven
    + isEven([[4]]) is truthy
    + isEven([[3]]) is falsy

  5 tests: 5 passed, 0 failed, 0 errors
```

How It Works

- The `tests()` function takes assertion objects and returns a decorator.
- The decorator wraps the function, registers it in the test registry, and returns the original function unchanged.
- The function works normally in production code -- tests only execute when you call `runAllTests()`.
- Named functions produce readable output. Anonymous functions show as "anonymous."

---

3. Assertion Functions

Function	Description
`assertEqual(args, expected)`	Call the function with `args` array, expect `expected` as the return value
`assertThrows(ErrorClass, args)`	Call the function with `args` array, expect it to throw `ErrorClass`
`assertTrue(args)`	Call the function with `args` array, expect a truthy return value
`assertFalse(args)`	Call the function with `args` array, expect a falsy return value

Each assertion specifies the arguments to pass and the expected outcome. The `args` parameter is always an array of arguments.

assertEqual

```
assertEqual([5, 3], 8)     // Call with 5, 3 -- expect 8
assertEqual(["hello"], 5)  // Call with "hello" -- expect 5
```

assertThrows

```
assertThrows(Error, [null])         // Expect Error when called with null
assertThrows(TypeError, ["bad"])    // Expect TypeError when called with "bad"
```

assertTrue / assertFalse

```
assertTrue([4])     // Expect a truthy return value
assertFalse([0])    // Expect a falsy return value
```

---

4. Testing Business Logic

```
import { tests, assertEqual, assertThrows, runAllTests } from "tina4-nodejs";

const calculateDiscount = tests(
    assertEqual([100, 10], 90),
    assertEqual([50, 0], 50),
    assertEqual([200, 50], 100),
    assertThrows(Error, [100, -5]),
    assertThrows(Error, [100, 101]),
)(function calculateDiscount(price: number, discountPercent: number): number {
    if (discountPercent < 0 || discountPercent > 100) {
        throw new Error("Discount must be between 0 and 100");
    }
    return price - (price * discountPercent / 100);
});

runAllTests();
```

The function works in production. The tests run only when you call `runAllTests()`.

---

5. Testing ORM Models

Test your models by writing functions that exercise create, read, update, and delete:

```
import { tests, assertTrue, assertEqual, runAllTests } from "tina4-nodejs";
import { Database } from "tina4-nodejs/orm";

const testCreateProduct = tests(
    assertTrue([]),
)(function testCreateProduct(): boolean {
    const product = new Product({
        name: "Test Widget",
        category: "Testing",
        price: 19.99,
    });
    product.save();
    return product.id !== undefined && product.id > 0;
});

const testLoadProduct = tests(
    assertTrue([]),
)(function testLoadProduct(): boolean {
    const product = new Product({ name: "Load Test", price: 29.99 });
    product.save();

    const loaded = Product.findById(product.id);
    return loaded !== null && loaded.name === "Load Test";
});
```

```
const testUpdateProduct = tests(
    assertTrue([]),
)(function testUpdateProduct(): boolean {
    const product = new Product({ name: "Update Test", price: 10 });
    product.save();

    product.name = "Updated Widget";
    product.price = 15;
    product.save();

    const reloaded = Product.findById(product.id);
    return reloaded !== null && reloaded.name === "Updated Widget" && reloaded.price === 15;
});

const testDeleteProduct = tests(
    assertTrue([]),
)(function testDeleteProduct(): boolean {
    const product = new Product({ name: "Delete Me", price: 5 });
    product.save();
    const id = product.id;

    product.delete();

    const gone = Product.findById(id);
    return gone === null;
});

runAllTests();
```

Test Database

Use a separate database for tests so development data stays safe:

```
TINA4_DATABASE_URL=sqlite:///data/test.db
```

---

6. Testing Routes

Write functions that exercise your API endpoints end-to-end:

```
import { tests, assertTrue, assertEqual, runAllTests } from "tina4-nodejs";

const testHealthEndpoint = tests(
    assertTrue([]),
)(function testHealthEndpoint(): boolean {
    // Use the test client to send HTTP requests
    const resp = testClient.get("/health");
    return resp.status === 200 && resp.body.status === "ok";
});

const testCreateProduct = tests(
    assertTrue([]),
)(function testCreateProduct(): boolean {
    const resp = testClient.post("/api/products", {
        name: "Route Test Product",
        category: "Testing",
        price: 42,
    });
```

```
    return resp.status === 201 && resp.body.name === "Route Test Product";
});

const testGetNotFound = tests(
    assertTrue([]),
)(function testGetNotFound(): boolean {
    const resp = testClient.get("/api/products/99999");
    return resp.status === 404;
});

runAllTests();
```

---

7. Testing Authentication

```
import { tests, assertTrue, runAllTests, Auth } from "tina4-nodejs";

const secret = "test-secret";

const testTokenRoundTrip = tests(
    assertTrue([{ userId: 1, role: "admin" }]),
)(function testTokenRoundTrip(payload: Record<string, unknown>): boolean {
    const token = Auth.getToken(payload, secret);
    const decoded = Auth.validToken(token, secret);
    return decoded !== null && decoded.userId === payload.userId;
});

const testPasswordHash = tests(
    assertTrue(["securePass123"]),
    assertTrue(["another-password"]),
)(function testPasswordHash(password: string): boolean {
    const hash = Auth.hashPassword(password);
    return Auth.checkPassword(password, hash);
});

const testInvalidToken = tests(
    assertTrue([]),
)(function testInvalidToken(): boolean {
    const decoded = Auth.validToken("invalid.token.here", secret);
    return decoded === null;
});

runAllTests();
```

---

8. Resetting Tests Between Files

When running tests across multiple files, use `resetTests()` to clear the registry:

```
import { resetTests, tests, assertEqual, runAllTests } from "tina4-nodejs";

resetTests();

const multiply = tests(
    assertEqual([3, 4], 12),
    assertEqual([0, 5], 0),
```

```
)(function multiply(a: number, b: number): number {
    return a * b;
});

runAllTests();
```

Without resetting, tests from previously imported modules accumulate in the registry.

---

9. Runner Options

`runAllTests()` accepts an options object:

```
// Quiet mode -- no console output, returns results only
const results = runAllTests({ quiet: true });
console.log(`${results.passed} passed, ${results.failed} failed`);

// Fail fast -- stop on the first failure
runAllTests({ failfast: true });
```

The returned `TestResults` object:

Property	Type	Description
`passed`	number	Assertions that passed
`failed`	number	Assertions that failed
`errors`	number	Unexpected errors
`details`	array	Array of `{ name, status, message? }` objects

CI Integration

Use the return value for CI exit codes:

```
const results = runAllTests();
process.exit(results.failed === 0 ? 0 : 1);
```

---

10. Running Tests

Run All Tests

```
npm test
```

This runs `tests/run-all.ts` which discovers and executes all test files.

Run a Specific Test File

```
tina4 test tests/basic.ts
```

With npm Scripts

Add to `package.json`:

```
{
  "scripts": {
    "test": "npx tsx tests/run-all.ts",
    "test:products": "npx tsx tests/products.ts",
    "test:auth": "npx tsx tests/auth.ts"
  }
}
```

---

11. Testing Best Practices

Test One Thing Per Function

Each test function should verify one behavior. When it fails, you know exactly what broke.

```
// Good: each function tests one thing
const testCreateReturnsId = tests(
    assertTrue([]),
)(function testCreateReturnsId(): boolean {
    const product = new Product({ name: "Widget", price: 9.99 });
    product.save();
    return product.id > 0;
});

const testCreateSetsDefaults = tests(
    assertTrue([]),
)(function testCreateSetsDefaults(): boolean {
    const product = new Product({ name: "Widget", price: 9.99 });
    product.save();
    return product.category === "Uncategorized";
});
```

Use Named Functions for Readable Output

```
// Good: readable test names
const testLoginRejects = tests(assertTrue([]))(
    function testLoginRejectsInvalidPassword(): boolean { /* ... */ }
);

// Bad: anonymous function
const test = tests(assertTrue([]))(
    () => { /* shows as "anonymous" in output */ }
);
```

Isolate Tests

Each test should create its own data. Never depend on data from another test or from the development database.

```
// Good: creates its own data
function testDeleteProduct(): boolean {
    const product = new Product({ name: "Temporary", price: 1 });
    product.save();
    product.delete();
    return Product.findById(product.id) === null;
}

// Bad: depends on external state
```

```
function testDeleteProduct(): boolean {
    const product = Product.findById(1); // Assumes ID 1 exists
    product.delete();
    return true;
}
```

Use Unique Values

Prevent unique constraint failures by generating unique data:

```
function testCreateUser(): boolean {
    const email = `test-${Date.now()}@example.com`;
    const user = new User({ name: "Test User", email });
    user.save();
    return user.id > 0;
}
```

---

12. Failed Test Output

When a test fails, the output shows exactly what went wrong:

```
  calculateDiscount
    + calculateDiscount([[100, 10]]) == 90
    - calculateDiscount([[200, 50]]) == 100
      Expected: 100
      Got: 150
    + calculateDiscount([[100, -5]]) raises Error

  3 tests: 2 passed, 1 failed, 0 errors
```

The failure message shows the function name, the arguments, what was expected, and what was received. Find the line. Fix the logic. Run again.

---

13. Exercise: Write Tests for Utility Functions

Write inline tests for the following functions:

- A `slugify` function that converts `"Hello World"` to `"hello-world"`
- A `clamp` function that constrains a number between a min and max
- A `parsePrice` function that extracts a number from `"$19.99"` and throws on invalid input

---

14. Solution

```
import { tests, assertEqual, assertThrows, runAllTests } from "tina4-nodejs";

const slugify = tests(
    assertEqual(["Hello World"], "hello-world"),
    assertEqual(["  Multiple   Spaces  "], "multiple-spaces"),
    assertEqual(["UPPERCASE"], "uppercase"),
    assertEqual(["already-slugged"], "already-slugged"),
)(function slugify(input: string): string {
    return input.trim().toLowerCase().replace(/\s+/g, "-");
});

const clamp = tests(
    assertEqual([5, 0, 10], 5),
    assertEqual([-5, 0, 10], 0),
    assertEqual([15, 0, 10], 10),
    assertEqual([0, 0, 0], 0),
)(function clamp(value: number, min: number, max: number): number {
    return Math.min(Math.max(value, min), max);
});

const parsePrice = tests(
    assertEqual(["$19.99"], 19.99),
    assertEqual(["$0.50"], 0.5),
    assertThrows(Error, ["not-a-price"]),
    assertThrows(Error, [""]),
)(function parsePrice(input: string): number {
    const match = input.match(/\$?([\d.]+)/);
    if (!match) throw new Error("Invalid price format");
    const value = parseFloat(match[1]);
    if (isNaN(value)) throw new Error("Invalid price format");
    return value;
});

runAllTests();
```

---

15. Gotchas

1. The `tests()` Decorator Returns the Original Function

The wrapped function works identically in production. Tests only run when you call `runAllTests()`. You can use the decorated function in your application code without side effects.

2. Arguments Are Passed as an Array

`assertEqual([5, 3], 8)` means "call the function with arguments `5` and `3`, expect `8`." The first argument is always an array.

3. Named Functions Are Required for Readable Output

Anonymous functions show up as "anonymous" in test output. Use named function expressions: `function testCreateProduct() {}` not `() => {}`.

4. Call `resetTests()` Between Separate Test Files

Without resetting, tests from previously imported modules accumulate in the registry. Each test file should call `resetTests()` at the top.

5. `runAllTests()` Returns Results

Use the return value for CI integration. Check `results.failed === 0` to determine the exit code. Without this, your CI pipeline passes even when tests fail.

6. Database State Carries Between Tests

Problem: A test fails because a previous test left data in the database.

Fix: Each test should create its own data and clean up after itself. Use a separate test database. Reset it before each test run.

7. Async Functions Need Special Handling

Problem: You test an async function but the test finishes before the Promise resolves.

Fix: Wrap async operations in a synchronous test function that blocks on the result. Or structure your test to validate the return value of the resolved Promise.

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.

```
tina4nodejs generate crud Product --fields "name:string,price:float"
```

That single command creates six files:

#	File	Purpose
1	`src/orm/Product.ts`	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.ts`	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	`test/products.test.ts`	Test stubs for all CRUD operations

What Each File Contains

The **model** maps the `product` table to a TypeScript class:

```
import { ORM } from "tina4-nodejs";

export class Product extends ORM {
  tableName = "product";

  id?: number;
```

```
  name?: string;
  price?: number;
  createdAt?: string;
  updatedAt?: string;
}
```

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:

```
import { get, post, put, del } from "tina4-nodejs";
import { Product } from "../orm/Product";

get("/api/products", "List all products", async (request, response) => {
  const products = await new Product().select();
  return response(products);
});

get("/api/products/:id", "Get a product by ID", async (request, response) => {
  const product = new Product();
  product.id = request.params.id;
  await product.load();
  return response(product);
});

post("/api/products", "Create a product", async (request, response) => {
  const product = new Product(request.body);
  await product.save();
  return response(product, 201);
});

put("/api/products/:id", "Update a product", async (request, response) => {
  const product = new Product(request.body);
  product.id = request.params.id;
  await product.save();
  return response(product);
});

del("/api/products/:id", "Delete a product", async (request, response) => {
  const product = new Product();
  product.id = request.params.id;
  await product.delete();
  return response(null, 204);
});
```

The **form template** renders typed inputs with CSRF protection:

```
<form method="POST" action="/api/products">
    <input type="hidden" name="form_token" value="{{ form_token }}">
    <label>Name</label>
    <input type="text" name="name" required>
    <label>Price</label>
    <input type="number" step="0.01" name="price" required>
    <button type="submit">Save</button>
</form>
```

The **test file** stubs out CRUD assertions using the built-in test runner:

```
import { describe, it } from "node:test";
import assert from "node:assert";
import { App } from "tina4-nodejs";

const client = new App().testClient();

describe("Products API", () => {
  it("creates a product", async () => {
    const response = await client.post("/api/products", { name: "Widget", price: 9.99 });
    assert.strictEqual(response.statusCode, 201);
  });

  it("lists products", async () => {
    const response = await client.get("/api/products");
    assert.strictEqual(response.statusCode, 200);
  });

  it("gets a product", async () => {
    const response = await client.get("/api/products/1");
    assert.strictEqual(response.statusCode, 200);
  });

  it("updates a product", async () => {
    const response = await client.put("/api/products/1", { name: "Updated Widget" });
    assert.strictEqual(response.statusCode, 200);
  });

  it("deletes a product", async () => {
    const response = await client.delete("/api/products/1");
    assert.strictEqual(response.statusCode, 204);
  });
});
```

Run It

After generating, run the migration and start the server:

```
tina4nodejs migrate
tina4nodejs serve
```

Open Swagger UI at `http://localhost:7148/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

```
tina4nodejs generate model Product --fields "name:string,price:float"
```

Creates three files: the ORM model (`src/orm/Product.ts`), the UP migration, and the DOWN migration. No routes, no templates, no tests.

Route

```
tina4nodejs generate route products --model Product
```

Creates one file: `src/routes/products.ts` with CRUD endpoints and Swagger annotations. The model must exist first.

Migration

```
tina4nodejs 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

```
tina4nodejs generate middleware AuthLog
```

Creates one file with before and after stubs:

```
import { middleware } from "tina4-nodejs";

middleware("AuthLogBefore", "before", async (request) => {
  console.log(`Request: ${request.method} ${request.url}`);
  return request;
});

middleware("AuthLogAfter", "after", async (request, response) => {
  console.log(`Response: ${response.statusCode}`);
  return response;
});
```

Test

```
tina4nodejs generate test products --model Product
```

Creates one file: `test/products.test.ts` with CRUD stubs using the built-in test runner.

Form

```
tina4nodejs generate form Product --fields "name:string,price:float"
```

Creates one file: `src/templates/products/form.html` with typed inputs and `form_token`.

View

```
tina4nodejs generate view Product --fields "name:string,price:float"
```

Creates two templates: a list view and a detail view in `src/templates/products/`.

CRUD

```
tina4nodejs generate crud Product --fields "name:string,price:float"
```

Shorthand for running all generators at once: model, migration, route, form, view, and test.

Auth

```
tina4nodejs 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

```
// AutoCRUD routes are auto-generated from discovered models
// Models in src/models/ get REST endpoints at /api/{tableName}
```

Place your ORM models in `src/models/` and the framework discovers and mounts all five standard endpoints automatically — no route files needed.

The Auth Generator

Authentication needs more than one file. The auth generator creates seven:

```
tina4nodejs generate auth
```

#	File	Purpose
1	`src/orm/User.ts`	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.ts`	Login, register, logout routes
5	`src/templates/auth/login.html`	Login form
6	`src/templates/auth/register.html`	Registration form
7	`test/auth.test.ts`	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:

```
tina4nodejs migrate
tina4nodejs 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:

```
tina4nodejs generate model Product --fields "name:string,price:float" --with-route --with-migrat
```

Available flags:

Flag	Adds
`--with-route`	CRUD route file
`--with-migration`	Migration files (included by default with model)
`--with-test`	Test file
`--with-form`	Form template
`--with-view`	View templates

The `generate crud` command is equivalent to using all `--with` flags at once.

Exercise: Scaffold a Blog

Build a blog with three resources using generators.

Step 1: Generate the auth system.

```
tina4nodejs generate auth
```

Step 2: Scaffold the Post resource.

```
tina4nodejs generate crud Post --fields "title:string,body:text,published:bool"
```

Step 3: Scaffold the Category resource.

```
tina4nodejs generate crud Category --fields "name:string,description:text"
```

Step 4: Add a migration to link posts to categories.

```
tina4nodejs 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.

```
tina4nodejs migrate
tina4nodejs 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 `tina4nodejs migrate`. Generate and migrate are separate steps by design.

File Naming Matters

The generator derives file names from the model name. `Product` becomes `Product.ts` for the model and `products.ts` for routes. Do not rename generated files unless you update all imports.

Field Changes Need New Migrations

Changing `--fields` and re-running the generator does not update existing migrations. Create a new migration with `generate migration` and write the ALTER TABLE by hand.

Singular Table Names

Tina4 uses singular table names: `product`, not `products`. The route paths use plural (`/api/products`), but the table stays singular. The generator handles this split.

npx Alternative

If `tina4nodejs` is not in your PATH, use `npx`:

```
npx tina4nodejs generate crud Product --fields "name:string,price:float"
```

All generator commands work the same way through npx.

API Documentation with Swagger

1. The 47-Endpoint Problem

Your team builds 47 API endpoints. The frontend developer asks "what does this endpoint accept?" Again. And again.

Swagger kills that question. It generates interactive API documentation from annotations in your route files. The docs stay current because they live in the code. Change a handler. The documentation changes with it.

Tina4 builds a Swagger UI at `/swagger` from JSDoc comments on your routes. No build step. No extra tooling. No separate spec file to maintain.

---

2. What Swagger/OpenAPI Is

OpenAPI is a specification format for describing REST APIs. Swagger is the toolset that reads OpenAPI specs and renders interactive documentation. Tina4 scans your route files, reads JSDoc annotations, and builds the OpenAPI spec automatically. No manual spec writing.

The generated spec is standard OpenAPI 3.0. It works with every tool in the OpenAPI ecosystem: client SDK generators, testing tools, API gateways, and monitoring services.

---

3. Enabling Swagger

Swagger runs when `TINA4_DEBUG=true`. Navigate to:

```
http://localhost:7148/swagger
```

The interactive UI loads. Every annotated route appears. Click any endpoint to see its documentation. Click "Try it out" to send a real request.

The raw OpenAPI JSON spec lives at:

```
http://localhost:7148/swagger/json
```

Swagger in Production

By default, Swagger only shows in debug mode. To expose it in production:

```
TINA4_SWAGGER_ENABLED=true
```

This enables the Swagger UI even when `TINA4_DEBUG=false`. Useful for developer portals and partner integrations.

---

4. Adding Descriptions to Routes

Every JSDoc annotation starts with a summary line. The `@description` tag adds detail:

```
import { Router } from "tina4-nodejs";

/**
 * List all products
 * @description Returns a paginated list of all products in the catalog
 * @tags Products
 */
Router.get("/api/products", async (req, res) => {
    return res.json({ products: [] });
});
```

The first line ("List all products") becomes the endpoint summary in Swagger. The `@description` text appears when a user expands the endpoint.

Path Parameters

Use `@param` to document path parameters:

```
/**
 * Get a product by ID
 * @description Returns a single product with full details
 * @tags Products
 * @param int id The unique product identifier
 */
Router.get("/api/products/{id:int}", async (req, res) => {
    return res.json({ id: req.params.id, name: "Wireless Keyboard", price: 79.99 });
});
```

Query Parameters

Use `@query` to document query string parameters:

```
/**
 * Search products
 * @description Search the product catalog by name or category
 * @tags Products
 * @query string q Search query
 * @query string category Filter by category
 * @query int page Page number (default: 1)
 * @query int limit Items per page (default: 20)
 */
Router.get("/api/products/search", async (req, res) => {
    return res.json({ query: req.query.q, results: [], total: 0 });
});
```

Each `@query` tag specifies the type, name, and description. Swagger renders these as a form in the "Try it out" view.

---

5. Documenting Request Bodies

Use `@body` to describe what the endpoint accepts:

```
/**
```

```
 * Create a new product
 * @description Creates a product in the catalog
 * @tags Products
 * @body {"name": "string", "category": "string", "price": "float", "inStock": "bool"}
 * @response 201 {"id": "int", "name": "string", "category": "string", "price": "float"}
 * @response 400 {"error": "string"}
 */
Router.post("/api/products", async (req, res) => {
    if (!req.body.name) {
        return res.status(400).json({ error: "Name is required" });
    }
    return res.status(201).json({
        id: 1,
        name: req.body.name,
        category: req.body.category ?? "Uncategorized",
        price: parseFloat(req.body.price ?? 0),
    });
});
```

The `@body` tag defines the JSON schema. Keys are field names. Values are type strings. Swagger renders this as a schema in the request body section.

Supported Types in @body and @response

Type String	OpenAPI Type
`"string"`	string
`"int"`	integer
`"float"`	number (float)
`"number"`	number
`"bool"`	boolean
`"object"`	object
`"array"`	array

---

6. Documenting Responses

Use `@response` to describe each possible response:

```
/**
 * Update a product
 * @tags Products
 * @param int id Product ID
 * @body {"name": "string", "price": "float"}
 * @response 200 {"id": "int", "name": "string", "price": "float", "updatedAt": "string"}
 * @response 404 {"error": "string"}
 * @response 400 {"error": "string"}
 */
Router.put("/api/products/{id:int}", async (req, res) => {
    // ...
});
```

Each `@response` tag starts with the status code, followed by the JSON schema. Document every status code your endpoint can return. The frontend developer sees exactly what to expect for success and failure.

---

7. Tags for Grouping

Tags organize endpoints into sections in the Swagger UI:

```
/**
 * List all users
 * @tags Users
 */
Router.get("/api/users", async (req, res) => {
    return res.json({ users: [] });
});

/**
 * List all orders
 * @tags Orders
 */
Router.get("/api/orders", async (req, res) => {
    return res.json({ orders: [] });
});
```

All endpoints with `@tags Users` appear under the "Users" heading in Swagger. An endpoint can belong to multiple tags:

```
/**
 * List user orders
 * @tags Users, Orders
 */
```

Without tags, endpoints land in a "default" group. Always add tags. They make the documentation navigable.

---

8. Example Values

Static schemas tell the developer what type each field is. Examples show what real data looks like:

```
/**
 * Create a product
 * @tags Products
 * @body {"name": "string", "category": "string", "price": "float"}
 * @example request {"name": "Ergonomic Keyboard", "category": "Electronics", "price": 89.99}
 * @example response {"id": 42, "name": "Ergonomic Keyboard", "category": "Electronics", "price"
 */
Router.post("/api/products", async (req, res) => {
    return res.status(201).json({ id: 42, name: req.body.name, price: req.body.price });
});
```

`@example request` pre-fills the request body in "Try it out." The developer clicks the button and the form already has realistic data. No manual typing.

`@example response` shows what a successful response looks like. The Swagger UI displays it in the response section.

---

9. Authentication in Swagger

Public and Secured Routes

Mark routes as public or secured with `@noauth` and `@secured`:

```
/**
 * Login
 * @noauth
 * @tags Auth
 * @body {"email": "string", "password": "string"}
 * @response 200 {"token": "string"}
 */
Router.post("/api/auth/login", async (req, res) => {
    // ...
});

/**
 * Get user profile
 * @secured
 * @tags Users
 * @response 200 {"id": "int", "name": "string", "email": "string"}
 * @response 401 {"error": "string"}
 */
Router.get("/api/profile", async (req, res) => {
    // ...
}, [authMiddleware]);
```

`@noauth` tells Swagger this endpoint does not require authentication. `@secured` adds a lock icon and includes the Authorization header in the spec.

Authorize Button

The Swagger UI has an "Authorize" button at the top. Click it. Paste your JWT token. All subsequent "Try it out" requests include the `Authorization: Bearer <token>` header automatically.

Workflow

- Call the login endpoint in Swagger -- get a token in the response
- Click the "Authorize" button at the top
- Paste the token value
- All secured endpoints now send the token with every request

No curl. No Postman. Test the full auth flow from the browser.

---

10. Try-It-Out from the Swagger UI

The Swagger UI puts a "Try it out" button on every endpoint. Click it. The UI expands the endpoint into a form. Fill in path parameters, query parameters, and the request body. Click "Execute." The UI sends a real HTTP request to your running server and displays the response.

The response panel shows:

- The HTTP status code
- Response headers
- The response body (formatted JSON)
- The curl command equivalent

This replaces curl for development testing. See the request. See the response. Adjust and try again. All in the browser.

---

11. A Complete Documented API

Here is a full product API with complete Swagger annotations:

```
import { Router } from "tina4-nodejs";

/**
 * List all products
 * @description Returns a paginated list of products. Supports filtering by category and sorting
 * @tags Products
 * @query string category Filter by category name
 * @query string sort Sort field (name, price, created_at)
 * @query string order Sort direction (ASC or DESC)
 * @query int page Page number (default: 1)
 * @query int limit Items per page (default: 20)
 * @response 200 {"products": [{"id": "int", "name": "string", "category": "string", "price": "f
 * @example response {"products": [{"id": 1, "name": "Keyboard", "category": "Electronics", "pri
 */
Router.get("/api/products", async (req, res) => {
    // implementation
});

/**
 * Get a product by ID
 * @description Returns full product details including category and stock status
 * @tags Products
 * @param int id Product ID
 * @response 200 {"id": "int", "name": "string", "category": "string", "price": "float", "inStoc
 * @response 404 {"error": "string"}
 * @example response {"id": 1, "name": "Wireless Keyboard", "category": "Electronics", "price":
 */
Router.get("/api/products/{id:int}", async (req, res) => {
    // implementation
});

/**
 * Create a product
```

```
 * @description Adds a new product to the catalog
 * @secured
 * @tags Products
 * @body {"name": "string", "category": "string", "price": "float", "inStock": "bool"}
 * @response 201 {"id": "int", "name": "string", "category": "string", "price": "float"}
 * @response 400 {"error": "string"}
 * @response 401 {"error": "string"}
 * @example request {"name": "Ergonomic Mouse", "category": "Electronics", "price": 49.99, "inSt
 * @example response {"id": 42, "name": "Ergonomic Mouse", "category": "Electronics", "price": 4
 */
Router.post("/api/products", async (req, res) => {
    // implementation
}, [authMiddleware]);

/**
 * Update a product
 * @description Updates an existing product. Only include fields you want to change.
 * @secured
 * @tags Products
 * @param int id Product ID
 * @body {"name": "string", "category": "string", "price": "float", "inStock": "bool"}
 * @response 200 {"id": "int", "name": "string", "category": "string", "price": "float"}
 * @response 404 {"error": "string"}
 * @response 401 {"error": "string"}
 * @example request {"price": 69.99}
 * @example response {"id": 1, "name": "Wireless Keyboard", "category": "Electronics", "price":
 */
Router.put("/api/products/{id:int}", async (req, res) => {
    // implementation
}, [authMiddleware]);

/**
 * Delete a product
 * @description Removes a product from the catalog
 * @secured
 * @tags Products
 * @param int id Product ID
 * @response 204
 * @response 404 {"error": "string"}
 * @response 401 {"error": "string"}
 */
Router.delete("/api/products/{id:int}", async (req, res) => {
    // implementation
}, [authMiddleware]);
```

---

12. Hiding Routes

Use `@hidden` to exclude a route from Swagger:

```
/**
 * Internal health check
 * @hidden
 */
Router.get("/internal/ping", async (req, res) => {
    return res.json({ pong: true });
});
```

The route still works. It does not appear in the Swagger UI or the JSON spec.

---

13. Customizing the Swagger Info Block

Configure the API metadata through `.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
```

These values appear in the header of the Swagger UI. Set them once. Every endpoint inherits the context.

---

14. Generating Client SDKs

The OpenAPI spec at `/swagger/json` feeds into code generators. Generate a typed API client for your frontend:

```
npm install -g @openapitools/openapi-generator-cli

openapi-generator-cli generate \
  -i http://localhost:7148/swagger/json \
  -g typescript-fetch \
  -o ./frontend/api-client
```

This produces a fully typed TypeScript client. Every endpoint becomes a method. Every request body and response has type definitions. The frontend developer gets autocompletion and type checking from your Swagger annotations.

Available Generators

Generator	Output
`typescript-fetch`	TypeScript with Fetch API
`typescript-axios`	TypeScript with Axios
`javascript`	Plain JavaScript
`python`	Python client
`java`	Java client
`swift5`	iOS Swift client
`kotlin`	Android Kotlin client
`csharp`	C# .NET client

One spec. Any language. The API documentation becomes a contract between your backend and every frontend that consumes it.

---

15. Auto-CRUD Routes in Swagger

Models with `static autoCrud = true` generate Swagger documentation automatically. The generated routes include:

- Summary and description
- Path parameters
- Query parameters for the list endpoint (page, limit, sort, order, filters)
- Request body schema from the model's `fields` definition
- Response schemas

No JSDoc annotations needed for auto-CRUD routes. The ORM model definition drives the documentation.

---

16. Exercise: Document a Complete User API

Document a User API with the following endpoints. Include full Swagger annotations: `@param`, `@query`, `@body`, `@response`, `@example`, `@tags`, and `@secured` where appropriate.

Requirements

- `PUT /api/users/{id}` -- Update user profile
- `GET /api/users/{id}/orders` -- List a user's orders (with status filter and pagination)
- `POST /api/users/{id}/avatar` -- Upload a user avatar URL

---

17. Solution

```
import { Router } from "tina4-nodejs";

/**
 * Update a user
 * @description Updates an existing user's profile information. Only send fields you want to cha
 * @secured
 * @tags Users
 * @param int id User ID
 * @body {"name": "string", "email": "string", "role": "string"}
 * @response 200 {"id": "int", "name": "string", "email": "string", "role": "string", "updatedAt
 * @response 404 {"error": "string"}
 * @response 401 {"error": "string"}
 * @example request {"name": "Alice Smith", "email": "alice.smith@example.com"}
 * @example response {"id": 1, "name": "Alice Smith", "email": "alice.smith@example.com", "role"
 */
Router.put("/api/users/{id:int}", async (req, res) => {
    return res.json({ id: req.params.id, name: req.body.name ?? "Alice", updatedAt: new Date().t
});

/**
 * List user orders
 * @description Returns a paginated list of orders for a specific user. Filter by status to see
 * @secured
 * @tags Users, Orders
 * @param int id User ID
 * @query string status Filter by order status (pending, shipped, delivered)
 * @query int page Page number (default: 1)
 * @query int limit Items per page (default: 20)
 * @response 200 {"orders": [{"id": "int", "product": "string", "total": "float", "status": "str
 * @response 404 {"error": "string"}
 * @example response {"orders": [{"id": 101, "product": "Wireless Keyboard", "total": 79.99, "st
 */
Router.get("/api/users/{id:int}/orders", async (req, res) => {
    return res.json({ orders: [], total: 0, page: parseInt(req.query.page ?? "1", 10) });
});

/**
 * Upload user avatar
 * @description Sets or updates the avatar URL for a user. The URL must point to an accessible i
 * @secured
 * @tags Users
 * @param int id User ID
 * @body {"avatarUrl": "string"}
 * @response 200 {"id": "int", "avatarUrl": "string", "updatedAt": "string"}
 * @response 400 {"error": "string"}
 * @response 404 {"error": "string"}
 * @example request {"avatarUrl": "https://cdn.example.com/avatars/alice.jpg"}
 * @example response {"id": 1, "avatarUrl": "https://cdn.example.com/avatars/alice.jpg", "update
 */
Router.post("/api/users/{id:int}/avatar", async (req, res) => {
    if (!req.body.avatarUrl) {
        return res.status(400).json({ error: "avatarUrl is required" });
    }
    return res.json({ id: req.params.id, avatarUrl: req.body.avatarUrl, updatedAt: new Date().to
});
```

---

18. Gotchas

1. Annotations Must Be Directly Above the Route

Problem: Swagger does not pick up your annotations.

Cause: A blank line or other code sits between the JSDoc comment and the `Router.get(...)` call.

Fix: The `*/` closing must be on the line directly before the Router call. No blank lines between them.

2. Missing @tags Makes Endpoints Hard to Find

Problem: All endpoints land in a "default" group in the Swagger UI.

Fix: Add `@tags ResourceName` to every route doc-block. Group related endpoints together.

3. @body Must Be Valid JSON

Problem: Swagger shows an error parsing the body schema.

Fix: Every key and string value must be in double quotes. `{"name": "string"}` works. `{name: string}` fails.

4. Swagger Shows Routes You Did Not Annotate

Problem: Un-annotated routes appear in Swagger with no documentation.

Fix: By design -- Tina4 shows all registered routes. Add `@hidden` to hide a route from Swagger.

5. Response Examples Do Not Match Actual Responses

Problem: The Swagger example shows one format but the actual response has different fields.

Fix: Update annotations when you change the handler's response format. Examples are static text in your source code. They do not update automatically.

6. Swagger UI Not Available in Production

Problem: Navigating to `/swagger` returns a 404 in production.

Fix: Set `TINA4_SWAGGER_ENABLED=true` in `.env` to enable Swagger in production. Without it, Swagger only runs when `TINA4_DEBUG=true`.

7. SDK Generation Produces Incorrect Types

Problem: The generated TypeScript client has `any` types instead of proper types.

Fix: Use correct OpenAPI type strings in your annotations: `"string"`, `"int"`, `"float"`, `"bool"`. Avoid using TypeScript type syntax -- Swagger annotations use their own type vocabulary.

8. @secured Routes Not Sending Auth Token

Problem: "Try it out" sends requests without the Authorization header even though the route is marked `@secured`.

Fix: Click the "Authorize" button at the top of the Swagger UI. Paste your JWT token. All subsequent requests include the header. The token persists until you close the tab or click "Logout."

API Client

1. Calling External APIs Without the Boilerplate

Your application calls a payment gateway. A shipping provider. A weather service. A CRM. Every call needs the same setup: base URL, auth header, error handling, JSON parsing, timeout.

Tina4 provides an `api` singleton — a preconfigured HTTP client that handles the repetitive parts. It covers GET, POST, PUT, and DELETE with JSON serialization, auth headers set once at configure time, and a consistent response format, all with no external dependencies.

---

2. The api Singleton

Import `api` from `@tina4/core`:

```
import { api } from "@tina4/core";
```

`api` is a ready-to-use HTTP client. Configure it once. Use it everywhere.

---

3. Configuring the Client

Set the base URL and default headers before making requests. Do this at startup in `src/index.ts` or a dedicated `src/services/http.ts` file:

```
import { api } from "@tina4/core";

api.configure({
    baseUrl: "https://api.example.com/v2",
    headers: {
        "Authorization": `Bearer ${process.env.TINA4_API_KEY}`,
        "X-App-Version": "1.0.0"
    },
    timeout: 10000  // 10 seconds
});
```

Option	Default	Description
`baseUrl`	`""`	Prepended to every request path
`headers`	`{}`	Sent with every request
`timeout`	`30000`	Request timeout in milliseconds

---

4. GET Requests

```
import { api } from "@tina4/core";

const response = await api.get("/products");

if (response.ok) {
    const products = response.data;
    console.log(`Fetched ${products.length} products`);
} else {
    console.error(`Error ${response.status}: ${response.error}`);
}
```

Pass query parameters as the second argument:

```
const response = await api.get("/products", {
    params: { category: "Electronics", page: 2, limit: 20 }
});
// Requests: GET /products?category=Electronics&page=2&limit=20
```

---

5. POST Requests

```
import { api } from "@tina4/core";

const response = await api.post("/orders", {
    customerId: 15,
    items: [
        { sku: "KB-001", qty: 1, price: 79.99 },
        { sku: "HDMI-2M", qty: 2, price: 12.99 }
    ],
    shippingAddress: {
        line1: "123 Main St",
        city: "Springfield",
        country: "US"
    }
});

if (response.ok) {
    console.log(`Order created: ${response.data.orderId}`);
} else {
    console.error("Order failed:", response.error);
}
```

The body is serialized as JSON automatically. The `Content-Type: application/json` header is set for you.

---

6. PUT and PATCH Requests

```
import { api } from "@tina4/core";

// Replace the entire resource
const putResponse = await api.put("/products/42", {
    name: "Wireless Keyboard Pro",
    price: 89.99,
    inStock: true,
    category: "Electronics"
});

// Update specific fields only
const patchResponse = await api.patch("/products/42", {
    price: 74.99
});

if (putResponse.ok) {
    console.log("Product updated:", putResponse.data);
}
```

---

7. DELETE Requests

```
import { api } from "@tina4/core";

const response = await api.delete("/products/42");

if (response.ok) {
    console.log("Product deleted");
} else if (response.status === 404) {
    console.log("Product not found");
} else {
    console.error("Delete failed:", response.error);
}
```

---

8. Response Format

Every method returns the same response shape:

```
interface ApiResponse<T = unknown> {
    ok: boolean;          // true if status 200-299
    status: number;       // HTTP status code
    data: T;              // Parsed JSON body (on success)
    error: string | null; // Error message (on failure)
    headers: Record<string, string>;
}
```

Check `response.ok` before using `response.data`. On failure, `response.error` contains the error message and `response.data` may be `null`.

```
const response = await api.get("/products/999");

if (!response.ok) {
    if (response.status === 404) {
```

```
        return res.status(404).json({ error: "Product not found upstream" });
    }
    return res.status(502).json({ error: "Upstream service error" });
}

return res.json(response.data);
```

---

9. Per-Request Headers

Override or extend headers for a specific request:

```
import { api } from "@tina4/core";

// Override the Authorization header for this request only
const response = await api.get("/admin/users", {
    headers: {
        "Authorization": `Bearer ${adminToken}`,
        "X-Request-Id": requestId
    }
});
```

Per-request headers are merged with the configured defaults. The per-request value wins on conflict.

---

10. Using api Inside Route Handlers

Proxy or transform external API calls inside your routes:

```
import { Router } from "tina4-nodejs";
import { api } from "@tina4/core";

api.configure({
    baseUrl: "https://api.openweathermap.org/data/2.5",
    headers: { "Accept": "application/json" }
});

Router.get("/api/weather/{city}", async (req, res) => {
    const city = req.params.city;

    const response = await api.get("/weather", {
        params: {
            q: city,
            appid: process.env.OPENWEATHER_API_KEY,
            units: "metric"
        }
    });

    if (!response.ok) {
        return res.status(response.status).json({
            error: `Weather service error: ${response.error}`
        });
    }

    const weather = response.data as {
```

```
        name: string;
        main: { temp: number; humidity: number };
        weather: { description: string }[];
    };

    return res.json({
        city: weather.name,
        temperature: weather.main.temp,
        humidity: weather.main.humidity,
        description: weather.weather[0]?.description ?? "unknown"
    });
});
```

---

11. Exercise: GitHub User Profile Proxy

Build a route that fetches a GitHub user profile via the `api` client and returns a simplified version.

Requirements

- Configure `api` with `https://api.github.com` as the base URL and a `User-Agent` header (GitHub requires one)
- Create a `GET /api/github/{username}` route that fetches the user's public profile
- Return only: `login`, `name`, `public_repos`, `followers`, `following`, `bio`, and `avatar_url`
- Return `404` with a message if the GitHub user does not exist

Test with:

```
curl http://localhost:7148/api/github/torvalds
curl http://localhost:7148/api/github/this-user-does-not-exist-xyzabc
```

---

12. Solution

`src/services/github.ts`:

```
import { api } from "@tina4/core";

api.configure({
    baseUrl: "https://api.github.com",
    headers: {
        "User-Agent": "tina4-book-example/1.0",
        "Accept": "application/vnd.github+json"
    },
    timeout: 8000
});
```

`src/routes/github.ts`:

```
import { Router } from "tina4-nodejs";
import { api } from "@tina4/core";
import "../services/github";
```

```
interface GitHubUser {
    login: string;
    name: string | null;
    bio: string | null;
    public_repos: number;
    followers: number;
    following: number;
    avatar_url: string;
}

Router.get("/api/github/{username}", async (req, res) => {
    const { username } = req.params;

    const response = await api.get<GitHubUser>(`/users/${username}`);

    if (!response.ok) {
        if (response.status === 404) {
            return res.status(404).json({ error: `GitHub user '${username}' not found` });
        }
        return res.status(502).json({ error: "GitHub API error", detail: response.error });
    }

    const user = response.data;

    return res.json({
        login: user.login,
        name: user.name,
        bio: user.bio,
        public_repos: user.public_repos,
        followers: user.followers,
        following: user.following,
        avatar_url: user.avatar_url
    });
});
```

```
curl http://localhost:7148/api/github/torvalds
```

```
{
  "login": "torvalds",
  "name": "Linus Torvalds",
  "bio": null,
  "public_repos": 7,
  "followers": 234156,
  "following": 0,
  "avatar_url": "https://avatars.githubusercontent.com/u/1024025?v=4"
}
```

---

13. Gotchas

1. Configure before first use

`api.configure()` must be called before any request is made. If a module imports `api` and immediately calls `api.get()` at module load time before `configure()` runs, the request goes to an empty base URL.

Fix: Call `api.configure()` in your app entry point (`src/index.ts`) before importing any service modules that use `api`.

2. Swallowing upstream errors

Checking only `response.ok` hides the status code. A `401` and a `503` both set `ok: false`, but require different handling.

Fix: Always check `response.status` for error branches. A `401` means your API key is wrong. A `503` means the service is down. Treat them differently.

3. Timeout not set

The default timeout is 30 seconds. If the upstream API hangs, your route handler hangs too, holding a connection open.

Fix: Set an explicit timeout appropriate for your service SLA. For real-time endpoints, 5-10 seconds is usually the right upper bound.

4. Sending secrets in URLs

Appending API keys as query parameters (e.g., `?apikey=secret`) logs the key to access logs.

Fix: Pass secrets in headers (`Authorization`, `X-API-Key`). Configure them once with `api.configure({ headers: { ... } })`.

GraphQL and SOAP

1. The Problem GraphQL Solves

Your mobile app needs a list of products. Each product carries a name, price, 20 image URLs, a full description, 15 review objects, and 8 fields you do not need. REST sends all of it -- 50KB of JSON when you needed 2KB. On a mobile connection, that waste stings.

Now your web dashboard needs the same products plus category, stock status, and supplier info. REST forces you to make three requests and stitch them together, or build a custom endpoint with query parameter parsing.

GraphQL ends both problems. The client asks for the fields it needs. The server returns those fields. One endpoint. One request. One response shaped to fit.

Tina4 includes a built-in GraphQL engine. No external packages. No Apollo Server. No graphql-yoga. Part of the framework.

---

2. GraphQL vs REST -- A Quick Comparison

Aspect	REST	GraphQL
Endpoints	One per resource (`/products`, `/users`)	One endpoint (`/graphql`)
Data shape	Server decides	Client decides
Over-fetching	Common (get all fields)	Never (get only requested fields)
Under-fetching	Common (multiple requests needed)	Never (get related data in one query)
Versioning	`/api/v1/`, `/api/v2/`	Schema evolves, no versioning needed
Caching	HTTP caching works naturally	Requires client-side cache management
Learning curve	Low	Moderate

REST is still great for simple APIs. GraphQL shines when clients have diverse data needs -- mobile apps, dashboards, third-party integrations.

---

3. Enabling GraphQL

GraphQL is available by default in Tina4. The engine serves requests at `/graphql`. If you want to change the endpoint, set it in `.env`:

```
TINA4_GRAPHQL_ENDPOINT=/graphql
```

To verify it is working, start the server and send a test query:

```
curl -X POST http://localhost:7148/graphql \
  -H "Content-Type: application/json" \
  -d '{"query": "{ __schema { queryType { name } } }"}'
```

```
{
  "data": {
    "__schema": {
      "queryType": {
        "name": "Query"
      }
    }
  }
}
```

If you see that response, GraphQL is running.

---

4. Defining a Schema

A GraphQL schema defines the types of data your API can return and the queries and mutations clients can execute.

Create `src/graphql/schema.graphql`:

```
type Product {
    id: Int!
    name: String!
    category: String!
    price: Float!
    inStock: Boolean!
    createdAt: String
}

type Query {
    products: [Product!]!
    product(id: Int!): Product
}
```

This schema says:

- A `Product` has six fields. The `!` means the field is non-nullable.
- The `Query` type has two operations: `products` returns a list of products, and `product(id)` returns a single product (or null if not found).

Tina4 auto-discovers `.graphql` files in `src/graphql/`. You do not need to register them.

---

5. Writing Resolvers

A resolver is the function that runs when a query or mutation is executed. Resolvers live in `src/graphql/` as TypeScript files.

Create `src/graphql/resolvers.ts`:

```
import { GraphQL } from "tina4-nodejs";
import { Product } from "../orm/Product";

// Resolve the "products" query
GraphQL.resolve("Query", "products", async (root, args) => {
    const product = new Product();
    const products = await product.select("*", "", {}, "name ASC");
    return products.map(p => p.toDict());
});

// Resolve the "product" query
GraphQL.resolve("Query", "product", async (root, args) => {
    const product = new Product();
    await product.load(args.id);
    return product.id ? product.toDict() : null;
});
```

`GraphQL.resolve()` takes three arguments:

- **Type name** -- The GraphQL type this resolver belongs to (`Query`, `Mutation`, or a custom type).
- **Field name** -- The field within the type this resolver handles.
- **Resolver function** -- Receives `root` (the parent value, if any) and `args` (the query arguments).

Testing the Queries

List all products:

```
curl -X POST http://localhost:7148/graphql \
  -H "Content-Type: application/json" \
  -d '{"query": "{ products { id name price inStock } }"}'
```

```
{
  "data": {
    "products": [
      {"id": 1, "name": "Wireless Keyboard", "price": 79.99, "inStock": true},
      {"id": 2, "name": "USB-C Hub", "price": 49.99, "inStock": true},
      {"id": 3, "name": "Standing Desk", "price": 549.99, "inStock": false}
    ]
  }
}
```

Notice: the response contains only the four fields we asked for (`id`, `name`, `price`, `inStock`), not `category` or `createdAt`. That is GraphQL in action.

Get a single product with all fields:

```
curl -X POST http://localhost:7148/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:7148/graphql \
  -H "Content-Type: application/json" \
  -d '{"query": "{ product(id: 999) { id name } }"}'
```

```
{
  "data": {
    "product": null
  }
}
```

---

6. Mutations

Mutations are how clients create, update, or delete data. They are defined in the schema alongside queries.

Update `src/graphql/schema.graphql`:

```
type Product {
    id: Int!
    name: String!
    category: String!
    price: Float!
    inStock: Boolean!
    createdAt: String
}

input ProductInput {
    name: String!
    category: String
    price: Float!
    inStock: Boolean
}

input ProductUpdateInput {
    name: String
    category: String
    price: Float
    inStock: Boolean
}

type DeleteResult {
```

```
    success: Boolean!
    message: String!
}

type Query {
    products: [Product!]!
    product(id: Int!): Product
    productsByCategory(category: String!): [Product!]!
}

type Mutation {
    createProduct(input: ProductInput!): Product!
    updateProduct(id: Int!, input: ProductUpdateInput!): Product
    deleteProduct(id: Int!): DeleteResult!
}
```

Key concepts:

- `input` **types** define the shape of data the client sends. They are like regular types but used for arguments.
- `Mutation` **type** lists all write operations. By convention, mutations are named with verbs: `createProduct`, `updateProduct`, `deleteProduct`.
- Fields in `ProductUpdateInput` are all optional (no `!`) because the client may only want to update one field.

Mutation Resolvers

Add to `src/graphql/resolvers.ts`:

```
GraphQL.resolve("Mutation", "createProduct", async (root, args) => {
    const input = args.input;
    const product = new Product();
    product.name = input.name;
    product.category = input.category ?? "Uncategorized";
    product.price = parseFloat(input.price);
    product.inStock = Boolean(input.inStock ?? true);
    await product.save();
    return product.toDict();
});

GraphQL.resolve("Mutation", "updateProduct", async (root, args) => {
    const product = new Product();
    await product.load(args.id);

    if (!product.id) return null;

    const input = args.input;
    if (input.name !== undefined) product.name = input.name;
    if (input.category !== undefined) product.category = input.category;
    if (input.price !== undefined) product.price = parseFloat(input.price);
    if (input.inStock !== undefined) product.inStock = Boolean(input.inStock);
    await product.save();

    return product.toDict();
});

GraphQL.resolve("Mutation", "deleteProduct", async (root, args) => {
```

```
    const product = new Product();
    await product.load(args.id);
    if (!product.id) return { success: false, message: "Product not found" };
    await product.delete();
    return { success: true, message: "Product deleted" };
});

GraphQL.resolve("Query", "productsByCategory", async (root, args) => {
    const product = new Product();
    const products = await product.select("*", "category = :category", { category: args.category
    return products.map(p => p.toDict());
});
```

Testing Mutations

Create a product:

```
curl -X POST http://localhost:7148/graphql \
  -H "Content-Type: application/json" \
  -d '{
    "query": "mutation { createProduct(input: { name: \"Desk Lamp\", category: \"Office\", price
  }'
```

```
{
  "data": {
    "createProduct": {
      "id": 4,
      "name": "Desk Lamp",
      "price": 39.99
    }
  }
}
```

Update a product:

```
curl -X POST http://localhost:7148/graphql \
  -H "Content-Type: application/json" \
  -d '{
    "query": "mutation { updateProduct(id: 4, input: { price: 44.99, inStock: false }) { id name
  }'
```

```
{
  "data": {
    "updateProduct": {
      "id": 4,
      "name": "Desk Lamp",
      "price": 44.99,
      "inStock": false
    }
  }
}
```

Delete a product:

```
curl -X POST http://localhost:7148/graphql \
  -H "Content-Type: application/json" \
  -d '{
    "query": "mutation { deleteProduct(id: 4) { success message } }"
  }'
```

```
{
```

```
  "data": {
    "deleteProduct": {
      "success": true,
      "message": "Product deleted"
    }
  }
}
```

---

7. Nested Types and Relationships

The real power of GraphQL: traversing relationships in a single query. Authors, posts, comments -- all in one request.

Update `src/graphql/schema.graphql`:

```
type User {
    id: Int!
    name: String!
    email: String!
    posts: [Post!]!
}

type Post {
    id: Int!
    title: String!
    body: String!
    published: Boolean!
    author: User!
    comments: [Comment!]!
    commentCount: Int!
}

type Comment {
    id: Int!
    authorName: String!
    body: String!
    post: Post!
}

type Query {
    posts: [Post!]!
    post(id: Int!): Post
    user(id: Int!): User
}

type Mutation {
    createPost(userId: Int!, title: String!, body: String!, published: Boolean): Post!
    addComment(postId: Int!, authorName: String!, body: String!): Comment!
}
```

Add resolvers for the nested types:

```
import { GraphQL } from "tina4-nodejs";
import { Post } from "../orm/Post";
import { User } from "../orm/User";
import { Comment } from "../orm/Comment";
```

```
GraphQL.resolve("Query", "posts", async (root, args) => {
    const post = new Post();
    const posts = await post.select("*", "published = :pub", { pub: 1 });
    return posts.map(p => p.toDict());
});

GraphQL.resolve("Query", "post", async (root, args) => {
    const post = new Post();
    await post.load(args.id);
    return post.id ? post.toDict() : null;
});

// Resolve Post.author (nested field)
GraphQL.resolve("Post", "author", async (post, args) => {
    const user = new User();
    await user.load(post.user_id);
    return user.toDict();
});

// Resolve Post.comments (nested field)
GraphQL.resolve("Post", "comments", async (post, args) => {
    const comment = new Comment();
    const comments = await comment.select("*", "post_id = :postId", { postId: post.id });
    return comments.map(c => c.toDict());
});

// Resolve Post.commentCount (computed field)
GraphQL.resolve("Post", "commentCount", async (post, args) => {
    const comment = new Comment();
    const result = await comment.select("count(*) as cnt", "post_id = :postId", { postId: post.i
    return parseInt(result[0]?.cnt ?? 0, 10);
});

// Resolve User.posts (nested field)
GraphQL.resolve("User", "posts", async (user, args) => {
    const post = new Post();
    const posts = await post.select("*", "user_id = :userId", { userId: user.id });
    return posts.map(p => p.toDict());
});
```

Now clients can query across relationships in a single request:

```
curl -X POST http://localhost:7148/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.

---

8. Auto-Generating Schema from ORM Models

If your ORM models have `static autoCrud = true`, Tina4 can auto-generate GraphQL types and basic CRUD resolvers for them. Enable this in `.env`:

```
TINA4_GRAPHQL_AUTO_SCHEMA=true
```

With this setting, every ORM model with `static autoCrud = true` gets:

- A GraphQL type with fields matching the model properties
- A query to list all records (e.g., `products`)
- A query to get one by ID (e.g., `product(id: Int!)`)
- A mutation to create (e.g., `createProduct(input: ProductInput!)`)
- A mutation to update (e.g., `updateProduct(id: Int!, input: ProductUpdateInput!)`)
- A mutation to delete (e.g., `deleteProduct(id: Int!)`)

You can still define custom resolvers that override the auto-generated ones. Custom resolvers take precedence.

---

9. The GraphiQL Playground

When `TINA4_DEBUG=true`, Tina4 serves a GraphiQL interactive playground at:

```
http://localhost:7148/graphql/playground
```

GraphiQL gives you:

- A query editor with syntax highlighting and auto-completion
- A documentation explorer showing all types, queries, and mutations
- A results panel showing the response
- Query history

This is the fastest way to explore and test your GraphQL API during development. Type a query on the left, click the play button, and see the results on the right. The documentation explorer on the right sidebar shows every type, field, and argument available in your schema.

GraphiQL is only available when `TINA4_DEBUG=true`. In production, it is disabled.

---

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:7148/graphql \
  -H "Content-Type: application/json" \
  -d '{
    "query": "query GetProduct($id: Int!) { product(id: $id) { id name price } }",
    "variables": {"id": 1}
  }'
```

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

The query uses `$id` as a placeholder, and the actual value comes from the `variables` object. This is safer and more efficient than string interpolation.

---

11. Exercise: Build a GraphQL API for a Blog

Build a complete GraphQL API for a blog with posts, authors, and comments.

Requirements

- **Schema** -- Define types for `User`, `Post`, and `Comment` with the following fields:
- `User`: id, name, email, posts (nested)
- `Post`: id, title, body, published, author (nested), comments (nested), commentCount (computed)
- `Comment`: id, authorName, body, createdAt
- **Queries**:
- `posts` -- List all published posts
- `post(id: Int!)` -- Get a single post by ID
- `user(id: Int!)` -- Get a user with their posts
- **Mutations**:
- `createPost(userId: Int!, title: String!, body: String!, published: Boolean)` -- Create a new post
- `addComment(postId: Int!, authorName: String!, body: String!)` -- Add a comment to a post

- Use the ORM models from Chapter 6 (User, Post, Comment).

Test with these queries:

```
# List published posts with author names
curl -X POST http://localhost:7148/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:7148/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:7148/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:7148/graphql \
  -H "Content-Type: application/json" \
  -d '{"query": "mutation { addComment(postId: 1, authorName: \"Carol\", body: \"Nice article!\"
```

---

12. Solution

Schema

Create `src/graphql/blog-schema.graphql`:

```
type User {
    id: Int!
    name: String!
    email: String!
    posts: [Post!]!
}

type Post {
    id: Int!
    title: String!
    body: String!
    published: Boolean!
    author: User!
    comments: [Comment!]!
    commentCount: Int!
    createdAt: String
}

type Comment {
    id: Int!
    authorName: String!
    body: String!
    createdAt: String
}

type Query {
    posts: [Post!]!
```

```
    post(id: Int!): Post
    user(id: Int!): User
}

type Mutation {
    createPost(userId: Int!, title: String!, body: String!, published: Boolean): Post!
    addComment(postId: Int!, authorName: String!, body: String!): Comment!
}
```

Resolvers

Create `src/graphql/blog-resolvers.ts`:

```
import { GraphQL } from "tina4-nodejs";
import { Post } from "../orm/Post";
import { User } from "../orm/User";
import { Comment } from "../orm/Comment";

GraphQL.resolve("Query", "posts", async () => {
    const post = new Post();
    const posts = await post.select("*", "published = :pub", { pub: 1 }, "created_at DESC");
    return posts.map(p => p.toDict());
});

GraphQL.resolve("Query", "post", async (root, args) => {
    const post = new Post();
    await post.load(args.id);
    return post.id ? post.toDict() : null;
});

GraphQL.resolve("Query", "user", async (root, args) => {
    const user = new User();
    await user.load(args.id);
    return user.id ? user.toDict() : null;
});

GraphQL.resolve("Post", "author", async (post) => {
    const user = new User();
    await user.load(post.user_id);
    return user.toDict();
});

GraphQL.resolve("Post", "comments", async (post) => {
    const comment = new Comment();
    const comments = await comment.select("*", "post_id = :postId", { postId: post.id }, "create
    return comments.map(c => c.toDict());
});

GraphQL.resolve("Post", "commentCount", async (post) => {
    const comment = new Comment();
    const result = await comment.select("count(*) as cnt", "post_id = :postId", { postId: post.i
    return parseInt(result[0]?.cnt ?? 0, 10);
});

GraphQL.resolve("User", "posts", async (user) => {
    const post = new Post();
    const posts = await post.select("*", "user_id = :userId", { userId: user.id }, "created_at D
    return posts.map(p => p.toDict());
});
```

```
GraphQL.resolve("Mutation", "createPost", async (root, args) => {
    const user = new User();
    await user.load(args.userId);

    if (!user.id) {
        throw new Error("User not found");
    }

    const post = new Post();
    post.userId = args.userId;
    post.title = args.title;
    post.body = args.body;
    post.published = Boolean(args.published ?? false);
    await post.save();

    return post.toDict();
});

GraphQL.resolve("Mutation", "addComment", async (root, args) => {
    const post = new Post();
    await post.load(args.postId);

    if (!post.id) {
        throw new Error("Post not found");
    }

    const comment = new Comment();
    comment.postId = args.postId;
    comment.authorName = args.authorName;
    comment.body = args.body;
    await comment.save();

    return comment.toDict();
});
```

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`:

```
import { GraphQL } from "tina4-nodejs";
import { User } from "../orm/User";

GraphQL.resolve("Query", "user", async (root, args) => {
    const user = new User();
    await user.load(args.id);
    return user.id ? user.toDict() : null;
});
```

With the schema declaring `user(id: ID!): User`, 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 undefined-check needed inside the function.

---

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()`:

```
const result = await gql.execute(query, variables, {
    user: { id: 1, role: "admin" }
});
```

When no user is logged in, pass an empty context or omit the `user` key:

```
const result = await gql.execute(query, variables, {});
```

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. Schema File Not Found

Problem: Queries return errors about unknown types or fields.

Cause: The `.graphql` file is not in `src/graphql/`, or has a syntax error that prevents parsing.

Fix: Make sure schema files are in `src/graphql/` and end with `.graphql`. Check for syntax errors -- a missing `!` or unmatched brace will cause the entire schema to fail silently.

2. Resolver Not Called

Problem: A query returns `null` even though data exists.

Cause: The resolver is not registered, or the type/field names do not match the schema. GraphQL is case-sensitive.

Fix: Verify that `GraphQL.resolve("Query", "products", ...)` matches `type Query { products: ... }` in your schema. Check capitalization.

3. Nested Resolver Returns Wrong Data

Problem: `post.author` returns the entire post instead of the user.

Cause: The nested resolver receives the parent object as its first argument (`post`), not a fresh model. If you return the parent instead of loading the related record, you get the wrong data.

Fix: In a nested resolver, load the related record from the database using the foreign key from the parent:

```
GraphQL.resolve("Post", "author", async (post, args) => {
    const user = new User();
    await user.load(post.user_id);  // Use the foreign key from the parent
    return user.toDict();
});
```

4. Mutation Input Not Parsed

Problem: `args.input` is `undefined` inside a mutation resolver.

Cause: The mutation signature in the schema uses an `input` type, but the client query passes arguments directly instead of wrapping them in an `input` object.

Fix: Match the query to the schema. If the schema says `createProduct(input: ProductInput!)`, the client must send `createProduct(input: { name: "Widget", price: 9.99 })`, not `createProduct(name: "Widget", price: 9.99)`.

5. N+1 Query Problem

Problem: Loading 50 posts with their authors generates 51 database queries (1 for posts + 50 for authors).

Cause: Each nested resolver runs on its own. When you resolve `author` for each post, that is one query per post.

Fix: Use data loader patterns or batch loading. For simple cases, pre-load all related records in the parent resolver and pass them down. 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 complex cases, consider using the REST API with eager loading instead of GraphQL for that particular endpoint.

6. GraphQL Playground Returns 404

Problem: `/graphql/playground` returns a 404 error.

Cause: `TINA4_DEBUG` is set to `false`. The playground is only available in debug mode.

Fix: Set `TINA4_DEBUG=true` in your `.env` file. The playground is disabled in production by design.

7. Type Mismatch Between Schema and Resolver

Problem: A field declared as `Int!` in the schema receives a string from the resolver, causing a type error.

Cause: JavaScript's loose typing means your resolver might return `"42"` (a string) for an `Int!` field. GraphQL rejects it.

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 with `parseInt()`, `parseFloat()`, `Boolean()`:

```
return {
    id: parseInt(product.id, 10),
    price: parseFloat(product.price),
    inStock: Boolean(product.inStock)
};
```

The `toDict()` method handles this for model properties with type declarations, but computed or derived fields need manual casting.

---

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 built-in SOAP 1.1 / WSDL 1.1 engine. Zero external dependencies -- XML parsing uses simple string matching.

Defining a SOAP Service

Create a class that extends `WSDLService`. Each method you want to expose gets the `@WSDLOp` decorator with `input` and `output` type maps.

```
import { WSDLService, WSDLOp } from "tina4-nodejs";

class Calculator extends WSDLService {
  serviceName = "Calculator";
  serviceUrl = "/api/calculator";

  @WSDLOp({
    description: "Add two numbers",
    input: { a: "int", b: "int" },
    output: { Result: "int" },
  })
  async Add(a: number, b: number): Promise<Record<string, unknown>> {
    return { Result: a + b };
  }

  @WSDLOp({
    description: "Multiply two numbers",
    input: { x: "float", y: "float" },
    output: { Product: "double" },
  })
  async Multiply(x: number, y: number): Promise<Record<string, unknown>> {
    return { Product: x * y };
  }
}
```

Key points:

- `serviceName` -- appears in the WSDL `<service>` element.
- `serviceUrl` -- the URL path for both WSDL and SOAP requests.
- `input` -- maps parameter names to type strings.
- `output` -- maps return field names to type strings.

• The method receives parameters in the order declared in `input` and returns a plain object matching `output`.

Registering the Service

Call `register()` with your router to wire up two routes:

```
const calc = new Calculator();
calc.register(router);
```

This creates:

Method	URL	Purpose
GET	`/api/calculator?wsdl`	Returns the auto-generated WSDL XML
POST	`/api/calculator`	Accepts SOAP XML requests

Auto-Generated WSDL

Fetch the WSDL document:

```
curl http://localhost:7148/api/calculator?wsdl
```

Tina4 generates a complete WSDL 1.1 document containing `<types>`, `<message>`, `<portType>`, `<binding>`, and `<service>` sections. The endpoint URL is inferred from the request's `Host` header.

Type Mappings

The `input` and `output` maps use short type names that are converted to XSD types:

Tina4 Type	XSD Type
`int`, `integer`	`xsd:int`
`float`	`xsd:float`
`double`, `number`, `numeric`	`xsd:double`
`string`	`xsd:string`
`bool`, `boolean`	`xsd:boolean`

Unknown types default to `xsd:string`.

Lifecycle Hooks

Override `onRequest` and `onResult` to add validation, logging, or transformation around every operation call.

```
import { WSDLService, WSDLOp } from "tina4-nodejs";

class AuditedService extends WSDLService {
  serviceName = "AuditedService";
  serviceUrl = "/api/audited";

  onRequest(request: any): void {
    console.log(`SOAP request from ${request.headers["x-forwarded-for"] ?? "unknown"}`);
  }

  onResult(result: Record<string, unknown>): Record<string, unknown> {
    result["Timestamp"] = new Date().toISOString();
    return result;
  }

  @WSDLOp({
    description: "Health check",
    input: {},
    output: { Status: "string", Timestamp: "string" },
  })
  async Ping(): Promise<Record<string, unknown>> {
    return { Status: "ok" };
  }
}
```

Testing with curl

Fetch the WSDL definition:

```
curl http://localhost:7148/api/calculator?wsdl
```

Send a SOAP request to the `Add` operation:

```
curl -X POST http://localhost:7148/api/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>
```

A More Complete Example

A user lookup service with multiple operations:

```
import { WSDLService, WSDLOp } from "tina4-nodejs";

class UserService extends WSDLService {
  serviceName = "UserService";
  serviceUrl = "/api/users/soap";

  @WSDLOp({
    description: "Look up a user by ID",
    input: { userId: "int" },
    output: { Name: "string", Email: "string", Active: "boolean" },
  })
  async GetUser(userId: number): Promise<Record<string, unknown>> {
    // Replace with real database lookup
    return { Name: "Alice", Email: "alice@example.com", Active: true };
  }

  @WSDLOp({
    description: "Search users by name",
    input: { query: "string" },
    output: { Count: "int", Names: "string" },
  })
  async SearchUsers(query: string): Promise<Record<string, unknown>> {
    return { Count: 1, Names: "Alice" };
  }
}
```

SOAP Gotchas

- **Content-Type must be** `text/xml` -- SOAP requests are XML, not JSON.
- **Operation name must match** -- the element name inside `<Body>` must match your method name (case-sensitive).
- **Parameter order matters** -- values are extracted in the order declared in `input`.
- **Namespace prefixes are handled** -- Tina4 strips namespace prefixes when matching element names, so `<ns1:Add>` works the same as `<Add>`.

When to Use SOAP vs REST vs GraphQL

Scenario	Use
New API for web/mobile clients	REST or GraphQL
Integrating with legacy enterprise systems (SAP, banks, government)	SOAP
Clients require a machine-readable service contract	SOAP (WSDL)
Simple internal microservices	REST
Clients with diverse data needs	GraphQL

SOAP is rarely the right choice for new APIs, but when you need to expose a service that legacy systems can consume, Tina4 makes it straightforward -- define a class, annotate your types, and the framework handles the XML.

Real-time with WebSocket

1. The Refresh Button Problem

Your project management app needs live updates. Someone moves a card from "In Progress" to "Done." Everyone else on the team should see it. No page refresh. No polling. No waiting.

Traditional HTTP is request-response. The client asks. The server answers. The server cannot push data on its own. WebSocket breaks that wall. A persistent, bi-directional connection between browser and server. Either side sends messages at any time. The connection stays open until one side closes it.

Tina4 treats WebSocket the same way it treats routing. Define a WebSocket handler the same way you define an HTTP route. It runs on Node's built-in HTTP server -- no `ws` or `socket.io` required.

---

2. What WebSocket Is

HTTP works like this:

```
Client: "Give me /api/products"
Server: "Here are the products" (connection closes)
Client: "Give me /api/products" (new connection)
Server: "Here are the products" (connection closes)
```

WebSocket works like this:

```
Client: "I want to upgrade to WebSocket on /ws/chat"
Server: "Upgrade accepted. Connection open."
Client: "Hello everyone!"
Server: "Alice says: Hello everyone!"  (pushed to all clients)
Server: "Bob joined the chat"  (pushed to all clients, at any time)
Client: "Goodbye!"
Server: "Alice says: Goodbye!"
Client: (closes connection)
```

Key differences:

- **Persistent**: The connection stays open. No repeated handshakes.
- **Bi-directional**: The server pushes data without the client asking.
- **Low overhead**: After the initial handshake, messages are tiny. No HTTP headers per message.
- **Real-time**: Messages arrive within milliseconds.

---

3. Router.websocket() -- WebSocket as a Route

In Tina4, you define WebSocket handlers using `Router.websocket()`:

```
import { Router } from "tina4-nodejs";
```

```
Router.websocket("/ws/echo", (connection, event, data) => {
    if (event === "message") {
        connection.send(`Echo: ${data}`);
    }
});
```

This is the simplest WebSocket handler: it receives a message and sends it back with "Echo: " prepended. The callback receives three arguments:

	Type	Description		
	`WebSocketConnection`	The client connection object. Use it to send messages.		
	`"open" \	"message" \	"close"`	Which lifecycle
	`string`	The message text (only meaningful when `event === "message"`)		

Starting the Server

WebSocket runs alongside your HTTP server:

```
tina4 serve

  Tina4 Node.js v3.10.3
  HTTP server running at http://0.0.0.0:7148
  WebSocket server running at ws://0.0.0.0:7148
  Press Ctrl+C to stop
```

Both HTTP and WebSocket share the same port. The server detects the protocol upgrade request and routes it to the correct handler.

---

4. Connection Events

Every WebSocket connection goes through three lifecycle events:

Open

Fires when a client connects:

```
import { Router } from "tina4-nodejs";

Router.websocket("/ws/notifications", (connection, event, data) => {
    if (event === "open") {
        console.log(`Client connected: ${connection.id}`);
        connection.send(JSON.stringify({
            type: "welcome",
            message: "Connected to notifications",
            connection_id: connection.id
        }));
    }
});
```

Message

Fires when a client sends data:

```
import { Router } from "tina4-nodejs";

Router.websocket("/ws/notifications", (connection, event, data) => {
    if (event === "open") {
        connection.send(JSON.stringify({
            type: "welcome",
            connection_id: connection.id
        }));
    }

    if (event === "message") {
        const message = JSON.parse(data);
        console.log(`Received from ${connection.id}: ${data}`);

        if (message.type === "ping") {
            connection.send(JSON.stringify({
                type: "pong",
                timestamp: new Date().toISOString()
            }));
        }
    }
});
```

Close

Fires when a client disconnects:

```
import { Router } from "tina4-nodejs";

Router.websocket("/ws/notifications", (connection, event, data) => {
    if (event === "open") {
        console.log(`Client connected: ${connection.id}`);
    }

    if (event === "message") {
        console.log(`Message from ${connection.id}: ${data}`);
    }

    if (event === "close") {
        console.log(`Client disconnected: ${connection.id}`);
        // Clean up: remove from tracking, notify others, etc.
    }
});
```

A Complete Handler

Here is a handler that responds to all three events:

```
import { Router } from "tina4-nodejs";

Router.websocket("/ws/chat", (connection, event, data) => {
    switch (event) {
        case "open":
            console.log(`[Chat] New connection: ${connection.id}`);
            connection.send(JSON.stringify({
                type: "system",
```

```
                message: "Welcome to the chat!",
                your_id: connection.id
            }));
            break;

        case "message":
            const message = JSON.parse(data);
            console.log(`[Chat] ${connection.id}: ${message.text ?? data}`);
            connection.send(JSON.stringify({
                type: "message",
                from: connection.id,
                text: message.text ?? data,
                timestamp: new Date().toISOString()
            }));
            break;

        case "close":
            console.log(`[Chat] Disconnected: ${connection.id}`);
            break;
    }
});
```

---

5. Sending to a Single Client

`connection.send()` sends a message to the specific client that triggered the event:

```
import { Router } from "tina4-nodejs";

Router.websocket("/ws/private", (connection, event, data) => {
    if (event === "message") {
        const message = JSON.parse(data);
        const action = message.action ?? "";

        if (action === "get-time") {
            connection.send(JSON.stringify({
                type: "time",
                server_time: new Date().toISOString()
            }));
        }

        if (action === "get-status") {
            const memUsage = process.memoryUsage();
            connection.send(JSON.stringify({
                type: "status",
                uptime: Math.floor(process.uptime()),
                connections: 42,
                memory_mb: Math.round(memUsage.rss / 1024 / 1024 * 100) / 100
            }));
        }
    }
});
```

Only the client that sent the message receives the response. Other connected clients do not see it.

---

6. Broadcasting to All Clients

`connection.broadcast()` sends a message to every client connected to the same WebSocket path:

```
Router.websocket("/ws/announcements", (connection, event, data) => {
    if (event === "open") {
        connection.broadcast(JSON.stringify({
            type: "system",
            message: "A new user joined",
            online_count: connection.connectionCount()
        }));
    }

    if (event === "message") {
        const message = JSON.parse(data);
        connection.broadcast(JSON.stringify({
            type: "announcement",
            from: connection.id,
            text: message.text ?? "",
            timestamp: new Date().toISOString()
        }));
    }

    if (event === "close") {
        connection.broadcast(JSON.stringify({
            type: "system",
            message: "A user left",
            online_count: connection.connectionCount()
        }));
    }
});
```

When one client sends a message, every client connected to `/ws/announcements` receives it. This is the foundation for chat rooms, live dashboards, and collaborative editing.

Broadcast Excluding Sender

Sometimes you want to send to everyone except the client that triggered the event:

```
if (event === "message") {
    const message = JSON.parse(data);

    // Send to sender (confirmation)
    connection.send(JSON.stringify({
        type: "sent",
        text: message.text
    }));

    // Send to everyone else
    connection.broadcast(JSON.stringify({
        type: "message",
        from: message.username ?? "Anonymous",
        text: message.text,
        timestamp: new Date().toISOString()
    }), { excludeSelf: true });
}
```

The `{ excludeSelf: true }` option on `broadcast()` excludes the sender. The sender gets a "sent" confirmation. Everyone else gets the "message".

Sending JSON

Use `connection.sendJson()` to send an object as a JSON string without calling `JSON.stringify()` yourself:

```
Router.websocket("/ws/status", (connection, event, data) => {
    if (event === "open") {
        connection.sendJson({
            type: "welcome",
            connection_id: connection.id
        });
    }

    if (event === "message") {
        connection.sendJson({
            type: "ack",
            received: data
        });
    }
});
```

`sendJson()` serialises the data to JSON for you. It is equivalent to `connection.send(JSON.stringify(data))` but saves you the call.

Closing a Connection

Use `connection.close()` to close the connection from the server side:

```
Router.websocket("/ws/secure", (connection, event, data) => {
    if (event === "open") {
        const token = connection.params.token;
        if (!token || !validToken(token)) {
            connection.sendJson({ error: "Unauthorized" });
            connection.close();
            return;
        }

        connection.sendJson({ type: "welcome" });
    }
});
```

Connection Methods Summary

Method	Description
`connection.send(message)`	Send a string message to this connection only
`connection.sendJson(data)`	Send an object as JSON to this connection only
`connection.broadcast(message)`	Send to all connections on the same path
`connection.broadcast(message, { excludeSelf: true })`	Send to all except this connection
`connection.close()`	Close this connection from the server side
`connection.id`	Unique identifier for this connection
`connection.params`	Path parameters extracted from the URL
`connection.connectionCount()`	Number of active connections on this path

---

7. Path-Scoped Isolation

Different WebSocket paths are isolated. Clients connected to `/ws/chat/room-1` do not see messages from `/ws/chat/room-2`:

```
Router.websocket("/ws/chat/{room}", (connection, event, data) => {
    const room = connection.params.room;

    if (event === "open") {
        console.log(`[Room ${room}] New connection: ${connection.id}`);
        connection.broadcast(JSON.stringify({
            type: "system",
            message: `Someone joined room ${room}`,
            room,
            online: connection.connectionCount()
        }));
    }

    if (event === "message") {
        const message = JSON.parse(data);
        connection.broadcast(JSON.stringify({
            type: "message",
            room,
            from: message.username ?? "Anonymous",
            text: message.text ?? "",
            timestamp: new Date().toISOString()
        }));
    }

    if (event === "close") {
        connection.broadcast(JSON.stringify({
            type: "system",
            message: `Someone left room ${room}`,
```

```
            room,
            online: connection.connectionCount()
        }));
    }
});
```

Connect to different rooms:

```
ws://localhost:7148/ws/chat/general    -- general chat
ws://localhost:7148/ws/chat/random     -- random chat
ws://localhost:7148/ws/chat/dev-team   -- dev team chat
```

Broadcasting in `/ws/chat/general` reaches only clients connected to `/ws/chat/general`. The `dev-team` and `random` rooms are separate.

Chat rooms. Project-specific notifications. Per-user channels. No extra configuration. The URL path is the isolation boundary.

Path parameters use `{name}` curly-brace syntax, not `:name` colon syntax. Both `connection.params.room` and `connection.params["room"]` work.

---

8. Building a Live Chat

Here is a complete chat application with usernames, typing indicators, and online counts:

WebSocket Handler

Create `src/routes/chat-ws.ts`:

```
import { Router } from "tina4-nodejs";

const chatUsers: Record<string, { id: string; username: string; room: string; joinedAt: string }

Router.websocket("/ws/livechat/{room}", (connection, event, data) => {
    const room = connection.params.room;

    if (event === "open") {
        chatUsers[connection.id] = {
            id: connection.id,
            username: "Anonymous",
            room,
            joinedAt: new Date().toISOString()
        };

        connection.send(JSON.stringify({
            type: "welcome",
            message: `Connected to room: ${room}`,
            your_id: connection.id,
            online: connection.connectionCount()
        }));
    }

    if (event === "message") {
        const message = JSON.parse(data);
        const type = message.type ?? "message";

        if (type === "set-username") {
```

```
            const oldName = chatUsers[connection.id].username;
            chatUsers[connection.id].username = message.username;
            connection.broadcast(JSON.stringify({
                type: "system",
                message: `${oldName} is now known as ${message.username}`
            }));
        }

        if (type === "message") {
            const username = chatUsers[connection.id].username;
            connection.broadcast(JSON.stringify({
                type: "message",
                from: username,
                from_id: connection.id,
                text: message.text ?? "",
                timestamp: new Date().toISOString()
            }));
        }

        if (type === "typing") {
            const username = chatUsers[connection.id].username;
            connection.broadcast(JSON.stringify({
                type: "typing",
                username
            }), { excludeSelf: true });
        }
    }

    if (event === "close") {
        const username = chatUsers[connection.id]?.username ?? "Unknown";
        delete chatUsers[connection.id];

        connection.broadcast(JSON.stringify({
            type: "system",
            message: `${username} left the chat`,
            online: connection.connectionCount()
        }));
    }
});
```

---

9. Live Notifications

WebSocket is built for pushing notifications to users in real time:

```
import { Router } from "tina4-nodejs";

Router.websocket("/ws/notifications/{userId}", (connection, event, data) => {
    const userId = connection.params.userId;

    if (event === "open") {
        console.log(`[Notifications] User ${userId} connected`);
        connection.send(JSON.stringify({
            type: "connected",
            message: "Listening for notifications"
        }));
    }
```

```
    if (event === "message") {
        const message = JSON.parse(data);
        if (message.type === "mark-read") {
            console.log(`[Notifications] User ${userId} read notification ${message.id}`);
        }
    }
});

Router.post("/api/orders/{orderId}/ship", async (req, res) => {
    const orderId = req.params.orderId;
    const userId = req.body.user_id ?? 0;

    // Update order status in database...

    // Send real-time notification via WebSocket
    Router.pushToWebSocket(`/ws/notifications/${userId}`, JSON.stringify({
        type: "notification",
        title: "Order Shipped",
        message: `Your order #${orderId} has been shipped!`,
        action_url: `/orders/${orderId}`,
        timestamp: new Date().toISOString()
    }));

    return res.status(201).json({ message: "Order shipped, user notified" });
});
```

`Router.pushToWebSocket()` lets your HTTP handlers send messages to WebSocket clients. This bridges the gap between traditional request-response endpoints and real-time notifications.

Use Cases

- **Dashboard updates** -- New orders, user signups, system alerts
- **Notification feeds** -- Task assignments, comments, mentions
- **Live data** -- Stock tickers, sensor readings, log streams
- **Progress tracking** -- File upload progress, background job status

Scoped Notifications

Path parameters scope the broadcast. Only clients connected to a specific path receive the message:

```
// Only clients connected to /ws/project/42 receive this
Router.pushToWebSocket("/ws/project/42", JSON.stringify({
    type: "task_completed",
    task: "Design review"
}));
```

---

10. Connecting from JavaScript

Tina4 provides `frond.js`, a built-in JavaScript helper library that includes WebSocket support:

Basic Connection

```
<script src="/js/frond.js"></script>
<script>
    const ws = frond.ws("/ws/chat/general");

    ws.on("open", function () {
        console.log("Connected to chat");
    });

    ws.on("message", function (data) {
        const message = JSON.parse(data);
        console.log("Received:", message);

        if (message.type === "message") {
            addMessageToUI(message.from, message.text, message.timestamp);
        }

        if (message.type === "system") {
            addSystemMessage(message.message);
        }
    });

    ws.on("close", function () {
        console.log("Disconnected from chat");
    });

    function sendMessage(text) {
        ws.send(JSON.stringify({
            type: "message",
            text: text
        }));
    }

    function setUsername(name) {
        ws.send(JSON.stringify({
            type: "set-username",
            username: name
        }));
    }
</script>
```

Auto-Reconnect

`frond.js` reconnects when the connection drops:

```
const ws = frond.ws("/ws/notifications/42", {
    reconnect: true,           // Enable auto-reconnect (default: true)
    reconnectInterval: 3000,   // Retry every 3 seconds
    maxReconnectAttempts: 10   // Give up after 10 attempts
});

ws.on("reconnect", function (attempt) {
    console.log("Reconnecting... attempt " + attempt);
});

ws.on("reconnect_failed", function () {
    console.log("Failed to reconnect after maximum attempts");
});
```

Using Native WebSocket

If you prefer not to use `frond.js`, the native WebSocket API works too:

```
const ws = new WebSocket("ws://localhost:7148/ws/chat/general");

ws.onopen = function () {
    console.log("Connected");
    ws.send(JSON.stringify({ type: "set-username", username: "Alice" }));
};

ws.onmessage = function (event) {
    const message = JSON.parse(event.data);
    console.log("Received:", message);
};

ws.onclose = function () {
    console.log("Disconnected");
};

ws.onerror = function (error) {
    console.error("WebSocket error:", error);
};
```

`frond.js` gives you auto-reconnect, message buffering during reconnection, and a cleaner event API. Native WebSocket does not.

Manual Auto-Reconnect with Native WebSocket

If you use the native API and want reconnection, build it yourself:

```
function connectWebSocket(path) {
    const ws = new WebSocket(`ws://${location.host}${path}`);

    ws.addEventListener("close", () => {
        console.log("Disconnected. Reconnecting in 3s...");
        setTimeout(() => connectWebSocket(path), 3000);
    });

    ws.addEventListener("message", (event) => {
        const data = JSON.parse(event.data);
        console.log("Received:", data);
    });

    return ws;
}

const ws = connectWebSocket("/ws/chat/general");
```

This reconnects on every close. It does not limit retries or buffer messages sent during the reconnection window. For production use, `frond.js` handles these edge cases for you.

---

11. A Complete Chat Page

Here is a full chat page using templates and WebSocket:

Create `src/templates/chat.html`:

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

{% block title %}Chat - {{ room }}{% endblock %}

{% block content %}
    <h1>Chat Room: {{ room }}</h1>
    <p id="status">Connecting...</p>
    <p id="online">Online: 0</p>

    <div id="messages" style="border: 1px solid #ddd; height: 400px; overflow-y: scroll; padding
    </div>

    <div id="typing" style="height: 20px; color: #888; font-style: italic; margin-bottom: 4px;">

    <form id="chat-form" style="display: flex; gap: 8px;">
        <input type="text" id="message-input" placeholder="Type a message..."
               style="flex: 1; padding: 8px; border: 1px solid #ddd; border-radius: 4px;">
        <button type="submit" style="padding: 8px 16px; background: #333; color: white; border:
            Send
        </button>
    </form>

    <script src="/js/frond.js"></script>
    <script>
        const room = "{{ room }}";
        const ws = frond.ws("/ws/livechat/" + room);
        const messagesDiv = document.getElementById("messages");
        const statusEl = document.getElementById("status");
        const onlineEl = document.getElementById("online");
        const typingEl = document.getElementById("typing");

        let username = prompt("Enter your username:") || "Anonymous";

        ws.on("open", function () {
            statusEl.textContent = "Connected";
            ws.send(JSON.stringify({ type: "set-username", username: username }));
        });

        ws.on("message", function (data) {
            const msg = JSON.parse(data);

            if (msg.type === "message") {
                const div = document.createElement("div");
                div.style.marginBottom = "8px";
                div.innerHTML = "<strong>" + msg.from + ":</strong> " + msg.text;
                messagesDiv.appendChild(div);
                messagesDiv.scrollTop = messagesDiv.scrollHeight;
            }

            if (msg.type === "system") {
                const div = document.createElement("div");
                div.style.cssText = "color: #888; font-style: italic; margin-bottom: 8px;";
                div.textContent = msg.message;
                messagesDiv.appendChild(div);
                messagesDiv.scrollTop = messagesDiv.scrollHeight;
            }
```

```
            if (msg.type === "typing") {
                typingEl.textContent = msg.username + " is typing...";
                setTimeout(function () { typingEl.textContent = ""; }, 2000);
            }

            if (msg.online !== undefined) {
                onlineEl.textContent = "Online: " + msg.online;
            }
        });

        ws.on("close", function () {
            statusEl.textContent = "Disconnected. Reconnecting...";
        });

        document.getElementById("chat-form").addEventListener("submit", function (e) {
            e.preventDefault();
            const input = document.getElementById("message-input");
            if (input.value.trim()) {
                ws.send(JSON.stringify({ type: "message", text: input.value }));
                input.value = "";
            }
        });

        document.getElementById("message-input").addEventListener("input", function () {
            ws.send(JSON.stringify({ type: "typing" }));
        });
    </script>
{% endblock %}
```

Create the route to serve the page:

```
Router.get("/chat/{room}", async (req, res) => {
    const room = req.params.room;
    return res.html("chat.html", { room });
});
```

Visit `http://localhost:7148/chat/general` in two browser tabs. Type in one tab and watch the message appear in the other.

---

12. Exercise: Build a Real-Time Chat Room

Build a WebSocket chat room with the following features:

Requirements

- WebSocket endpoint at `/ws/room/{roomName}` that handles:
- `open`: Send welcome message with connection count
- `message` with type `set-name`: Set the user's display name
- `message` with type `chat`: Broadcast the message to all clients with the sender's name
- `close`: Broadcast that the user left
- HTTP endpoint at `GET /room/{roomName}` that serves an HTML chat page
- The chat page should:

- Prompt for a username on load
- Display messages in real time
- Show system messages (join/leave) in a different style
- Show the count of online users

Test by:

- Open `http://localhost:7148/room/test` in two browser tabs
- Set different usernames in each
- Send messages from both tabs and verify they appear in both
- Close one tab and verify the "user left" message appears in the other

---

13. Solution

Create `src/routes/chat-room.ts`:

```
import { Router } from "tina4-nodejs";

const roomUsers: Record<string, string> = {};

Router.websocket("/ws/room/{roomName}", (connection, event, data) => {
    const room = connection.params.roomName;
    const key = `${room}:${connection.id}`;

    if (event === "open") {
        roomUsers[key] = "Anonymous";

        connection.send(JSON.stringify({
            type: "system",
            message: `Welcome to room: ${room}. Set your name with: `
                + '{"type": "set-name", "name": "YourName"}',
            online: connection.connectionCount()
        }));

        connection.broadcast(JSON.stringify({
            type: "system",
            message: "A new user joined",
            online: connection.connectionCount()
        }), { excludeSelf: true });
    }

    if (event === "message") {
        const msg = JSON.parse(data);
        const msgType = msg.type ?? "chat";

        if (msgType === "set-name") {
            const oldName = roomUsers[key] ?? "Anonymous";
            const newName = msg.name ?? "Anonymous";
            roomUsers[key] = newName;

            connection.broadcast(JSON.stringify({
                type: "system",
```

```
                message: `${oldName} changed their name to ${newName}`
            }));
        }

        if (msgType === "chat") {
            const username = roomUsers[key] ?? "Anonymous";

            connection.broadcast(JSON.stringify({
                type: "chat",
                from: username,
                text: msg.text ?? "",
                timestamp: new Date().toTimeString().substring(0, 8)
            }));
        }
    }

    if (event === "close") {
        const username = roomUsers[key] ?? "Anonymous";
        delete roomUsers[key];

        connection.broadcast(JSON.stringify({
            type: "system",
            message: `${username} left the room`,
            online: connection.connectionCount()
        }));
    }
});

Router.get("/room/{roomName}", async (req, res) => {
    const room = req.params.roomName;
    return res.html("room.html", { room });
});
```

Create `src/templates/room.html`:

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

{% block title %}Room: {{ room }}{% endblock %}

{% block content %}
    <h1>Room: {{ room }}</h1>
    <p id="online" style="color: #666;">Online: 0</p>

    <div id="messages" style="border: 1px solid #ddd; height: 400px; overflow-y: auto; padding:
    </div>

    <form id="form" style="display: flex; gap: 8px;">
        <input type="text" id="input" placeholder="Type a message..."
               style="flex: 1; padding: 10px; border: 1px solid #ddd; border-radius: 4px; font-s
        <button type="submit"
                style="padding: 10px 20px; background: #1a1a2e; color: white; border: none; bord
            Send
        </button>
    </form>

    <script src="/js/frond.js"></script>
    <script>
        const room = "{{ room }}";
        const username = prompt("Choose a username:") || "Anonymous";
```

```
        const ws = frond.ws("/ws/room/" + room);
        const msgs = document.getElementById("messages");

        ws.on("open", function () {
            ws.send(JSON.stringify({ type: "set-name", name: username }));
        });

        ws.on("message", function (raw) {
            const msg = JSON.parse(raw);
            const div = document.createElement("div");
            div.style.marginBottom = "8px";

            if (msg.type === "chat") {
                div.innerHTML = "<strong>" + msg.from + "</strong> <span style='color:#999;font-
            } else if (msg.type === "system") {
                div.style.color = "#888";
                div.style.fontStyle = "italic";
                div.textContent = msg.message;
            }

            msgs.appendChild(div);
            msgs.scrollTop = msgs.scrollHeight;

            if (msg.online !== undefined) {
                document.getElementById("online").textContent = "Online: " + msg.online;
            }
        });

        document.getElementById("form").addEventListener("submit", function (e) {
            e.preventDefault();
            const input = document.getElementById("input");
            if (input.value.trim()) {
                ws.send(JSON.stringify({ type: "chat", text: input.value }));
                input.value = "";
            }
        });
    </script>
{% endblock %}
```

Open `http://localhost:7148/room/test` in two browser tabs. Set different usernames. Send messages from one tab and watch them appear in both. Close one tab and verify the "left the room" message appears.

---

14. Scaling with a Backplane

When you run a single server instance, `broadcast()` reaches every connected client. But in production you often run multiple instances behind a load balancer. Each instance only knows about its own connections. A message broadcast on instance A never reaches clients connected to instance B.

A backplane solves this. It relays WebSocket messages across all instances using a shared pub/sub channel. Tina4 supports Redis as a backplane out of the box.

Configuration

Set two environment variables in your `.env`:

```
TINA4_WS_BACKPLANE=redis
TINA4_WS_BACKPLANE_URL=redis://localhost:6379
```

When `TINA4_WS_BACKPLANE` is set, every `broadcast()` call publishes the message to Redis. Every instance subscribes to the same channel and forwards the message to its local connections. No code changes required -- your existing WebSocket routes work as before.

Requirements

The Redis backplane requires a Redis client package as an optional dependency:

```
npm install ioredis
```

If `TINA4_WS_BACKPLANE` is not set (the default), Tina4 broadcasts only to local connections. This is fine for single-instance deployments.

---

15. Gotchas

1. WebSocket Needs a Persistent Server

Problem: WebSocket connections drop right away or do not work at all.

Cause: You are running behind a serverless platform or a proxy that closes idle connections. Traditional request-response servers do not support persistent connections. Each request is handled and the connection is closed.

Fix: Use `tina4 serve` which runs Tina4's built-in HTTP server that handles both HTTP and WebSocket. For production, you can use Nginx as a reverse proxy, but it must be configured for WebSocket proxying with `proxy_set_header Upgrade $http_upgrade` and `proxy_set_header Connection "upgrade"`.

2. Messages Are Strings, Not Objects

Problem: `data` in the message handler is a string, not a JavaScript object.

Cause: WebSocket transmits raw strings. If the client sends JSON, you receive the JSON string, not a parsed object.

Fix: Always `JSON.parse(data)` when you expect JSON messages. Always `JSON.stringify()` when you send structured data. WebSocket does not know about content types -- it is just bytes on a wire.

3. Connection Count Is Per-Path

Problem: `connection.connectionCount()` returns a lower number than expected.

Cause: Connection count is scoped to the WebSocket path. Clients on `/ws/chat/room-1` are counted separately from `/ws/chat/room-2`.

Fix: This is by design. Each path is an isolated group. If you need a global connection count across all paths, maintain a counter in a shared variable or use a module-level map.

4. Broadcasting Does Not Scale Across Servers

Problem: Users connected to different server instances do not see each other's messages.

Cause: `connection.broadcast()` sends only to clients connected to the same server process. With multiple server instances behind a load balancer, each instance has its own set of connections.

Fix: Use a pub/sub backend like Redis to relay messages across server instances. Each server subscribes to a Redis channel, and broadcast messages are published to Redis so all servers receive them. See section 14 for configuration.

5. Large Messages Cause Disconnects

Problem: The connection drops when sending a large message.

Cause: WebSocket servers have a maximum message size. The default is usually around 1MB. If your message exceeds this, the server closes the connection.

Fix: Keep messages small. Under 64KB is a good target. For large data transfers, use HTTP endpoints instead of WebSocket. WebSocket is designed for small, frequent messages -- not bulk data transfer.

6. Memory Leak from Tracking Connected Users

Problem: The server's memory usage grows over time and the process crashes.

Cause: You store user data in an object (like `chatUsers`) but do not clean it up when users disconnect. If the `close` event handler does not remove the user from the object, the object grows without limit.

Fix: Always clean up in the `close` handler. Remove disconnected users from tracking objects: `delete chatUsers[connection.id]`. Test by connecting and disconnecting many times to verify memory stays stable.

7. No Authentication on WebSocket Connect

Problem: Anyone can connect to your WebSocket endpoint and see all messages.

Cause: The WebSocket upgrade request does not carry your JWT token in the `Authorization` header. Browsers do not support custom headers on WebSocket connections.

Fix: Pass the token as a query parameter: `ws://localhost:7148/ws/chat?token=eyJ...`. In your `open` handler, validate the token and disconnect if invalid. Use a short-lived token for WebSocket connections. You can also authenticate via an HTTP endpoint first, store the session, and check the session cookie during the WebSocket upgrade.

8. Route Params Use `{id}` Not `:id`

Problem: Your WebSocket path parameters do not resolve. `connection.params` is empty.

Cause: You used Express-style `:param` colon syntax instead of Tina4's `{param}` curly-brace syntax.

Fix: WebSocket path parameters follow the same `{param}` syntax as HTTP routes. Write `/ws/chat/{room}`, not `/ws/chat/:room`. Both `connection.params.room` and `connection.params["room"]` work.

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:7148/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:7148;
    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:7148/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:7148/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. Legacy Services Are Still Services

SOAP is not new. It is also not gone. Banks, government agencies, ERP systems, and healthcare providers expose SOAP APIs. If you integrate with any of them, you need to speak SOAP.

WSDL (Web Services Description Language) is the schema for SOAP services. It defines operations, input types, and output types in XML. You call an operation, you get XML back, you parse it.

Tina4 provides a `WSDL` class that both consumes external SOAP services and publishes your own. You write normal TypeScript functions. Tina4 generates the WSDL document and handles the XML envelope plumbing.

---

2. Consuming an External SOAP Service

Loading the WSDL

```
import { WSDL } from "tina4-nodejs";

const client = await WSDL.load("https://www.dataaccess.com/webservicesserver/NumberConversion.ws
```

`WSDL.load()` fetches the WSDL document, parses it, and returns a client object. Operations defined in the WSDL become methods on the client.

Calling an Operation

```
const result = await client.call("NumberToWords", {
    ubiNum: 42
});

console.log(result);
// "forty two "
```

`call(operationName, params)` builds the SOAP envelope, sends the request, parses the XML response, and returns the unwrapped value.

Handling Errors

```
try {
    const result = await client.call("NumberToWords", { ubiNum: -1 });
    console.log(result);
} catch (err) {
    if (err instanceof SOAPFault) {
        console.error("SOAP fault:", err.faultString, err.detail);
    } else {
        console.error("Network error:", err.message);
    }
}
```

---

3. wsdl_operation Decorator

When exposing your own functions as SOAP operations, use the `wsdl_operation` decorator to describe the input and output types:

```
import { wsdl_operation } from "tina4-nodejs";

@wsdl_operation({
    name: "GetProductPrice",
    input: {
        productId: "string"
    },
    output: {
        price: "decimal",
        currency: "string",
        available: "boolean"
    },
    description: "Returns the current price for a product by ID"
})
async function GetProductPrice(params: { productId: string }) {
    const price = await lookupPrice(params.productId);
    return {
        price: price.amount,
        currency: price.currency,
        available: price.inStock
    };
}
```

The decorator registers the function as a SOAP operation and provides type information for WSDL generation.

---

4. Publishing a SOAP Service

Register your operations and mount the SOAP endpoint:

```
import { Router, WSDL, wsdl_operation } from "tina4-nodejs";

// Define operations
@wsdl_operation({
    name: "ConvertCurrency",
    input: { amount: "decimal", from: "string", to: "string" },
    output: { result: "decimal", rate: "decimal" }
})
async function ConvertCurrency(params: { amount: number; from: string; to: string }) {
    // Simulate a conversion
    const rate = params.from === "USD" && params.to === "EUR" ? 0.92 : 1.0;
    return {
        result: parseFloat((params.amount * rate).toFixed(2)),
        rate
    };
}

@wsdl_operation({
    name: "GetExchangeRate",
    input: { from: "string", to: "string" },
    output: { rate: "decimal", timestamp: "string" }
```

```
})
async function GetExchangeRate(params: { from: string; to: string }) {
    return {
        rate: params.from === "USD" && params.to === "EUR" ? 0.92 : 1.0,
        timestamp: new Date().toISOString()
    };
}

// Mount the SOAP endpoint
const service = new WSDL({
    name: "CurrencyService",
    namespace: "urn:currency-service",
    operations: [ConvertCurrency, GetExchangeRate]
});

Router.all("/soap/currency", service.handler());
Router.get("/soap/currency?wsdl", service.wsdlDocument());
```

Auto WSDL

The WSDL document is generated automatically from your operation decorators:

```
curl "http://localhost:7148/soap/currency?wsdl"
```

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="CurrencyService"
  targetNamespace="urn:currency-service"
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:tns="urn:currency-service"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">

  <message name="ConvertCurrencyRequest">
    <part name="amount" type="xsd:decimal"/>
    <part name="from" type="xsd:string"/>
    <part name="to" type="xsd:string"/>
  </message>

  <message name="ConvertCurrencyResponse">
    <part name="result" type="xsd:decimal"/>
    <part name="rate" type="xsd:decimal"/>
  </message>

  <portType name="CurrencyServicePortType">
    <operation name="ConvertCurrency">
      <input message="tns:ConvertCurrencyRequest"/>
      <output message="tns:ConvertCurrencyResponse"/>
    </operation>
  </portType>

  ...
</definitions>
```

---

5. Calling Your Published Service

Test with a SOAP client or curl:

```
curl -X POST http://localhost:7148/soap/currency \
  -H "Content-Type: text/xml" \
  -H "SOAPAction: ConvertCurrency" \
  -d '<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
               xmlns:tns="urn:currency-service">
  <soap:Body>
    <tns:ConvertCurrency>
      <amount>100</amount>
      <from>USD</from>
      <to>EUR</to>
    </tns:ConvertCurrency>
  </soap:Body>
</soap:Envelope>'
```

Response:

```
<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <ConvertCurrencyResponse>
      <result>92.00</result>
      <rate>0.92</rate>
    </ConvertCurrencyResponse>
  </soap:Body>
</soap:Envelope>
```

---

6. Exercise: Expose a Product Catalogue as a SOAP Service

Build a SOAP service with two operations: `GetProduct` and `ListProducts`.

Requirements

- `GetProduct` — takes `productId: string`, returns `id`, `name`, `price`, `inStock`
- `ListProducts` — takes `category: string`, returns an array of products
- Mount at `/soap/products` with auto WSDL at `/soap/products?wsdl`

Test with:

```
# Fetch the WSDL
curl "http://localhost:7148/soap/products?wsdl"

# Call GetProduct
curl -X POST http://localhost:7148/soap/products \
  -H "Content-Type: text/xml" \
  -H "SOAPAction: GetProduct" \
  -d '<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
        <soap:Body><GetProduct><productId>1</productId></GetProduct></soap:Body>
      </soap:Envelope>'
```

---

7. Solution

```
import { Router, WSDL, wsdl_operation } from "tina4-nodejs";

const PRODUCTS = [
    { id: "1", name: "Wireless Keyboard", category: "Electronics", price: 79.99, inStock: true },
    { id: "2", name: "Yoga Mat", category: "Fitness", price: 29.99, inStock: true },
    { id: "3", name: "Coffee Grinder", category: "Kitchen", price: 49.99, inStock: false },
    { id: "4", name: "Standing Desk", category: "Electronics", price: 549.99, inStock: true },
];

@wsdl_operation({
    name: "GetProduct",
    input: { productId: "string" },
    output: { id: "string", name: "string", price: "decimal", inStock: "boolean" }
})
async function GetProduct(params: { productId: string }) {
    const product = PRODUCTS.find(p => p.id === params.productId);
    if (!product) {
        throw new Error(`Product ${params.productId} not found`);
    }
    return { id: product.id, name: product.name, price: product.price, inStock: product.inStock
}

@wsdl_operation({
    name: "ListProducts",
    input: { category: "string" },
    output: { products: "array" }
})
async function ListProducts(params: { category: string }) {
    const filtered = params.category
        ? PRODUCTS.filter(p => p.category.toLowerCase() === params.category.toLowerCase())
        : PRODUCTS;
    return { products: filtered };
}

const service = new WSDL({
    name: "ProductService",
    namespace: "urn:product-service",
    operations: [GetProduct, ListProducts]
});

Router.all("/soap/products", service.handler());
Router.get("/soap/products", service.wsdlDocument());  // ?wsdl returns the schema
```

---

8. Gotchas

1. SOAP is XML -- whitespace matters in some parsers

Extra whitespace inside element values can cause remote parsers to reject your response.

Fix: Tina4's WSDL class generates clean XML without extra whitespace. Do not manually construct SOAP envelopes.

2. Namespace mismatches break everything

SOAP parsers are strict about XML namespaces. A mismatch between the namespace in your request and the namespace declared in the WSDL causes a fault.

Fix: Always copy the namespace from the generated WSDL document into your client requests. Do not guess namespace URNs.

3. WSDL.load() is slow on first call

Fetching and parsing a remote WSDL on every request defeats the purpose of using a client.

Fix: Call `WSDL.load()` once at startup and store the result. The client instance is stateless between calls.

4. Decimal types lose precision in JavaScript

SOAP `decimal` values are arbitrary precision. JavaScript `number` is IEEE 754 floating point. `0.1 + 0.2 !== 0.3`.

Fix: For financial values, use string representation in the SOAP layer and a decimal library internally. Return values as strings formatted to the required precision.

DI Container

1. Stop Passing Dependencies Through Every Function Call

Your database connection, email client, cache client, payment gateway, and logger are needed in dozens of places. You could import them all directly. You could pass them as function parameters. Both approaches make testing hard and create tight coupling.

Dependency injection (DI) separates what you need from how you get it. A container holds instances. You register once. You resolve anywhere. Tests swap implementations without touching production code.

Tina4's `Container` class is a lightweight DI container. Register classes or factories. Resolve by name. Singleton or transient. No reflection magic, no decorators required unless you want them.

---

2. The Container Class

```
import { Container } from "tina4-nodejs";
```

`Container` is a class you instantiate. You can have multiple containers for different subsystems, or use a single application-wide container.

---

3. Registering a Service

`register()` stores a factory function or class constructor under a name:

```
import { Container } from "tina4-nodejs";

const container = new Container();

// Register a factory function
container.register("logger", () => ({
    info: (msg: string) => console.log(`[INFO] ${msg}`),
    error: (msg: string) => console.error(`[ERROR] ${msg}`)
}));

// Register a class constructor
class EmailService {
    send(to: string, subject: string, body: string) {
        console.log(`Sending "${subject}" to ${to}`);
    }
}

container.register("email", () => new EmailService());
```

Every call to `container.get("email")` invokes the factory and returns a new instance by default.

---

4. Singleton Registration

Use `singleton()` to register a service that is only instantiated once. Every call to `get()` returns the same instance:

```
import { Container } from "tina4-nodejs";
import { Database } from "tina4-nodejs/orm";

const container = new Container();

// Database connections are expensive -- only create one
container.singleton("db", () => {
    return Database.getConnection();
});

// Both calls return the exact same connection object
const db1 = container.get("db");
const db2 = container.get("db");
console.log(db1 === db2);  // true
```

Use `singleton` for: database connections, cache clients, API clients, configuration objects. Use `register` (transient) for: request-scoped services, services with per-call state.

---

5. Resolving Services with get()

```
const logger = container.get<Logger>("logger");
logger.info("Application started");

const email = container.get<EmailService>("email");
email.send("alice@example.com", "Welcome!", "Thanks for signing up.");
```

Provide a type parameter for TypeScript type safety. The container returns `unknown` without one; with a type parameter it returns the typed instance.

---

6. Checking Registration with has()

```
if (container.has("payment")) {
    const payment = container.get("payment");
    // ...
} else {
    console.warn("Payment service not registered");
}
```

`has()` returns `true` if a name is registered, regardless of whether the service has been instantiated yet.

---

7. Resetting the Container

Remove all registrations, useful in tests:

```
// Remove a specific registration
container.unregister("email");

// Clear everything
container.reset();
```

After `reset()`, any call to `get()` throws until services are re-registered.

---

8. Services That Depend on Other Services

Factories receive the container as an argument, enabling nested resolution:

```
import { Container } from "tina4-nodejs";

const container = new Container();

container.singleton("config", () => ({
    smtpHost: process.env.TINA4_MAIL_HOST ?? "localhost",
    smtpPort: parseInt(process.env.TINA4_MAIL_PORT ?? "587")
}));

container.singleton("mailer", (c) => {
    const config = c.get<{ smtpHost: string; smtpPort: number }>("config");
    return new SMTPMailer(config.smtpHost, config.smtpPort);
});

container.singleton("notifier", (c) => {
    const mailer = c.get<SMTPMailer>("mailer");
    return new NotificationService(mailer);
});

// All dependencies resolved automatically
const notifier = container.get<NotificationService>("notifier");
```

---

9. Application-Wide Container Pattern

Expose a single container instance from a module so it can be imported anywhere:

`src/container.ts`:

```
import { Container } from "tina4-nodejs";
import { Database } from "tina4-nodejs/orm";
import { Messenger } from "tina4-nodejs";
import { Log } from "tina4-nodejs";

export const app = new Container();

app.singleton("db", () => Database.getConnection());

app.singleton("mailer", () => new Messenger());

app.singleton("log", () => Log);

app.register("orderService", (c) => {
```

```
    return new OrderService(
        c.get("db"),
        c.get("mailer"),
        c.get("log")
    );
});
```

`src/routes/orders.ts`:

```
import { Router } from "tina4-nodejs";
import { app } from "../container";

Router.post("/api/orders", async (req, res) => {
    const orderService = app.get<OrderService>("orderService");
    const result = await orderService.place(req.body);
    return res.status(201).json(result);
});
```

---

10. Testing with the DI Container

Swap real services for test doubles without touching production code:

```
import { Container } from "tina4-nodejs";

// Production container
import { app } from "../src/container";

// In tests: override specific services
beforeEach(() => {
    app.unregister("mailer");
    app.singleton("mailer", () => ({
        send: async () => ({ success: true, messageId: "test-123" })
    }));
});

afterEach(() => {
    // Restore after each test
    app.unregister("mailer");
    app.singleton("mailer", () => new Messenger());
});
```

The test installs a no-op mailer. Route handlers use `app.get("mailer")` — they never know the difference.

---

11. Exercise: Build a Service Container for a Store API

Wire together a product store API using the DI container.

Requirements

- Register three singletons: `productRepo` (in-memory data), `priceService` (computes discounts), and `logger` (structured logging)
- Register a transient `productService` that depends on all three

- Create a `GET /api/store/{id}` route that uses `productService` to fetch a product with its discounted price

Test with:

```
curl http://localhost:7148/api/store/1
curl http://localhost:7148/api/store/99  # should return 404
```

---

12. Solution

`src/store/container.ts`:

```
import { Container, Log } from "tina4-nodejs";

export const store = new Container();

// Repository: source of truth
store.singleton("productRepo", () => ({
    findById: (id: number) => {
        const products = [
            { id: 1, name: "Wireless Keyboard", basePrice: 79.99, category: "Electronics" },
            { id: 2, name: "Yoga Mat", basePrice: 29.99, category: "Fitness" },
            { id: 3, name: "Standing Desk", basePrice: 549.99, category: "Electronics" }
        ];
        return products.find(p => p.id === id) ?? null;
    },
    findAll: () => []
}));

// Price service: applies category discounts
store.singleton("priceService", () => ({
    calculate: (basePrice: number, category: string) => {
        const discounts: Record<string, number> = { Electronics: 0.05, Fitness: 0.10 };
        const discount = discounts[category] ?? 0;
        return {
            original: basePrice,
            discount: parseFloat((basePrice * discount).toFixed(2)),
            final: parseFloat((basePrice * (1 - discount)).toFixed(2))
        };
    }
}));

// Logger singleton
store.singleton("logger", () => Log);

// Product service: transient, depends on repo and price service
store.register("productService", (c) => {
    const repo = c.get<ReturnType<typeof createProductRepo>>("productRepo");
    const pricing = c.get<ReturnType<typeof createPriceService>>("priceService");
    const log = c.get<typeof Log>("logger");

    return {
        getById: (id: number) => {
            log.debug("ProductService.getById", { id });
            const product = repo.findById(id);
            if (!product) return null;
```

```
            const price = pricing.calculate(product.basePrice, product.category);
            return { ...product, price };
        }
    };
});

// Type helpers (not real -- illustrative)
function createProductRepo() { return {} as any; }
function createPriceService() { return {} as any; }
```

`src/routes/store.ts`:

```
import { Router } from "tina4-nodejs";
import { store } from "../store/container";

Router.get("/api/store/{id:int}", async (req, res) => {
    const id = req.params.id;
    const productService = store.get<{ getById: (id: number) => unknown }>("productService");
    const product = productService.getById(id);

    if (!product) {
        return res.status(404).json({ error: `Product ${id} not found` });
    }

    return res.json(product);
});
```

---

13. Gotchas

1. Registering after first get()

If you call `get("db")` before `register("db", ...)`, the container throws. There is no lazy fallback.

Fix: Register all services at application startup before any code calls `get()`. Keep all registrations in one file (`src/container.ts`) that is imported first.

2. Circular dependencies

`serviceA` depends on `serviceB`, which depends on `serviceA`. The container will recurse infinitely and crash.

Fix: Circular dependencies indicate a design problem. Extract shared logic into a third service that neither depends on the other.

3. Singletons holding stale state

A singleton database connection created at startup may become invalid if the database server restarts. Every `get()` returns the same dead connection.

Fix: Implement reconnect logic inside the singleton service, not outside it. The container manages the lifecycle; the service manages its own health.

4. Over-registering transient services

Registering everything as transient creates a new instance on every `get()` call, including heavyweight services with connection pools.

Fix: Use `singleton()` for anything that is expensive to create or that holds persistent state. Use `register()` only for services that must be fresh per request.

Service Runner

1. Work That Never Stops

Some tasks run once per request. Some tasks run forever. A cache warmer that refreshes data every 60 seconds. A queue drainer that processes jobs in a loop.

These are background services -- long-running loops that live alongside your HTTP server. They start when the application starts. They stop when the application stops. If they crash, they restart.

Tina4's Service Runner manages background services. Define a service. Register it. The runner handles lifecycle, restart-on-crash, and clean shutdown.

---

2. Defining a Service

A service is a class that extends `Service` and implements a `run()` method:

```
import { Service } from "tina4-nodejs";

class CacheWarmer extends Service {
    name = "CacheWarmer";
    interval = 60_000;  // milliseconds between runs

    async run() {
        console.log("Warming cache...");
        await warmProductCache();
        await warmCategoryCache();
        console.log("Cache warm");
    }
}
```

The `run()` method is called once per interval. When it completes, the runner waits `interval` milliseconds before calling it again.

---

3. Registering and Starting Services

```
import { ServiceRunner } from "tina4-nodejs";
import { CacheWarmer } from "./services/CacheWarmer";
import { HealthMonitor } from "./services/HealthMonitor";
import { InvoiceScheduler } from "./services/InvoiceScheduler";

const runner = new ServiceRunner();

runner.register(new CacheWarmer());
runner.register(new HealthMonitor());
runner.register(new InvoiceScheduler());

runner.start();
```

`runner.start()` starts all registered services in parallel. Each service gets its own loop. They do not block each other.

---

4. Service Lifecycle

```
register() --> IDLE
start()     --> RUNNING --> run() --> wait(interval) --> run() --> ...
stop()      --> STOPPING --> (current run() finishes) --> STOPPED
crash       --> ERROR    --> wait(restartDelay)       --> RUNNING
```

Stopping a Service

```
// Stop a specific service by name
runner.stop("CacheWarmer");

// Stop all services
runner.stopAll();
```

A stopped service waits for its current `run()` call to complete before shutting down. It does not kill mid-execution.

Restart on Crash

If `run()` throws an uncaught exception, the runner logs the error and restarts the service after a configurable delay:

```
class UnreliableService extends Service {
    name = "UnreliableService";
    interval = 5_000;
    restartDelay = 10_000;  // Wait 10 seconds before restarting after a crash

    async run() {
        if (Math.random() < 0.2) {
            throw new Error("Random failure (20% chance)");
        }
        await doWork();
    }
}
```

The default `restartDelay` is 5 seconds. Set it higher for services that connect to external systems (avoid hammering a downed dependency).

---

5. Service Options

Option	Default	Description
`name`	class name	Identifier for logging and `stop()`
`interval`	`60_000`	Milliseconds between `run()` calls
`restartDelay`	`5_000`	Delay before restarting after a crash
`runOnStart`	`true`	Whether to call `run()` immediately on start
`maxRestarts`	`Infinity`	Stop restarting after this many crashes

```
class HourlyReport extends Service {
    name = "HourlyReport";
    interval = 3_600_000;  // 1 hour
    runOnStart = false;    // Wait a full hour before first run
    maxRestarts = 5;       // Give up after 5 consecutive crashes
    restartDelay = 30_000; // Wait 30 seconds between restarts

    async run() {
        await generateHourlyReport();
        await sendReportEmail();
    }
}
```

---

6. A Real-World Example: Queue Drainer

A background service that continuously processes a job queue:

```
import { Service, Queue } from "tina4-nodejs";

class EmailQueueDrainer extends Service {
    name = "EmailQueueDrainer";
    interval = 1_000;  // Check every second
    private queue = new Queue({ topic: "emails" });

    async run() {
        const job = this.queue.pop();

        if (job === null) {
            return;  // Nothing to process this tick
        }

        try {
            await sendEmail(job.payload.to, job.payload.subject, job.payload.body);
            job.complete();
        } catch (err) {
```

```
            const message = err instanceof Error ? err.message : String(err);
            job.retry(30);  // Retry after 30 seconds
            console.error(`Email job failed: ${message}`);
        }
    }
}
```

The service pops one job per second. Errors do not crash the service -- they call `job.retry()` and the service continues to the next tick.

---

7. Health Check Endpoint

Expose service status via an HTTP endpoint:

```
import { Router, ServiceRunner } from "tina4-nodejs";

const runner = new ServiceRunner();

Router.get("/api/health/services", async (req, res) => {
    const status = runner.status();
    return res.json(status);
});
```

```
{
  "services": [
    {
      "name": "CacheWarmer",
      "state": "RUNNING",
      "lastRun": "2026-04-02T08:00:00.000Z",
      "runs": 48,
      "errors": 0,
      "nextRun": "2026-04-02T08:01:00.000Z"
    },
    {
      "name": "EmailQueueDrainer",
      "state": "RUNNING",
      "lastRun": "2026-04-02T08:00:59.123Z",
      "runs": 2880,
      "errors": 3,
      "nextRun": "2026-04-02T08:01:00.123Z"
    }
  ]
}
```

---

8. Integrating with the Application

Register services in your main entry point so they start alongside the HTTP server:

```
import { Tina4 } from "tina4-nodejs";
import { ServiceRunner } from "tina4-nodejs";
import { CacheWarmer } from "./services/CacheWarmer";
import { EmailQueueDrainer } from "./services/EmailQueueDrainer";
import { HealthMonitor } from "./services/HealthMonitor";
import "./routes/index";
```

```
const runner = new ServiceRunner();
runner.register(new CacheWarmer());
runner.register(new EmailQueueDrainer());
runner.register(new HealthMonitor());

const app = new Tina4();

app.start(() => {
    // Start background services after the server is up
    runner.start();
    console.log("HTTP server and background services started");
});

// Clean shutdown: stop services before the process exits
process.on("SIGTERM", () => {
    runner.stopAll();
    process.exit(0);
});

process.on("SIGINT", () => {
    runner.stopAll();
    process.exit(0);
});
```

---

9. Exercise: Daily Report Service

Build a background service that generates a daily sales summary every 24 hours.

Requirements

- Create a `DailySalesReport` service with a 24-hour interval
- The service should compute: total orders, total revenue, and top-selling category (use in-memory mock data)
- Log the report with `Log.info()` and optionally store it
- Expose a `GET /api/reports/last` endpoint that returns the most recent report

Test by setting the interval to 5 seconds so you can see it run:

```
# Start the server and watch the console for report logs
# After 5 seconds:
curl http://localhost:7148/api/reports/last
```

---

10. Solution

`src/services/DailySalesReport.ts`:

```
import { Service, Log } from "tina4-nodejs";

interface SalesReport {
    generatedAt: string;
    totalOrders: number;
    totalRevenue: number;
```

```
    topCategory: string;
}

const MOCK_ORDERS = [
    { id: 1, category: "Electronics", total: 79.99 },
    { id: 2, category: "Fitness", total: 29.99 },
    { id: 3, category: "Electronics", total: 549.99 },
    { id: 4, category: "Kitchen", total: 49.99 },
    { id: 5, category: "Electronics", total: 39.99 },
    { id: 6, category: "Fitness", total: 14.99 }
];

export let lastReport: SalesReport | null = null;

export class DailySalesReport extends Service {
    name = "DailySalesReport";
    interval = 86_400_000;  // 24 hours (use 5000 for testing)

    async run() {
        const totalOrders = MOCK_ORDERS.length;
        const totalRevenue = MOCK_ORDERS.reduce((sum, o) => sum + o.total, 0);

        const categoryCounts: Record<string, number> = {};
        for (const order of MOCK_ORDERS) {
            categoryCounts[order.category] = (categoryCounts[order.category] ?? 0) + 1;
        }

        const topCategory = Object.entries(categoryCounts)
            .sort((a, b) => b[1] - a[1])[0]?.[0] ?? "None";

        lastReport = {
            generatedAt: new Date().toISOString(),
            totalOrders,
            totalRevenue: parseFloat(totalRevenue.toFixed(2)),
            topCategory
        };

        Log.info("Daily sales report generated", {
            totalOrders,
            totalRevenue: lastReport.totalRevenue,
            topCategory
        });
    }
}
```

src/routes/reports.ts:

```
import { Router } from "tina4-nodejs";
import { lastReport } from "../services/DailySalesReport";

Router.get("/api/reports/last", async (req, res) => {
    if (!lastReport) {
        return res.status(404).json({ error: "No report generated yet" });
    }
    return res.json(lastReport);
});
```

src/index.ts:

```
import { Tina4, ServiceRunner } from "tina4-nodejs";
```

```
import { DailySalesReport } from "./services/DailySalesReport";
import "./routes/reports";

const runner = new ServiceRunner();
runner.register(new DailySalesReport());

const app = new Tina4();
app.start(() => {
    runner.start();
});
```

---

11. Gotchas

1. Services run immediately by default

If `runOnStart` is `true` (the default), `run()` is called as soon as `runner.start()` is invoked. For services that depend on a warm database or a running HTTP server, this can fail.

Fix: Set `runOnStart = false` if the service needs the application to be fully up. Or start the runner inside the `app.start()` callback, which fires after the server is ready.

2. Uncaught exceptions outside run() crash the process

If an exception escapes your `run()` method without being caught, the runner catches it and restarts the service. But an unhandled promise rejection at the top level of your service class bypasses the runner's error handling entirely.

Fix: Wrap the entire body of `run()` in try/catch. Never let async errors escape.

3. interval is wall-clock delay between runs, not between start times

If `run()` takes 45 seconds and `interval` is 60 seconds, the service runs every 105 seconds -- not every 60. This is intentional: no overlapping runs.

Fix: If you need wall-clock scheduling (exactly at :00 each minute), compute the time until the next scheduled run and use that as a dynamic interval.

4. Services do not share state safely

Two services reading and writing the same global variable without synchronization can produce inconsistent data.

Fix: Use a shared service (e.g., a singleton from the DI container) as the data layer. Services interact through it, not directly through shared globals.

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:7148/__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:7148/__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:7148/__dev/mcp/sse` -- returns the message endpoint URL
- **Message endpoint**: `POST http://localhost:7148/__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/core` 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, Node.js 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 `package.json`)
`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:7148/__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 `initDatabase()` in your server setup
- File tools are sandboxed to the project directory
- `database_execute` only works on localhost

Remote server -- MCP disabled:

- This is intentional. Set `TINA4_MCP_REMOTE=true` only on staging environments you trust
- Never enable MCP on production servers

Building Custom MCP Servers

1. Beyond Dev Tools

Chapter 27 covered the built-in MCP server that ships with Tina4. It exposes framework internals for AI-assisted development. This chapter goes further: you build your own MCP servers that expose your application's business logic.

A CRM system exposes customer lookup. An accounting system exposes invoice queries. A warehouse system exposes inventory checks. Any domain logic that an AI assistant should access becomes an MCP tool.

---

2. Creating an MCP Server

Import `McpServer` and create an instance on any path:

```
import { McpServer } from "@tina4/core";

const mcp = new McpServer("/api/my-tools", "My App Tools", "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 your server setup:

```
mcp.registerRoutes(router);
```

---

3. Registering Tools with mcpTool

The `mcpTool` function registers a handler as an MCP tool. Parameter metadata becomes the input schema:

```
import { McpServer, mcpTool, schemaFromParams } from "@tina4/core";

const mcp = new McpServer("/crm/mcp", "CRM Tools");

mcpTool("lookup_customer", "Find a customer by email", mcp, [
  { name: "email", type: "string" },
])((args) => {
  return db.fetchOne("SELECT * FROM customers WHERE email = ?", [args.email]);
});

mcpTool("recent_orders", "Get recent orders for a customer", mcp, [
  { name: "customer_id", type: "integer" },
  { name: "limit", type: "integer", default: 10 },
])((args) => {
  return db.fetch(
    "SELECT * FROM orders WHERE customer_id = ? ORDER BY created_at DESC",
    [args.customer_id], { limit: (args.limit as number) || 10 }
```

```
  );
});
```

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 mcpResource

Resources are read-only data endpoints. They expose reference data that AI assistants can browse:

```
import { mcpResource } from "@tina4/core";

mcpResource("crm://product-catalog", "All active products", "application/json", mcp)(
  () => db.fetch("SELECT id, name, price, category FROM products WHERE active = 1")
);

mcpResource("crm://tax-rates", "Current tax rates by region", "application/json", mcp)(
  () => db.fetch("SELECT region, rate FROM tax_rates")
);
```

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:

```
import { McpServer, schemaFromParams } from "@tina4/core";

const mcp = new McpServer("/accounting/mcp", "Accounting Tools");

class AccountingService {
  constructor(private db: any) {}

  lookup(invoiceNo: string) {
```

```
    return this.db.fetchOne(
      "SELECT * FROM invoices WHERE invoice_no = ?", [invoiceNo]
    );
  }

  balances(minAmount = 0.0) {
    return this.db.fetch(
      "SELECT * FROM invoices WHERE paid = 0 AND total >= ?", [minAmount]
    );
  }

  summary(year: number, month: number) {
    return this.db.fetchOne(
      "SELECT SUM(total) as revenue, COUNT(*) as invoice_count " +
      "FROM invoices WHERE strftime('%Y', created_at) = ? " +
      "AND strftime('%m', created_at) = ?",
      [String(year), String(month).padStart(2, "0")]
    );
  }
}

const svc = new AccountingService(db);

mcp.registerTool("invoice_lookup",
  (args) => svc.lookup(args.invoice_no as string),
  "Find an invoice by number",
  schemaFromParams([{ name: "invoice_no", type: "string" }])
);

mcp.registerTool("outstanding_balances",
  (args) => svc.balances((args.min_amount as number) || 0),
  "List all unpaid invoices",
  schemaFromParams([{ name: "min_amount", type: "number", default: 0.0 }])
);

mcp.registerTool("monthly_summary",
  (args) => svc.summary(args.year as number, args.month as number),
  "Revenue summary for a month",
  schemaFromParams([{ name: "year", type: "integer" }, { name: "month", type: "integer" }])
);
```

---

6. Securing MCP Endpoints

By default, developer MCP servers are public. Add authentication using Tina4 middleware:

```
// Secure the MCP routes via middleware
import { secured } from "@tina4/core";

// Apply auth middleware before registering routes
mcp.registerRoutes(router); // then protect the path with middleware
```

Or check the bearer token inside individual tools:

```
import { Auth } from "@tina4/core";

mcp.registerTool("sensitive_data",
  (args) => {
```

```
    const payload = Auth.validToken(args.token as string, secret);
    if (!payload) return { error: "Unauthorized" };
    return db.fetch("SELECT * FROM sensitive_table");
  },
  "Access restricted data",
  schemaFromParams([{ name: "token", type: "string" }])
);
```

---

7. Testing MCP Tools

Test tool functions directly, or test via the MCP protocol:

```
// Test the tool function directly
function testLookupCustomer() {
  const result = db.fetchOne("SELECT * FROM customers WHERE email = ?", ["alice@example.com"]);
  assert(result !== null);
  assert(result.email === "alice@example.com");
}

// Test via MCP protocol
function testMcpToolCall() {
  const resp = JSON.parse(mcp.handleMessage({
    jsonrpc: "2.0",
    id: 1,
    method: "tools/call",
    params: {
      name: "lookup_customer",
      arguments: { email: "alice@example.com" },
    },
  }));
  assert("result" in resp);
  const content = resp.result.content[0].text;
  assert(content.toLowerCase().includes("alice"));
}
```

---

8. Complete Example: CRM MCP Server

Here is a full working example -- a CRM system with customer, order, and product tools:

```
// src/routes/mcp/setup.ts
import { McpServer, mcpTool, mcpResource, schemaFromParams } from "@tina4/core";
import { initDatabase } from "@tina4/orm";

const db = await initDatabase({ url: "sqlite:///crm.db" });

// Create MCP server
const crmMcp = new McpServer("/crm/mcp", "CRM Assistant", "1.0.0");

// Tools
mcpTool("find_customer", "Search customers by name or email", crmMcp, [
  { name: "query", type: "string" },
])((args) => {
  const q = args.query as string;
  return db.fetch(
```

```
    "SELECT * FROM customers WHERE name LIKE ? OR email LIKE ?",
    [`%${q}%`, `%${q}%`]
  );
});

mcpTool("customer_orders", "Get all orders for a customer", crmMcp, [
  { name: "customer_id", type: "integer" },
])((args) => {
  return db.fetch(
    "SELECT o.*, GROUP_CONCAT(oi.product_name) as items " +
    "FROM orders o LEFT JOIN order_items oi ON o.id = oi.order_id " +
    "WHERE o.customer_id = ? GROUP BY o.id ORDER BY o.created_at DESC",
    [args.customer_id]
  );
});

mcpTool("create_note", "Add a note to a customer record", crmMcp, [
  { name: "customer_id", type: "integer" },
  { name: "note", type: "string" },
])((args) => {
  db.execute("INSERT INTO customer_notes (customer_id, note) VALUES (?, ?)",
    [args.customer_id, args.note]);
  return { success: true };
});

// Resources
mcpResource("crm://products", "Product catalog", "application/json", crmMcp)(
  () => db.fetch("SELECT * FROM products WHERE active = 1")
);

mcpResource("crm://stats", "CRM statistics", "application/json", crmMcp)(() => {
  const customers = db.fetch("SELECT COUNT(*) as count FROM customers");
  const orders = db.fetch("SELECT COUNT(*) as count, SUM(total) as revenue FROM orders");
  return {
    customers: customers[0]?.count ?? 0,
    orders: orders[0]?.count ?? 0,
    revenue: orders[0]?.revenue ?? 0,
  };
});

// Register routes
crmMcp.registerRoutes(router);

export { crmMcp };
```

Connect Claude Code to `http://localhost:7148/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** -- objects and arrays, not formatted strings. Let the AI format for the user

• **Secure production endpoints** -- use middleware for any MCP server that runs outside localhost

• **Test tools directly** -- call the TypeScript function in your test suite, not just through the MCP protocol

Dev Tools

1. Debugging at 2am

2am. Production monitoring pings you. A 500 error on checkout. You pull up the dev dashboard. The request inspector shows the failing request. The stack trace points to line 47 of `src/routes/checkout.ts` -- a null reference on the shipping address because the user skipped the form field. You add a null check. Push the fix. Back to sleep. Total time: 30 seconds.

Tina4's dev tools are not an afterthought. They ship with the framework from day one. When `TINA4_DEBUG=true`, you get a development dashboard, an error overlay with source code, live reload, a request inspector with replay, a SQL query runner, a queue monitor, and system info -- all without installing extra packages.

---

2. Enabling the Dev Dashboard

Set `TINA4_DEBUG=true` in your `.env`:

```
TINA4_DEBUG=true
```

Restart your server and navigate to:

```
http://localhost:7148/__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 Node.js version
- **Node.js version** -- The running Node.js version and loaded packages
- **Uptime** -- How long the server has been running
- **Memory usage** -- Current and peak memory consumption
- **Database status** -- Connection status, database engine, file size (for SQLite)
- **Environment** -- Current `.env` variables (sensitive values masked)
- **Project structure** -- Directory listing of your project with file counts

Check here first when something feels off. Is the database connected? Is the right Node.js version running? Are the environment variables loaded?

Request Inspector

- Recent HTTP requests with method, path, status, duration
- Click any request to see headers, body, database queries, template renders

Error Log

- Unhandled exceptions with stack traces
- Occurrence counts and timestamps

Queue Manager

- Queue status: pending, reserved, completed, failed, dead counts
- Recent jobs with status and duration

WebSocket Monitor

- Active WebSocket connections with metadata
- Message history

Routes

- All registered routes with methods, paths, middleware, auth status

Mail

- Intercepted emails with To, Subject, HTML body, attachments

---

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	Node.js 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:

```
import { Router } from "tina4-nodejs";

Router.get("/api/data", async (req, res) => {
    return res.header("X-Debug-Toolbar", "off").json({ data: "no toolbar" });
});
```

This is useful for API endpoints that return JSON -- the toolbar only matters for HTML pages.

---

5. Error Overlay

An unhandled exception occurs. Tina4 does not show a generic "500 Internal Server Error" page. It shows a detailed error overlay:

- **Exception type and message** -- What went wrong, in plain language
- **Stack trace** -- Every function call that led to the error, from entry point to crash
- **Source code** -- The TypeScript 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:

```
Router.get("/api/users/{userId}", async (req, res) => {
    const userId = req.params.userId;
    const user = await getUserFromDb(userId);
    return res.json({ name: user.name, email: user.email });
});
```

If `getUserFromDb()` returns `null` for a missing user, the error overlay shows:

```
TypeError: Cannot read properties of null (reading 'name')

  File: src/routes/users.ts, line 4

  2 |     const userId = req.params.userId;
  3 |     const user = await getUserFromDb(userId);
> 4 |     return res.json({ name: user.name, email: user.email });
  5 | });
```

```
  Request: GET /api/users/999
  Headers: {"Accept": "application/json"}
```

The highlighted line makes the cause obvious. `user` is `null` and the code accesses `.name` on it.

Error Overlay in Production

When `TINA4_DEBUG=false`, the error overlay is disabled. Users see a clean error page. Full error details go to `logs/error.log` for later investigation. Custom error pages go in `src/templates/errors/`:

```
src/templates/errors/404.html
src/templates/errors/500.html
```

---

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 Node.js v3.11.12
  HTTP server running at http://0.0.0.0:7148
  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 (`watcher.ts` using `fs.watch` 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`, `dist`, `target`, `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 Node server. The server keeps running — no process restart for file edits.
- `DevAdmin` bumps its in-memory `_reloadMtime` 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. The server process stays up throughout.

---

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.

---

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 function name
- The payload (serialized arguments)
- When it was enqueued
- When it started processing (if active)
- The error message and stack trace (if failed)
- How many times it has been retried

You can also take action:

- **Retry a failed job** -- Click "Retry" to move it back to the pending queue
- **Delete a job** -- Remove it from any queue
- **Pause/resume the queue** -- Stop processing without losing jobs

The queue monitor gives you a window into your background work. A job fails. You see the error. You fix the code. You hit Retry. The job processes. No log file hunting. No guesswork about what payload caused the failure.

---

11. System Info

The System Info tab shows detailed information about your environment:

- **Node.js Configuration** -- Version, installed packages, 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 Node.js version. A missing package. A misconfigured environment variable. The answer is in the panel.

---

12. Logging

```
import { Log } from "tina4-nodejs";

Log.debug("Debug message");
Log.info("Info message");
Log.warning("Warning message");
Log.error("Error message");
```

Log levels are controlled by `TINA4_LOG_LEVEL` in `.env`:

Level	Shows
`ALL`	Everything
`DEBUG`	Debug and above
`INFO`	Info and above
`WARNING`	Warning and above
`ERROR`	Errors only
`NONE`	Nothing

Logs are written to `logs/app.log` and to stdout.

---

13. Health Check

```
http://localhost:7148/health
```

Returns system status: database connectivity, uptime, and version. Your monitoring tools and load balancers hit this endpoint.

---

14. Exercise: Debug a Failing Route

Your colleague wrote a route that fails. Use the dev tools to find and fix the bug.

Setup

Create `src/routes/buggy.ts`:

```
import { Router } from "tina4-nodejs";
import { Database } from "tina4-nodejs/orm";
import { cacheGet, cacheSet } from "tina4-nodejs";

Router.get("/api/buggy/users", async (req, res) => {
    const users = await cacheGet("all_users");
    if (users) {
        return res.json({ users, source: "cache" });
    }

    // Bug 1: This query has an error
    const db = Database.getConnection();
    const result = await db.fetchAll("SELCT * FROM users ORDER BY name");

    await cacheSet("all_users", result, 300);
    return res.json({ users: result, source: "database" });
});

Router.post("/api/buggy/users", async (req, res) => {
    const body = req.body;

    // Bug 2: Missing validation
    const user = new User();
    user.name = body.name;
    user.email = body.email;
    await user.save();

    // Bug 3: Wrong status code
    return res.json({ message: "User created", user: user.toDict() });
});
```

Requirements

- Open the dev dashboard at `http://localhost:7148/__dev`
- Hit the `GET /api/buggy/users` endpoint and find Bug 1 using the error overlay
- Hit the `POST /api/buggy/users` endpoint without a body and find Bug 2 using the request inspector
- Fix all three bugs:
- Bug 1: Fix the SQL syntax error
- Bug 2: Add validation for required fields
- Bug 3: Return 201 instead of 200
- Use Request Replay to verify each fix

Solution

```
import { Router } from "tina4-nodejs";
import { Database } from "tina4-nodejs/orm";
import { cacheGet, cacheSet } from "tina4-nodejs";

Router.get("/api/buggy/users", async (req, res) => {
    const users = await cacheGet("all_users");
    if (users) {
        return res.json({ users, source: "cache" });
    }

    const db = Database.getConnection();
    const result = await db.fetchAll("SELECT * FROM users ORDER BY name");  // Fixed: SELCT -> S

    await cacheSet("all_users", result, 300);
    return res.json({ users: result, source: "database" });
});

Router.post("/api/buggy/users", async (req, res) => {
    const body = req.body;

    // Fixed: Added validation
    if (!body.name || !body.email) {
        return res.status(400).json({ error: "Name and email are required" });
    }

    const user = new User();
    user.name = body.name;
    user.email = body.email;
    await user.save();

    return res.status(201).json({ message: "User created", user: user.toDict() });  // Fixed: 20
});
```

The dev tools made each bug visible. The error overlay showed the SQL syntax error with the exact line. The request inspector showed the missing body causing a null reference. Request Replay let you re-send the fixed requests without leaving the dashboard.

---

15. Gotchas

1. Dev Dashboard Accessible on Network

Problem: Anyone on your network can access the dev dashboard.

Cause: `TINA4_DEBUG=true` makes the dashboard available at `/__dev`.

Fix: In production, set `TINA4_DEBUG=false` to disable the dashboard. In shared development environments, restrict network access.

2. Live Reload Causes Connection Drops

Problem: WebSocket connections drop every time you save a file.

Cause: The server restarts on file change, which closes all active connections.

Fix: Use the file-based routing approach where only affected routes reload. For WebSocket development, clients with frond.js reconnect automatically after restart.

3. Error Overlay Shows in Production

Problem: Users see TypeScript 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. 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: Ensure your queue consumers are running. The queue monitor reflects the state of the queue storage. If no consumer processes jobs, they sit in "Pending" and never move to "Active" or "Completed."

8. Debug Mode in Version Control

Fix: Add `.env` to `.gitignore`.

---

9. Logs Fill Up Disk

Fix: Set `TINA4_LOG_LEVEL=WARNING` in production.

10. Performance Overhead

Fix: Debug mode adds overhead. Always disable in production.

Tina4 CLI

1. Getting a New Developer Up to Speed

Monday morning. A new developer joins your team. You hand them the repo URL. By 10am they have a running project, a new database model, CRUD routes, a migration, and a deployment to staging. All from the command line. No documentation scavenger hunt. No boilerplate copy-paste.

The Tina4 CLI is a single Rust binary. It manages all four Tina4 frameworks (PHP, Python, Ruby, Node.js). The commands are identical across languages. Learn the CLI for Node.js. You know it for Python.

---

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: Node.js (package.json)
  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
  npm install
  tina4 serve
```

Language Detection

The CLI detects the language from existing files:

File Present	Language
`composer.json`	PHP

`pyproject.toml` or `requirements.txt`	Python
`Gemfile`	Ruby
`package.json`	Node.js

If no language-specific file exists, the CLI asks:

```
tina4 init my-project
```

```
No language detected. Which language?
  1. PHP
  2. Python
  3. Ruby
  4. Node.js
> 4

Creating Tina4 Node.js project in ./my-project ...
```

Explicit Language Selection

Skip the prompt by specifying the language:

```
tina4 init nodejs my-project
```

This creates a Node.js project with `package.json`, `app.ts`, 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 Node.js v3.10.3
  HTTP server running at http://0.0.0.0:7148
  WebSocket server running at ws://0.0.0.0:7148
  Live reload enabled
  Press Ctrl+C to stop
```

`tina4 serve` detects the language and starts the appropriate server. For Node.js, it spawns the Tina4 Node.js 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) set `TINA4_OVERRIDE_CLIENT=true` in `.env` and only then run:

```
TINA4_OVERRIDE_CLIENT=true npx tsx app.ts
```

This bypasses SCSS compilation and live reload — use it only when the Rust CLI is genuinely unavailable.

---

4. tina4 generate model -- ORM Scaffolding

The `generate model` command creates an ORM model file and a matching migration. One command produces both.

```
tina4 generate model Product
```

```
Created src/orm/Product.ts
Created migrations/20260322120000_create_products_table.sql
```

The generated model:

```
import { BaseModel } from "tina4-nodejs/orm";

export class Product extends BaseModel {
    static tableName = "products";
    static primaryKey = "id";

    id!: number;
    createdAt!: string;
    updatedAt!: string;
}
```

The generated migration:

```
-- UP
CREATE TABLE products (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    created_at TEXT DEFAULT CURRENT_TIMESTAMP,
    updated_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,inStock:bool"
```

The generated model includes all the fields:

```
import { BaseModel } from "tina4-nodejs/orm";

export class Product extends BaseModel {
```

```
    static tableName = "products";
    static primaryKey = "id";

    id!: number;
    name!: string;
    price!: number;
    category!: string;
    inStock!: boolean;
    createdAt!: string;
    updatedAt!: string;
}
```

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,
    updated_at TEXT DEFAULT CURRENT_TIMESTAMP
);

-- DOWN
DROP TABLE IF EXISTS products;
```

Field Types

CLI Type	TypeScript Type	SQLite Column
`string`	`string`	`TEXT`
`int`	`number`	`INTEGER`
`float`	`number`	`REAL`
`bool`	`boolean`	`INTEGER`
`text`	`string`	`TEXT`
`date`	`string`	`TEXT`

Options

Flag	Description	Example
`--fields`	Comma-separated field definitions	`--fields "name:string,price:float"`
`--auto-crud`	Enable auto-CRUD on the model	`--auto-crud`
`--soft-delete`	Add soft delete support	`--soft-delete`
`--no-migration`	Skip migration generation	`--no-migration`
`--with-route`	Also generate a CRUD route file	`--with-route`

---

5. tina4 generate route -- CRUD Route Scaffolding

The `generate route` command creates a complete CRUD route file with all five REST endpoints. It reads the model's properties and builds routes with proper imports and response handling.

```
tina4 generate route products
```

```
Created src/routes/products.ts
```

The generated route file:

```
import { Router } from "tina4-nodejs";

Router.get("/api/products", async (req, res) => {
    const page = parseInt(req.query.page ?? "1", 10);
    const perPage = parseInt(req.query.per_page ?? "20", 10);
    const offset = (page - 1) * perPage;

    const products = await Product.findAll({ limit: perPage, offset });
    const results = products.map(p => p.toDict());

    return res.json({
        data: results,
        page,
        per_page: perPage,
        count: results.length
    });
});

Router.get("/api/products/{productId:int}", async (req, res) => {
    const product = await Product.findById(req.params.productId);

    if (product === null) {
        return res.status(404).json({ error: "Product not found" });
    }

    return res.json(product.toDict());
```

```
});

Router.post("/api/products", async (req, res) => {
    const body = req.body;

    const product = new Product();
    product.name = body.name ?? "";
    product.price = parseFloat(body.price ?? "0");
    product.category = body.category ?? "";
    product.inStock = Boolean(body.inStock ?? false);
    await product.save();

    return res.status(201).json(product.toDict());
});

Router.put("/api/products/{productId:int}", async (req, res) => {
    const product = await Product.findById(req.params.productId);

    if (product === null) {
        return res.status(404).json({ error: "Product not found" });
    }

    const body = req.body;
    if (body.name !== undefined) product.name = body.name;
    if (body.price !== undefined) product.price = parseFloat(body.price);
    if (body.category !== undefined) product.category = body.category;
    if (body.inStock !== undefined) product.inStock = Boolean(body.inStock);
    await product.save();

    return res.json(product.toDict());
});

Router.delete("/api/products/{productId:int}", async (req, res) => {
    const product = await Product.findById(req.params.productId);

    if (product === null) {
        return res.status(404).json({ error: "Product not found" });
    }

    await product.delete();
    return res.status(204).end();
});
```

The generator reads the model's properties and creates routes with type casting and null checks. Customize the generated code immediately. It is regular TypeScript, not magic.

Options

Flag	Description	Example
`--prefix`	Custom route prefix (default: `/api`)	`--prefix /api/v2`
`--middleware`	Add middleware to all routes	`--middleware authMiddleware`

---

6. tina4 generate migration -- Migration Scaffolding

The `generate migration` command creates a timestamped migration file with `UP` and `DOWN` sections. The timestamp ensures migrations run in order.

```
tina4 generate migration add_category_to_products
```

```
Created migrations/20260322120500_add_category_to_products.sql
```

The generated file:

```
-- UP
-- Add your forward migration SQL here


-- DOWN
-- Add your rollback migration SQL here
```

Fill in the SQL:

```
-- UP
ALTER TABLE products ADD COLUMN category TEXT DEFAULT '';

-- DOWN
ALTER TABLE products DROP COLUMN category;
```

The timestamp prefix (`20260322120500`) ensures migrations run in order. Each migration runs once. The framework tracks which ones have been applied.

When to Generate a Migration

Generate a new migration when you need to change the database schema after the initial model migration. Adding a column. Creating an index. Renaming a table. Each change gets its own migration file with a unique timestamp.

---

7. tina4 generate middleware -- Middleware Scaffolding

The `generate middleware` command creates a middleware function with the correct signature and a placeholder for your logic.

```
tina4 generate middleware rateLimit
```

```
Created src/middleware/rateLimit.ts
```

The generated file:

```
import { Middleware } from "tina4-nodejs";

async function rateLimit(req: any, res: any, next: Function) {
    // Add your middleware logic here

    // Continue to the next middleware or route handler
    return next();

    // Or return early to block the request:
    // return res.status(429).json({ error: "Rate limit exceeded" });
}
```

```
export default rateLimit;
```

The middleware is a named function. Reference it in route definitions:

```
Router.get("/api/data", async (req, res) => {
    return res.json({ data: "protected" });
}, "rateLimit");
```

---

8. Generate All at Once

Combine flags to generate multiple files in a single command:

```
tina4 generate model Product --with-route --with-migration
```

```
Created src/orm/Product.ts
Created src/routes/products.ts
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] Node.js v20.11.1 detected
  [OK] npm package manager found
  [OK] tina4-nodejs package installed (v3.10.3)
  [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:

- Node.js version and package manager
- Tina4 package installation and version
- `.env` file existence and critical variables
- Database connectivity
- Directory structure

- Missing files or configurations
- AI tool context files
- Git configuration

The warnings give actionable advice. If your database is not configured, it tells you exactly what to add to `.env`.

---

10. tina4 test -- Running Tests

```
tina4 test

Running tests...

  ProductTest
    [PASS] test_create_product
    [PASS] test_load_product

  2 tests, 2 passed, 0 failed (0.12s)
```

This runs all tests in the `tests/` directory. See Chapter 18 for full testing documentation.

Test Options

```
tina4 test --filter ProductTest           # Specific test class
tina4 test --verbose                      # Show assertion details
```

Or use npm:

```
npm test
```

---

11. tina4 routes -- Route Listing

See all registered routes in your project:

```
tina4 routes

Registered Routes:

  Method  Path                        Handler                     Middleware
  ------  --------------------------  --------------------------  ----------
  GET     /health                     healthCheck                 -
  GET     /api/products               listProducts                ResponseCache:300
  GET     /api/products/{productId}   getProduct                  -
  POST    /api/products               createProduct               authMiddleware
  PUT     /api/products/{productId}   updateProduct               authMiddleware
  DELETE  /api/products/{productId}   deleteProduct               authMiddleware
  GET     /api/auth/login             -                           -
  POST    /api/auth/login             login                       -
  GET     /admin                      adminDashboard              -

  9 routes registered
```

This is useful for verifying that your routes are registered and for finding the handler function for a specific URL.

Filtering

```
tina4 routes --method POST          # Filter by HTTP method
tina4 routes --filter products      # Filter by path pattern
tina4 routes --middleware auth      # Filter by middleware
```

When debugging routing issues, check here first. If a route does not match, `tina4 routes` shows whether it was registered and what middleware is attached.

---

12. tina4 migrate -- Database Migrations

Run pending migrations:

```
tina4 migrate
```

```
Running migrations...
  [UP] 20260322000100_create_users_table.sql
  [UP] 20260322000200_create_products_table.sql
  [UP] 20260322000300_add_category_to_products.sql

  3 migrations applied
```

Rollback

```
tina4 migrate --rollback
```

```
Rolling back last migration...
  [DOWN] 20260322000300_add_category_to_products.sql

  1 migration rolled back
```

Migration Table Auto-Upgrade

If your project was created with an 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
```

---

13. tina4 queue -- Queue Management

```
tina4 queue work                        # Start processing jobs
tina4 queue work --queue send-email     # Process specific queue
tina4 queue:dead                        # List dead letter jobs
tina4 queue:retry 42                    # Retry a dead job
tina4 queue:clear --older-than 7d       # Clear old dead jobs
```

---

14. tina4 build -- Production Build

```
tina4 build
```

Compiles TypeScript to JavaScript. Bundles for production. Optimizes:

```
Building for production...
  Compiled 47 TypeScript files
  Output: dist/
  Size: 245KB

Ready for deployment:
  node dist/app.js
```

---

15. Custom CLI Commands

Create custom seed scripts as standalone TypeScript files:

```
// scripts/seed-products.ts
import { Database } from "tina4-nodejs/orm";

async function seedProducts() {
    const db = Database.getConnection();

    const products = [
        { name: "Keyboard", price: 79.99 },
        { name: "Mouse", price: 29.99 },
        { name: "Monitor", price: 399.99 }
    ];

    for (const p of products) {
        await db.execute(
            "INSERT INTO products (name, price) VALUES (:name, :price)",
            p
        );
    }

    console.log(`Seeded ${products.length} products`);
}

seedProducts();
```

Run it:

```
npx tsx scripts/seed-products.ts
```

---

16. Exercise: Scaffold a Feature in 5 Commands

Scaffold a complete "Customer" feature from scratch using only CLI commands.

Requirements

Starting from an existing Tina4 Node.js project, run 5 commands to create:

- A Customer ORM model with name, email, phone, and company fields
- A CRUD route file with all five REST endpoints
- A migration that creates the customers table
- Run the migration to create the table
- Run the doctor to verify everything

Expected Commands

```
# 1. Generate the model with route and migration
tina4 generate model Customer --with-route --with-migration

# 2. Edit the model to add fields (manual step)
# Add fields to src/orm/Customer.ts

# 3. Edit the migration to add columns (manual step)
# Add columns to the migration file

# 4. Run the migration
tina4 migrate

# 5. Run the doctor to verify everything is set up
tina4 doctor
```

Solution

Command 1: Generate everything:

```
tina4 generate model Customer --with-route --with-migration
```

Command 2: Edit `src/orm/Customer.ts`:

```
import { BaseModel } from "tina4-nodejs/orm";

export class Customer extends BaseModel {
    static tableName = "customers";
    static primaryKey = "id";

    id!: number;
    name!: string;
    email!: string;
    phone!: string;
    company!: string;
    createdAt!: string;
}
```

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:7148/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.

---

17. 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 Node.js.

Cause: A `composer.json` file exists in the directory from a previous project.

Fix: Use explicit language selection: `tina4 init nodejs my-project`. Or delete the conflicting language file before running `init`.

3. Generated Files Overwrite Existing Code

Problem: Running `tina4 generate route products` overwrites your custom route file.

Cause: The generate command creates files at fixed paths. If the file exists, it is overwritten.

Fix: The CLI warns you before overwriting. Check if the file exists first. If you need to regenerate, rename the existing file: `mv src/routes/products.ts src/routes/products_backup.ts`.

4. Migration Order Issues

Problem: A migration fails because it references a table that does not exist yet.

Cause: Migration files run in alphabetical (timestamp) order. If migration B depends on a table created by migration A, but A has a later timestamp, B runs first and fails.

Fix: Use consistent timestamps. The `tina4 generate migration` command uses the current timestamp. Generating migrations in order ensures correct execution order. If you need to fix ordering, rename the migration files to adjust their timestamps.

5. tina4 serve Uses Wrong Port

Problem: `tina4 serve` starts on port 7148 but you need port 8080.

Cause: The default port is 7148 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 TypeScript 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 Fails

Fix: Check TypeScript errors with `npx tsc --noEmit`.

9. Port Conflict

Fix: Use `tina4 serve --port 8080` or set `TINA4_PORT` in `.env`.

---

18. 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.

---

19. Test Port (Dual-Port Development)

When `TINA4_DEBUG=true`, Tina4 automatically starts a second HTTP server on `port + 1000`:

- **Main port** (e.g. 7148) — hot-reload enabled, for AI dev tools
- **Test port** (e.g. 8148) — 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, and how to wire everything together. It hallucinates half of it.

With Tina4, the AI reads one file -- `CLAUDE.md` -- and knows everything. Every method signature. Every import path. Every convention. It generates correct, runnable code on the first try because there is only one way to do things in Tina4.

This is not accidental. Tina4 was built from the ground up as the best framework for AI-assisted development.

The Zero-Dependency Advantage

When an AI generates code for Express, it might suggest `express-session` or `cookie-session` or `connect-redis`. All exist. All work differently. The AI guesses.

With Tina4, there is one cache. One queue. One ORM. One template engine. No ambiguity. No alternatives. No "it depends on which package you installed."

```
// There is only one way to cache in Tina4
const data = await cacheGet("products");

// There is only one way to queue in Tina4
const queue = new Queue({ topic: "emails" });
queue.push({ to: "user@test.com" });

// There is only one way to send email in Tina4
const mail = new Messenger();
await mail.send({ to: "user@test.com", subject: "Welcome", body: "<h1>Welcome!</h1>", html: true
```

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:

- 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 `async (req, res) =>`, ORM uses `toDict()`)
- 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.

The Convention Advantage

AI thrives on convention. Every Tina4 project follows the same structure. The AI never asks "where should I put this?"

```
src/
  routes/hello.ts       → AI knows: this is a route file
  orm/Product.ts        → AI knows: this is an ORM model
  templates/home.twig   → AI knows: this is a template
```

Tell the AI "add a User model" and it creates `src/orm/User.ts` with the correct class definition, the correct imports, and the correct property types. Every time.

Real-World Vibe Coding Session

Here is an actual conversation with Claude Code in a Tina4 Node.js project:

You: "Add a blog with posts and comments. Posts belong to users. Comments belong to posts. I want a REST API and a template page that lists posts."

Claude Code generates:

- `src/orm/Post.ts` -- Model with belongsTo User, hasMany Comments
- `src/orm/Comment.ts` -- Model with belongsTo Post

- `src/routes/api/posts.ts` -- Full CRUD API
- `src/routes/api/comments.ts` -- Nested comments API
- `src/routes/blog.ts` -- Template route for the blog page
- `src/templates/blog.twig` -- Page with tina4css cards for each post
- `migrations/20260322_create_posts_and_comments.sql` -- Database schema

All correct. All runnable. First try.

You: "Add pagination to the posts list"

Claude adds `req.query.page` handling and proper offset calculation -- because CLAUDE.md told it exactly how Tina4 pagination works.

You: "Cache the post list for 5 minutes"

Claude adds `cacheGet("posts_page_" + page)` with `cacheSet("posts_page_" + page, data, 300)` -- because it knows the cache API.

You: "Send an email when a new comment is posted"

Claude adds `const mail = new Messenger(); await mail.send(...)` -- because it knows Messenger reads from `.env`.

No research. No Stack Overflow. No "which package should I use?" Describe what you want. The AI builds it.

TypeScript-Specific Advantages

TypeScript makes AI coding even more reliable:

```
// The AI knows the exact types
import { Router, Auth, Queue, Messenger, cacheGet, cacheSet } from "tina4-nodejs";
import { BaseModel, Database } from "tina4-nodejs/orm";

// Type declarations guide the AI
export class Product extends BaseModel {
    static tableName = "products";
    static primaryKey = "id";
    static hasMany = [{ model: "Review", foreignKey: "product_id" }];

    id!: number;
    name!: string;
    price: number = 0;
    inStock: boolean = true;
}

// The AI generates type-safe route handlers
Router.get("/api/products/{id:int}", async (req, res) => {
    const product = new Product();
    await product.load(req.params.id);
    return product.id ? res.json(product.toDict()) : res.status(404).json({ error: "Not found" }
});
```

TypeScript's type system is a guardrail. It prevents the AI from generating code that fails at runtime. Property types, method signatures, and return types steer the AI toward correct code.

Supported AI Tools

Tina4 auto-detects and installs context for eight tools. The exact list lives in `@tina4/core` (the 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 AI has full context of your Tina4 project. Ask it anything:

- "Why is my /api/users route returning 500?"
- "How do I add WebSocket support?"

- "Write a migration to add an avatar column to users"

The "Ask about this error" button on the error overlay sends the full stack trace and source code to the AI for instant debugging.

Tips for Effective Vibe Coding with Tina4

- **Be specific about what you want** -- "Add a product API with search by name and price range" works better than "add products"
- **Let the AI use tina4 generate** -- "Run tina4 generate model Product then add price and category fields"
- **Reference the .env** -- "Configure the queue to use RabbitMQ" -- the AI knows to update .env
- **Ask for tests** -- "Write tests for the Product API" -- the AI uses Tina4's inline testing
- **Review, don't rewrite** -- The AI's first attempt is usually 90% correct. Tweak the details.

Exercise

Open your Tina4 Node.js project in Claude Code (or your preferred AI tool) and try these prompts:

- "Add a Contact model with name, email, phone, and message fields"
- "Create a contact form page at /contact with a Twig template using tina4css"
- "Add an API endpoint that accepts the form submission and saves it to the database"
- "Send an email notification when a new contact is submitted"
- "Add rate limiting to the contact form -- max 3 submissions per minute"

Each prompt should generate correct, runnable TypeScript code on the first try. If it does not, check that CLAUDE.md is in your project root.

Gotchas

- **No CLAUDE.md?** Run `tina4 init` or `tina4 doctor` to generate it
- **AI suggests wrong import paths?** The CLAUDE.md might be outdated -- regenerate it
- **AI hallucinates a package?** Remind it: "Tina4 has zero dependencies, use only built-in features"
- **AI creates files in wrong location?** Tell it: "Follow the src/routes, src/orm, src/templates convention"
- **AI code doesn't run?** Check that routes return `res.json()` not just `console.log()` -- Tina4 convention matters

- **AI uses Express patterns?** Remind it: "This is Tina4, not Express. Import from tina4-nodejs."
- **AI forgets await?** All Tina4 database and ORM methods are async and need await

---

The Philosophy

Tina4 was not retrofitted for AI coding tools. It was designed for them from the ground up.

Convention over configuration means the AI knows where things go. Zero dependencies means the AI never chooses between packages. A single CLAUDE.md file means the AI has complete framework knowledge. Identical APIs across 4 languages means that knowledge transfers instantly.

You describe the intent. The AI writes the code. You review and refine. Tina4 is the ground the partnership stands on.

The Intelligent Native Application 4ramework. Built for AI. Built for you.

Environment Variables

> **■■ *BREAKING CHANGE — Tina4 v3.12.0***
>
> *Every framework env var now requires the `TINA4_` prefix. The legacy un-prefixed names (`DATABASE_URL`, `SECRET`, `SMTP_HOST`, `HOST_NAME`, etc.) no longer work. Setting them at startup makes the framework refuse to boot with a list of renames.*
>
> *Run `tina4 env --migrate` to rewrite your existing `.env` automatically, or rename manually using the table below. The runtime guard prints the same mapping if it detects legacy names.*
>
> ***Conventional names stay un-prefixed:** `PORT`, `HOST`, `NODE_ENV`, `RACK_ENV`, `RUBY_ENV`, `ENVIRONMENT`. These are runtime/PaaS conventions, not framework config.*

Tina4 Node.js 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 Node.js 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`	`0.0.0.0`	Framework-prefixed bind address. Takes precedence over `HOST` when both are set.
`PORT`	`7148`	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:7148`	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_SUPPRESS`	`false`	Suppresses the boot banner on startup. Useful for clean CI logs and embedded usage.
`TINA4_ENV_FILE`	`.env`	Path to the dotenv file loaded at startup. Relative paths resolve from the project root.
`TINA4_HEALTH_PATH`	`/__health`	Path for the built-in health endpoint. Legacy `/health` alias is also registered when this points at `/__health`.
`TINA4_NO_BROWSER`	`false`	Stops `tina4 serve` from opening your browser on every restart.

`TINA4_NO_RELOAD`	`false`	Disables the dev hot-reload signal from the Rust CLI. Use when you want a stable server for debugging.
`TINA4_PRODUCTION`	`false`	Forces production-mode startup (cluster mode, debug overlays disabled). Set automatically by `tina4 serve --production`.
`TINA4_ALLOW_LEGACY_ENV`	`false`	Bypass the v3.12 env-prefix guard. CI / migration scripts only — never enable in production.

Secrets and Authentication

Variable	Default	Description
`TINA4_SECRET`	*(empty)*	JWT signing secret. Must be long, random, and unique per environment. **Never commit.**
`TINA4_JWT_ALGORITHM`	`HS256`	JWT signing algorithm. Supports `HS256`, `HS384`, `HS512`.
`TINA4_TOKEN_LIMIT`	`60`	JWT token lifetime in minutes.
`TINA4_API_KEY`	*(empty)*	Static API key used as a fallback to JWT.

Database

Variable	Default	Description
`TINA4_DATABASE_URL`	*(required)*	Connection URL. Scheme selects the driver: `sqlite`, `postgres`, `mysql`.
`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_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_DB_POOL`	`0`	Connection-pool size override. Applied to the database adapter's `pool` config when not set explicitly in code. `0` leaves the adapter default in place.
`TINA4_ORM_PLURAL_TABLE_NAMES`	`true`	When `true`, the ORM pluralises class names into table names (`User` → `users`).

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`	Comma-separated list of allowed methods on preflight responses.
`TINA4_CORS_HEADERS`	`Content-Type,Authorization`	Comma-separated list of allowed request headers on preflight responses.
`TINA4_CORS_MAX_AGE`	`86400`	Preflight cache lifetime in seconds.
`TINA4_CORS_CREDENTIALS`	`true`	Sets `Access-Control-Allow-Credentials`.

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.
`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_NAME`	`tina4_session`	Session cookie name.
`TINA4_SESSION_TTL`	`1800`	Session expiry in seconds (30 minutes).
`TINA4_SESSION_SAMESITE`	`Lax`	SameSite cookie attribute. Options: `Strict`, `Lax`, `None`.
`TINA4_SESSION_HTTPONLY`	`true`	Emit the `HttpOnly` flag on the session cookie.
`TINA4_SESSION_SECURE`	`false`	Emit the `Secure` flag on the session cookie. Enable behind HTTPS.
`TINA4_SESSION_PATH`	`data/sessions`	Filesystem path for the file backend.

Redis/Valkey session backend

Variable	Default	Description
`TINA4_SESSION_REDIS_URL`	*(none)*	Full `redis://` URL. Overrides host/port when set.
`TINA4_SESSION_REDIS_HOST`	`127.0.0.1`	Redis host.
`TINA4_SESSION_REDIS_PORT`	`6379`	Redis port.
`TINA4_SESSION_REDIS_PASSWORD`	*(none)*	Redis auth password.
`TINA4_SESSION_REDIS_DB`	`0`	Redis database number.
`TINA4_SESSION_REDIS_PREFIX`	`tina4:session:`	Key prefix for stored sessions.
`TINA4_SESSION_VALKEY_HOST`	`127.0.0.1`	Valkey host.
`TINA4_SESSION_VALKEY_PORT`	`6379`	Valkey port.
`TINA4_SESSION_VALKEY_PASSWORD`	*(none)*	Valkey auth password.
`TINA4_SESSION_VALKEY_DB`	`0`	Valkey database number.
`TINA4_SESSION_VALKEY_PREFIX`	`tina4:session:`	Key prefix for stored sessions.

MongoDB session backend

Variable	Default	Description
`TINA4_SESSION_MONGO_URI`	*(none)*	Full MongoDB connection string. Overrides host/port when set.
`TINA4_SESSION_MONGO_HOST`	`127.0.0.1`	MongoDB host.
`TINA4_SESSION_MONGO_PORT`	`27017`	MongoDB port.
`TINA4_SESSION_MONGO_USERNAME`	*(none)*	MongoDB username.
`TINA4_SESSION_MONGO_PASSWORD`	*(none)*	MongoDB password.
`TINA4_SESSION_MONGO_DB`	`tina4_sessions`	MongoDB database name.
`TINA4_SESSION_MONGO_COLLECTION`	`sessions`	MongoDB collection name.

Cache

Variable	Default	Description
`TINA4_CACHE_BACKEND`	`memory`	Response cache backend. Options: `memory`, `file`, `redis`.
`TINA4_CACHE_DIR`	`data/cache`	Cache directory for the file backend.
`TINA4_CACHE_TTL`	`60`	Default cache TTL in seconds.
`TINA4_CACHE_MAX_ENTRIES`	`1000`	Maximum cache entries.
`TINA4_CACHE_URL`	`redis://localhost:6379`	Connection URL for remote cache backends.

Queues

Variable	Default	Description
`TINA4_QUEUE_BACKEND`	`file`	Queue backend. Options: `file`, `rabbitmq`, `kafka`, `mongodb`.
`TINA4_QUEUE_PATH`	`data/queue`	Filesystem path for the file backend.
`TINA4_QUEUE_URL`	*(none)*	Connection URL for remote backends (rabbitmq/kafka).

Kafka queue backend

Variable	Default	Description
`TINA4_KAFKA_BROKERS`	`localhost:9092`	Comma-separated broker list.
`TINA4_KAFKA_GROUP_ID`	`tina4_consumer_group`	Kafka consumer group ID.

RabbitMQ queue backend

Variable	Default	Description
`TINA4_RABBITMQ_HOST`	`localhost`	RabbitMQ host.
`TINA4_RABBITMQ_PORT`	`5672`	RabbitMQ port.
`TINA4_RABBITMQ_USERNAME`	`guest`	RabbitMQ username.
`TINA4_RABBITMQ_PASSWORD`	`guest`	RabbitMQ password.
`TINA4_RABBITMQ_VHOST`	`/`	RabbitMQ virtual host.

MongoDB queue backend

Variable	Default	Description
`TINA4_MONGO_URI`	*(none)*	Full MongoDB connection string. Overrides host/port when set.
`TINA4_MONGO_HOST`	`localhost`	MongoDB host.
`TINA4_MONGO_PORT`	`27017`	MongoDB port.
`TINA4_MONGO_USERNAME`	*(none)*	MongoDB username.
`TINA4_MONGO_PASSWORD`	*(none)*	MongoDB password.
`TINA4_MONGO_DB`	`tina4`	MongoDB database name.
`TINA4_MONGO_COLLECTION`	`tina4_queue`	MongoDB collection name for jobs.

WebSocket

Variable	Default	Description
`TINA4_WS_PORT`	`8080`	Default WebSocket server port when one isn't passed to the constructor.
`TINA4_WS_BACKPLANE`	*(none)*	Backplane type for multi-instance broadcasts. Options: `redis`, `nats`.
`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`	*(none)*	SMTP server hostname.
`TINA4_MAIL_PORT`	`587`	SMTP server port.
`TINA4_MAIL_USERNAME`	*(none)*	SMTP authentication username.
`TINA4_MAIL_PASSWORD`	*(none)*	SMTP authentication password.
`TINA4_MAIL_FROM`	*(none)*	Default sender email address.
`TINA4_MAIL_FROM_NAME`	*(none)*	Default sender display name.
`TINA4_MAIL_ENCRYPTION`	`tls`	Connection encryption. Options: `tls`, `ssl`, `none`.
`TINA4_MAIL_IMAP_HOST`	*(none)*	IMAP server for inbound mail.
`TINA4_MAIL_IMAP_PORT`	`993`	IMAP server port.
`TINA4_MAIL_IMAP_ENCRYPTION`	`tls`	IMAP transport security. Options: `tls`, `starttls`, `none`. Node also accepts `ssl` as an alias for `tls` for back-compat.
`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`, `TINA4_MAIL_IMAP_USERNAME`, `TINA4_MAIL_IMAP_PASSWORD` *are accepted as legacy aliases. New projects should use the* `TINA4_MAIL_*` *names.*

Logging

Logs default to stdout. Set `TINA4_LOG_OUTPUT=file` plus `TINA4_LOG_FILE=app.log` to write to disk; the framework rotates at `TINA4_LOG_ROTATE_SIZE` bytes and keeps `TINA4_LOG_ROTATE_KEEP` backups using stdlib `fs` calls.

Variable	Default	Description

`TINA4_LOG_LEVEL`	`DEBUG`	Minimum log level written to console and files. Options: `DEBUG`, `INFO`, `WARNING`, `ERROR`, `ALL`.
`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.

File logging

Variable	Default	Description
`TINA4_LOG_FILE`	*(empty = stdout only)*	Explicit log file path. Absolute paths are used as-is; relative paths resolve against `TINA4_LOG_DIR`.
`TINA4_LOG_DIR`	`logs`	Directory for log files when `TINA4_LOG_FILE` is relative or unset.
`TINA4_LOG_FORMAT`	`text`	File log format. Options: `text`, `json`.
`TINA4_LOG_OUTPUT`	`stdout`	Where to write logs. Options: `stdout`, `file`, `both`.
`TINA4_LOG_CRITICAL`	`false`	Enable the `CRITICAL` level shortcut. Off by default to keep the log surface compact.
`TINA4_LOG_ROTATE_SIZE`	`10485760`	Rotate the file when it reaches this many bytes (10 MB). `0` disables rotation. Node uses an atomic `fs.renameSync` shift on each write.
`TINA4_LOG_ROTATE_KEEP`	`5`	Number of rotated historical files to retain.

Localisation

Variable	Default	Description
`TINA4_LOCALE`	`en`	Default locale for the `I18n` module.
`TINA4_LOCALE_DIR`	`src/locales`	Directory containing locale JSON files.

Uploads

Variable	Default	Description
`TINA4_MAX_UPLOAD_SIZE`	`10485760`	Maximum multipart upload size in bytes (10 MB).

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 empty polls.

Routing

Variable	Default	Description
`TINA4_TEMPLATE_ROUTING`	`on`	Auto-route any request without a matching handler to a same-named Frond template. Set to `off`, `false`, `0`, `no`, or `disabled` for explicit-only routing.
`TINA4_TRAILING_SLASH_REDIRECT`	`false`	When `true`, requests with a trailing slash are 301-redirected to the canonical no-trailing-slash form.

Templates

Variable	Default	Description
`TINA4_TEMPLATE_CACHE_TTL`	`0`	Frond template cache TTL in seconds. `0` (the default) caches permanently for the process lifetime.

GraphQL

Variable	Default	Description
`TINA4_GRAPHQL_AUTO_SCHEMA`	`true`	Auto-build a schema from registered ORM models when the dev server starts. Set `false` to require an explicit schema.
`TINA4_GRAPHQL_ENDPOINT`	`/graphql`	Path the GraphQL handler is mounted on.

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 `true` when `TINA4_DEBUG=true`, `false` otherwise.
`TINA4_MCP_PORT`	*(main port + 2000)*	Port the MCP server listens on. Defaults to the framework HTTP port plus 2000.

Swagger / OpenAPI

Variable	Default	Description
`TINA4_SWAGGER_TITLE`	`Tina4 API`	OpenAPI spec title.
`TINA4_SWAGGER_DESCRIPTION`	`Auto-generated API documentation`	OpenAPI spec description.
`TINA4_SWAGGER_VERSION`	`0.0.1`	OpenAPI spec version.
`TINA4_SWAGGER_CONTACT_EMAIL`	*(empty)*	Contact email written into the generated OpenAPI `info.contact` block.
`TINA4_SWAGGER_LICENSE`	*(empty)*	License name written into the generated OpenAPI `info.license` block.
`TINA4_SWAGGER_ENABLED`	*(inherits `TINA4_DEBUG`)*	Mount the Swagger UI. Defaults to enabled when `TINA4_DEBUG=true`. Force `true`/`false` to override.

Minimal `.env` for Development

```
TINA4_DEBUG=true
TINA4_LOG_LEVEL=DEBUG
TINA4_NO_BROWSER=true
```

Debug mode lights up the Swagger UI, the dev dashboard, detailed error pages, and live reload. Keeping the browser flag on stops a new tab opening every time you save a file.

Minimal `.env` for Production

```
TINA4_SECRET=your-long-random-secret-here
TINA4_DATABASE_URL=postgresql://user:password@db-host:5432/myapp
TINA4_CORS_ORIGINS=https://myapp.com,https://www.myapp.com
TINA4_HSTS=31536000
TINA4_MAIL_HOST=smtp.example.com
TINA4_MAIL_PORT=587
TINA4_MAIL_USERNAME=noreply@myapp.com
TINA4_MAIL_PASSWORD=your-smtp-password
TINA4_MAIL_FROM=noreply@myapp.com
```

No `TINA4_DEBUG`. It defaults to `false`, which is what you want in production. Set a real secret, a real database, locked-down CORS origins, HSTS, and SMTP credentials if you send email. Everything else has a production-appropriate default.

Deployment

1. From Development to Production

The app works on `localhost:7148`. 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, build process, 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 npm 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=7148

# Database
TINA4_DATABASE_URL=sqlite:///data/app.db

# Security
CORS_ORIGINS=https://yourdomain.com
TINA4_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 TINA4_SECRET=your-secret -e TINA4_DATABASE_URL=sqlite:///data/app.db my-app

# Fly.io: set secrets
fly secrets set TINA4_SECRET=your-secret

# Railway: use the dashboard or CLI
railway variables set TINA4_SECRET=your-secret
```

---

3. Building for Production

```
tina4 build
```

```
Building for production...
  Compiled 47 TypeScript files
  Output: dist/
  Size: 245KB
```

Run in production:

```
node dist/app.js
```

---

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

Create `Dockerfile`:

```
FROM node:20-alpine

WORKDIR /app

# Copy dependency files first (for better layer caching)
COPY package*.json ./
RUN npm ci --only=production

# Copy application code
COPY dist/ ./dist/
COPY src/templates/ ./src/templates/
COPY src/public/ ./src/public/
COPY migrations/ ./migrations/

# Create directories for data and logs
RUN mkdir -p data logs

ENV TINA4_DEBUG=false
```

```
ENV TINA4_PORT=7148

EXPOSE 7148

# Health check
HEALTHCHECK --interval=30s --timeout=5s --retries=3 \
  CMD wget -qO- http://localhost:7148/health || exit 1

CMD ["node", "dist/app.js"]
```

.dockerignore

Create `.dockerignore`:

```
.git
.env
node_modules
.claude
data/*.db
logs/*.log
src/
!src/templates/
!src/public/
!migrations/
```

Building and Running

```
# Build the image
tina4 build
docker build -t my-tina4-app .

# Run the container
docker run -d \
  --name my-app \
  -p 7148:7148 \
  -e TINA4_SECRET=your-production-secret \
  -e TINA4_DATABASE_URL=sqlite:///data/app.db \
  -v $(pwd)/data:/app/data \
  -v $(pwd)/logs:/app/logs \
  my-tina4-app
```

Docker Compose

For a complete setup with supporting services, use Docker Compose.

Create `docker-compose.yml`:

```
services:
  app:
    build: .
    ports:
      - "7148:7148"
    environment:
      - TINA4_DEBUG=false
      - TINA4_LOG_LEVEL=WARNING
      - TINA4_SECRET=${TINA4_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", "wget", "-qO-", "http://localhost:7148/health"]
      interval: 30s
      timeout: 5s
      retries: 3

  redis:
    image: redis:7-alpine
    ports:
      - "6379:6379"
    volumes:
      - redis-data:/data
    restart: unless-stopped

volumes:
  app-data:
  app-logs:
  redis-data:
```

```
# Start everything
docker compose up -d

# View logs
docker compose logs -f app

# Stop everything
docker compose down
```

---

5. Node.js Cluster for Production

For multi-core utilization, Tina4 supports Node.js cluster mode:

Or set workers to `auto` to match CPU cores:

This spawns multiple worker processes. Each handles requests independently. A worker crashes. The cluster master respawns it. No downtime.

How Many Workers?

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.

CPU Cores	Workers	Use Case
1	2	Small VPS, hobbyist
2	4	Small production app
4	8	Medium production app
8	16	High-traffic app

---

6. 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:

```
import { Router } from "tina4-nodejs";
import { Database } from "tina4-nodejs/orm";

Router.get("/health", async (req, res) => {
    let dbOk = false;

    try {
        const db = Database.getConnection();
        await db.fetchOne("SELECT 1");
        dbOk = true;
    } catch (e) {
        // Database is down
    }

    const status = dbOk ? "ok" : "degraded";
    const statusCode = dbOk ? 200 : 503;

    return res.status(statusCode).json({
        status,
        version: "1.0.0",
        database: dbOk ? "connected" : "disconnected",
        timestamp: new Date().toISOString()
    });
});
```

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)

Broken File Check

Tina4 watches for a `.broken` file in production. When the file exists, the health check returns `503`. A signal to the load balancer: stop sending traffic.

```
touch .broken          # Health check returns 503
# Deploy new code...
rm .broken             # Health check returns 200
```

---

7. Graceful Shutdown

Deploy a new version. The old process must stop. Graceful shutdown finishes active requests before terminating.

Tina4 handles this when it receives a `SIGTERM` signal (the standard shutdown signal from Docker, Kubernetes, and systemd):

- Stop accepting new connections
- Wait for active requests to complete (up to 30 seconds)
- Close database connections
- Flush logs
- Exit

Configuring Shutdown Timeout

```
TINA4_SHUTDOWN_TIMEOUT=30
```

If active requests do not finish within the timeout, the server terminates them. Set this based on your longest expected request time. For most applications, 30 seconds is generous.

Docker Stop Grace Period

Docker sends `SIGTERM`, waits for the grace period, then sends `SIGKILL`. Match the Docker grace period to your shutdown timeout:

```
services:
  app:
    stop_grace_period: 30s
```

---

8. Log Rotation

In production, logs grow without limit unless rotated. Tina4 writes logs to `logs/app.log` and `logs/error.log`.

Using logrotate (Linux)

Create `/etc/logrotate.d/tina4`:

```
/app/logs/*.log {
    daily
    rotate 14
    compress
    delaycompress
    missingok
    notifempty
    create 0640 www-data www-data
    postrotate
        kill -USR1 $(cat /app/tina4.pid) 2>/dev/null || true
    endscript
}
```

This rotates logs daily, keeps 14 days of history, and compresses old logs. The `USR1` signal tells Tina4 to reopen its log files after rotation.

Docker Logging

Docker captures stdout/stderr automatically. Configure log rotation in the Docker daemon or compose file:

```
services:
  app:
    logging:
      driver: json-file
      options:
        max-size: "10m"
        max-file: "5"
```

This keeps at most 50MB of logs (5 files of 10MB each).

---

9. Nginx Reverse Proxy

In production, Tina4 runs behind Nginx. Nginx handles:

- SSL/TLS termination (HTTPS)
- Static file serving (faster than Node.js)
- Request buffering
- Rate limiting
- WebSocket proxying

Create `/etc/nginx/sites-available/my-app`:

```
server {
    listen 80;
    server_name yourdomain.com;
    return 301 https://$host$request_uri;
}

server {
    listen 443 ssl;
    server_name yourdomain.com;

    ssl_certificate /etc/letsencrypt/live/yourdomain.com/fullchain.pem;
    ssl_certificate_key /etc/letsencrypt/live/yourdomain.com/privkey.pem;

    # Static files (served by Nginx, faster than Node.js)
    location /css/ {
        alias /app/src/public/css/;
        expires 30d;
        add_header Cache-Control "public, immutable";
    }

    location /js/ {
        alias /app/src/public/js/;
        expires 30d;
```

```
        add_header Cache-Control "public, immutable";
    }

    location /images/ {
        alias /app/src/public/images/;
        expires 30d;
        add_header Cache-Control "public, immutable";
    }

    # WebSocket upgrade
    location /ws/ {
        proxy_pass http://127.0.0.1:7148;
        proxy_http_version 1.1;
        proxy_set_header Upgrade $http_upgrade;
        proxy_set_header Connection "upgrade";
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_read_timeout 86400;
    }

    # Application
    location / {
        proxy_pass http://127.0.0.1:7148;
        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;
    }
}
```

---

10. 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
      - TINA4_SECRET=${TINA4_SECRET}

volumes:
  letsencrypt:
```

Traefik detects your app container through Docker labels, provisions a certificate from Let's Encrypt, and handles all HTTPS traffic. No Nginx configuration needed.

Certificate Monitoring

Certificates expire. Even with auto-renewal, things go wrong -- DNS changes, firewall rules blocking port 80 for ACME challenges, or a crashed renewal service. Set up monitoring:

```
# Check certificate expiry manually
echo | openssl s_client -connect yourdomain.com:443 2>/dev/null | openssl x509 -noout -dates

# Verify Certbot timer is active
sudo systemctl list-timers | grep certbot
```

Use an external monitoring service (Uptime Robot, Better Uptime) that checks certificate expiry and alerts you 14 days before it expires.

---

11. Process Management with systemd

Create `/etc/systemd/system/tina4-app.service`:

```
[Unit]
Description=Tina4 Node.js Application
After=network.target

[Service]
Type=simple
User=www-data
WorkingDirectory=/var/www/my-app
ExecStart=/usr/bin/node /var/www/my-app/dist/app.js
Restart=always
RestartSec=5
Environment=NODE_ENV=production
EnvironmentFile=/var/www/my-app/.env.production

[Install]
WantedBy=multi-user.target
```

```
sudo systemctl enable tina4-app
sudo systemctl start tina4-app
sudo systemctl status tina4-app
```

---

12. Queue Workers in Production

```
[Unit]
Description=Tina4 Queue Worker
After=network.target

[Service]
Type=simple
User=www-data
WorkingDirectory=/var/www/my-app
ExecStart=/usr/local/bin/tina4 queue work
Restart=always
RestartSec=5
EnvironmentFile=/var/www/my-app/.env.production

[Install]
WantedBy=multi-user.target
```

---

13. Zero-Downtime Deployment

```
#!/bin/bash
set -e

cd /var/www/my-app

# Signal load balancer to stop sending traffic
touch .broken
sleep 5

# Pull latest code
git pull origin main

# Install dependencies and build
npm ci --only=production
tina4 build

# Run migrations
tina4 migrate

# Restart the service
sudo systemctl restart tina4-app

# Wait for health check
sleep 2
curl -f http://localhost:7148/health

# Re-enable traffic
rm .broken

echo "Deployment complete"
```

---

14. Scaling

A single server handles many applications. When traffic outgrows one server, you scale.

Vertical Scaling

Use cluster mode with more workers:

Load Balancing with Nginx

When you run multiple Tina4 instances, Nginx distributes traffic across them:

```
upstream tina4_backend {
    server 127.0.0.1:7148;
    server 127.0.0.1:7248;
    server 127.0.0.1:7348;
    server 127.0.0.1:7448;
}

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. Node's framework default is `7148`; use any free ports for the additional instances. Set `TINA4_OVERRIDE_CLIENT=true` so Node can run the build directly without going through `tina4 serve`:

```
TINA4_OVERRIDE_CLIENT=true TINA4_PORT=7148 node dist/app.js &
TINA4_OVERRIDE_CLIENT=true TINA4_PORT=7248 node dist/app.js &
TINA4_OVERRIDE_CLIENT=true TINA4_PORT=7348 node dist/app.js &
TINA4_OVERRIDE_CLIENT=true TINA4_PORT=7448 node dist/app.js &
```

Nginx distributes requests in round-robin order by default. If a backend goes down, Nginx routes traffic to the remaining instances.

Docker Scaling

With Docker Compose, scale horizontally with a single command:

```
docker compose up -d --scale app=4
```

Horizontal Scaling

Run multiple instances behind a load balancer. Use Redis for shared sessions, cache, and queues:

```
TINA4_SESSION_HANDLER=redis
TINA4_CACHE_BACKEND=redis
TINA4_QUEUE_BACKEND=rabbitmq
# Or use MongoDB for queues:
# TINA4_QUEUE_BACKEND=mongodb
# TINA4_MONGO_URI=mongodb://user:pass@mongo.internal:27017/tina4
```

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.

---

15. Monitoring

Your app runs in production. You need to know when it breaks, slows down, or runs out of resources.

Log Aggregation

Switch to JSON-formatted logs for production. Structured logs feed into aggregation services:

```
TINA4_LOG_FORMAT=json
```

Example JSON log entry:

```
{
  "timestamp": "2026-03-22T14:30:00.123Z",
  "level": "WARNING",
  "message": "Rate limit exceeded",
  "request_id": "req-abc123",
  "ip": "203.0.113.42",
  "path": "/api/products",
  "method": "GET"
}
```

Services that ingest JSON logs:

Service	Strengths
Grafana Loki	Open source, pairs with Grafana dashboards
Elastic Stack (ELK)	Full-text search across logs
Datadog	Managed service, correlates logs with metrics
AWS CloudWatch	Native for AWS deployments

Docker makes log aggregation straightforward. Configure the logging driver to send container output to your chosen service:

```
services:
  app:
    logging:
      driver: "fluentd"
      options:
        fluentd-address: "localhost:24224"
        tag: "tina4.app"
```

Uptime Monitoring

Point an external monitoring service at your health endpoint:

```
https://yourdomain.com/health
```

Services like Uptime Robot, Pingdom, or Better Uptime ping this endpoint every 30-60 seconds. When it stops responding or returns a non-200 status, you receive an alert via email, SMS, or Slack.

The health endpoint from section 6 serves double duty: container orchestrators use it for restart decisions, and uptime monitors use it for alerting.

Application Performance Monitoring (APM)

Uptime monitoring tells you the app is running. APM tells you how well it performs. APM agents track:

- Request latency (average, p95, p99)
- Database query performance (slow queries, connection pool usage)
- Error rates (which endpoints fail, how often)
- Memory and CPU usage over time

Since Tina4 Node.js runs on standard Node.js, any Node.js APM agent works:

- **Datadog APM**: `npm install dd-trace` and `require('dd-trace').init()` at the top of your app
- **New Relic**: `npm install newrelic` and `require('newrelic')` at the top of your app
- **Elastic APM**: `npm install elastic-apm-node` and configure in your app startup

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.

---

16. Exercise: Docker Deploy

Deploy a Tina4 Node.js application using Docker.

Requirements

- Create a `Dockerfile` that:
- Uses `node:20-alpine` as the base image
- Installs production dependencies with `npm ci`
- Copies the built application code
- Exposes port 7148
- Includes a health check
- Runs the app with `node dist/app.js`
- Create a `docker-compose.yml` that:
- Builds and runs the app
- Starts a Redis container for caching
- Mounts volumes for data persistence
- Sets environment variables for production
- Create a `/health` endpoint that checks database connectivity
- Build, run, and verify:

```
# Build
```

```
tina4 build
docker compose build

# Start
docker compose up -d

# Test health
curl http://localhost:7148/health

# Test the app
curl http://localhost:7148/api/products

# View logs
docker compose logs -f app

# Stop
docker compose down
```

Solution

The Dockerfile and docker-compose.yml are shown in section 4. The health check route is shown in section 6. Combine them in your project, then:

```
tina4 build
docker compose up -d --build
```

```
[+] Building 12.3s
[+] Running 2/2
  Container redis    Started
  Container my-app   Started
```

```
curl http://localhost:7148/health
```

```
{
  "status": "ok",
  "version": "1.0.0",
  "database": "connected",
  "timestamp": "2026-03-22T14:30:00.000Z"
}
```

The app runs in production mode with Redis caching, persistent data volumes, automatic restarts, and health monitoring.

---

17. Gotchas

1. .env Not Loaded in Docker

Problem: Environment variables from `.env` are not available in the container.

Cause: Docker does not read `.env` files automatically. The `.env` file belongs in `.dockerignore` (never ship secrets in the image).

Fix: Pass environment variables via `docker run -e`, the `environment` section in `docker-compose.yml`, or an `env_file` directive. For secrets, use Docker secrets or your platform's secret management.

2. SQLite Database Lost on Container Restart

Problem: All data disappears when the container restarts.

Cause: The SQLite database file sits inside the container. When the container is recreated, the file is gone.

Fix: Mount a volume for the data directory: `-v $(pwd)/data:/app/data`. In Docker Compose, use a named volume: `volumes: [app-data:/app/data]`.

3. WebSocket Connections Drop Behind Nginx

Problem: WebSocket connections fail or drop immediately behind Nginx.

Cause: Nginx does not proxy WebSocket by default. It treats the upgrade request as a regular HTTP request.

Fix: Add WebSocket proxy headers in your Nginx config:

```
proxy_http_version 1.1;
proxy_set_header Upgrade $http_upgrade;
proxy_set_header Connection "upgrade";
proxy_read_timeout 86400;
```

4. Static Files Slow

Problem: CSS, JS, and images load slowly in production.

Cause: Node.js serves static files. Every static file request goes through the Node.js process.

Fix: Serve static files from Nginx (see the Nginx config in section 9). Nginx serves static files from memory-mapped files, which is orders of magnitude faster than Node.js.

5. Memory Usage Grows Over Time

Problem: The container's memory usage climbs until it crashes with OOM (out of memory).

Cause: Memory leaks in the application -- unclosed database connections, growing caches without TTL, accumulating request data in global variables.

Fix: Set TTLs on all cache entries. Close database connections properly. Avoid storing request data in module-level variables. Use `docker stats` to monitor memory usage. Set memory limits in Docker Compose: `deploy: {resources: {limits: {memory: 512M}}}`.

6. Container Starts Before Database Is Ready

Problem: The app crashes on startup because the database is not ready.

Cause: Docker Compose starts services in parallel. The app container starts before the database container finishes initializing.

Fix: For SQLite, this is not an issue (the file is created automatically). For PostgreSQL or MySQL, use a startup script that waits for the database, or use Docker Compose healthcheck on the database service with `depends_on: {db: {condition: service_healthy}}`.

7. SSL Certificate Not Renewing

Problem: Your HTTPS certificate expires and the site goes down.

Cause: The auto-renewal process (Certbot or Traefik) failed. Common reasons: DNS changes, firewall blocking port 80 for ACME challenges, or the renewal service crashed.

Fix: Monitor certificate expiry with an external service. Check renewal logs and verify the renewal timer is active:

```
sudo certbot renew --dry-run
sudo systemctl list-timers | grep certbot
```

8. 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.

9. Debug Mode in Production

Fix: Set `TINA4_DEBUG=false`.

10. .env in Version Control

Fix: Add to `.gitignore`.

11. Missing Migrations

Fix: Always run `tina4 migrate` during deployment.

12. Port Conflicts

Fix: Ensure only one process listens on port 7148.

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.

We are building **TaskFlow** -- a task management system with:

- User registration and JWT authentication
- Task creation, assignment, and tracking
- A dashboard with real-time updates
- Email notifications when tasks are assigned
- Caching for dashboard performance
- A full test suite
- Docker deployment

Not a toy project. A production-ready application. Every major Tina4 feature in one codebase.

---

2. Planning the App

Models

Model	Table	Fields
User	users	id, name, email, passwordHash, role, createdAt
Task	tasks	id, title, description, status, priority, createdBy, assignedTo, dueDate, completedAt, createdAt, updatedAt

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
cd taskflow
npm install
```

Update `.env`:

```
TINA4_DEBUG=true
TINA4_SECRET=taskflow-dev-secret-change-in-production
TINA4_TOKEN_LIMIT=86400
```

Create Migrations

Create `migrations/20260322000100_create_users_table.sql`:

```
-- UP
CREATE TABLE users (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    name TEXT NOT NULL,
    email TEXT NOT NULL UNIQUE,
```

```
    password_hash TEXT NOT NULL,
    role TEXT NOT NULL DEFAULT 'user',
    created_at TEXT DEFAULT CURRENT_TIMESTAMP
);

CREATE INDEX idx_users_email ON users(email);

-- DOWN
DROP TABLE IF EXISTS users;
```

Create `migrations/20260322000200_create_tasks_table.sql`:

```
-- UP
CREATE TABLE tasks (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    title TEXT NOT NULL,
    description TEXT 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] 20260322000100_create_users_table.sql
  [UP] 20260322000200_create_tasks_table.sql

  2 migrations applied
```

Verify the database:

```
curl http://localhost:7148/health
```

```
{"status": "ok", "database": "connected"}
```

---

4. Step 2: Define Models

Create `src/orm/User.ts`:

```
import { BaseModel, Auth } from "tina4-nodejs";

export class User extends BaseModel {
```

```
    static tableName = "users";
    static primaryKey = "id";
    static hasMany = [
        { model: "Task", foreignKey: "created_by", as: "createdTasks" },
        { model: "Task", foreignKey: "assigned_to", as: "assignedTasks" }
    ];

    id!: number;
    name!: string;
    email!: string;
    passwordHash!: string;
    role: string = "user";
    createdAt!: string;

    async setPassword(password: string): Promise<void> {
        this.passwordHash = await Auth.hashPassword(password);
    }

    async verifyPassword(password: string): Promise<boolean> {
        if (!this.passwordHash) return false;
        return Auth.checkPassword(password, this.passwordHash);
    }

    safeDict(): Record<string, any> {
        const data = this.toDict();
        delete data.passwordHash;
        delete data.password_hash;
        return data;
    }
}
```

The `setPassword` method hashes the plain-text password before storage. `verifyPassword` compares a candidate password against the stored hash. `safeDict` strips the hash from any response -- never expose password data to the client.

Create `src/orm/Task.ts`:

```
import { BaseModel } from "tina4-nodejs/orm";

export class Task extends BaseModel {
    static tableName = "tasks";
    static primaryKey = "id";
    static belongsTo = [
        { model: "User", foreignKey: "created_by", as: "creator" },
        { model: "User", foreignKey: "assigned_to", as: "assignee" }
    ];

    static STATUSES = ["todo", "in_progress", "done", "cancelled"];
    static PRIORITIES = ["low", "medium", "high", "urgent"];

    id!: number;
    title!: string;
    description: string = "";
    status: string = "todo";
    priority: string = "medium";
    createdBy!: number;
    assignedTo: number | null = null;
    dueDate: string | null = null;
    completedAt: string | null = null;
```

```
    createdAt!: string;
    updatedAt!: string;
}
```

The static `STATUSES` and `PRIORITIES` arrays serve as validation lists. Any route that accepts a status or priority value checks it against these arrays before writing to the database.

---

5. Step 3: Auth Middleware

Create `src/routes/middleware.ts`:

```
import { Auth } from "tina4-nodejs";

export function authMiddleware(req, res, next) {
    const authHeader = req.headers["authorization"] ?? "";

    if (!authHeader || !authHeader.startsWith("Bearer ")) {
        res({ error: "Authorization required" }, 401);
        return;
    }

    const token = authHeader.substring(7);

    const secret = process.env.TINA4_SECRET || "tina4-default-secret";
    const payload = Auth.validToken(token, secret);
    if (payload === null) {
        res({ error: "Invalid or expired token" }, 401);
        return;
    }

    req.user = payload;
    req.userId = payload.user_id;
    next();
}
```

Verify the middleware blocks unauthenticated requests:

```
curl http://localhost:7148/api/tasks
```

```
{"error": "Authorization required"}
```

The middleware stands guard. No token, no access.

---

6. Step 4: Auth Routes

Create `src/routes/auth.ts`:

```
import { Router, Auth } from "tina4-nodejs";
import { Database } from "tina4-nodejs/orm";
import { authMiddleware } from "./middleware";

/**
 * Register a new user
 * @noauth
 * @tags Auth
```

```
 * @body {"name": "string", "email": "string", "password": "string"}
 * @response 201 {"message": "string", "user": {"id": "int", "name": "string", "email": "string"
 */
Router.post("/api/auth/register", async (req, res) => {
    const { name, email, password } = req.body;

    if (!name || !email || !password) {
        return res.status(400).json({ error: "Name, email, and password are required" });
    }

    if (password.length < 8) {
        return res.status(400).json({ error: "Password must be at least 8 characters" });
    }

    const db = Database.getConnection();
    const existing = await db.fetchOne("SELECT id FROM users WHERE email = :email", { email });

    if (existing) {
        return res.status(409).json({ error: "Email already registered" });
    }

    const hash = await Auth.hashPassword(password);
    await db.execute(
        "INSERT INTO users (name, email, password_hash) VALUES (:name, :email, :hash)",
        { name, email, hash }
    );

    const user = await db.fetchOne("SELECT id, name, email, role FROM users WHERE id = last_inse

    return res.status(201).json({ message: "Registration successful", user });
});

/**
 * Login and get JWT token
 * @noauth
 * @tags Auth
 * @body {"email": "string", "password": "string"}
 * @response 200 {"token": "string", "user": {"id": "int", "name": "string", "email": "string"}}
 */
Router.post("/api/auth/login", async (req, res) => {
    const { email, password } = req.body;

    if (!email || !password) {
        return res.status(400).json({ error: "Email and password are required" });
    }

    const db = Database.getConnection();
    const user = await db.fetchOne(
        "SELECT id, name, email, password_hash, role FROM users WHERE email = :email",
        { email }
    );

    if (!user || !(await Auth.checkPassword(password, user.password_hash))) {
        return res.status(401).json({ error: "Invalid email or password" });
    }

    const secret = process.env.TINA4_SECRET || "tina4-default-secret";
    const token = Auth.getToken({
        user_id: user.id,
```

```
        email: user.email,
        name: user.name,
        role: user.role
    }, secret);

    return res.json({
        message: "Login successful",
        token,
        user: { id: user.id, name: user.name, email: user.email, role: user.role }
    });
});

/**
 * Get current user profile
 * @tags Auth
 * @response 200 {"id": "int", "name": "string", "email": "string", "role": "string", "created_a
 */
Router.get("/api/profile", async (req, res) => {
    const db = Database.getConnection();
    const user = await db.fetchOne(
        "SELECT id, name, email, role, created_at FROM users WHERE id = :id",
        { id: req.user.user_id }
    );
    return res.json(user);
}, [authMiddleware]);
```

Test Registration and Login

```
# Start the server
tina4 serve
```

```
# Register a user
curl -X POST http://localhost:7148/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:7148/api/auth/login \
  -H "Content-Type: application/json" \
  -d '{"email": "alice@example.com", "password": "securePass123"}'
```

```
{
  "message": "Login successful",
  "token": "eyJhbGciOiJIUzI1NiIs...",
  "user": {"id": 1, "name": "Alice Johnson", "email": "alice@example.com", "role": "user"}
}
```

Authentication works. Save the token for the next steps.

```
export TOKEN="eyJhbGciOiJIUzI1NiIs..."
```

---

7. Step 5: Task Routes

Create `src/routes/tasks.ts`:

```
import { Router, Queue } from "tina4-nodejs";
import { Database } from "tina4-nodejs/orm";
import { cacheDelete } from "tina4-nodejs";
import { authMiddleware } from "./middleware";
import { Task } from "../orm/Task";
import { pushTaskUpdate } from "./ws-tasks";

/**
 * List tasks with filters
 * @tags Tasks
 * @query string status Filter by status (todo, in_progress, done)
 * @query string priority Filter by priority (low, medium, high)
 * @query string assigned Filter by assigned user ID
 * @query int page Page number
 * @query int limit Results per page
 */
Router.get("/api/tasks", async (req, res) => {
    const db = Database.getConnection();
    const userId = req.user.user_id;
    const page = parseInt(req.query.page ?? "1", 10);
    const limit = parseInt(req.query.limit ?? "20", 10);
    const offset = (page - 1) * limit;

    const conditions = ["(created_by = :userId OR assigned_to = :userId)"];
    const params: Record<string, any> = { userId };

    if (req.query.status) {
        conditions.push("status = :status");
        params.status = req.query.status;
    }

    if (req.query.priority) {
        conditions.push("priority = :priority");
        params.priority = req.query.priority;
    }

    if (req.query.assigned) {
        conditions.push("assigned_to = :assignedTo");
        params.assignedTo = parseInt(req.query.assigned, 10);
    }

    const sql = `SELECT t.*,
        creator.name as creator_name,
        assignee.name as assignee_name
        FROM tasks t
        LEFT JOIN users creator ON t.created_by = creator.id
        LEFT JOIN users assignee ON t.assigned_to = assignee.id
        WHERE ${conditions.join(" AND ")}
        ORDER BY
            CASE t.priority WHEN 'high' THEN 1 WHEN 'medium' THEN 2 WHEN 'low' THEN 3 END,
            t.created_at DESC
        LIMIT :limit OFFSET :offset`;

    params.limit = limit;
    params.offset = offset;
```

```
    const tasks = await db.fetch(sql, params);

    return res.json({ tasks, page, limit, count: tasks.length });
}, [authMiddleware]);

/**
 * Get a single task
 * @tags Tasks
 * @param int id Task ID
 */
Router.get("/api/tasks/{id:int}", async (req, res) => {
    const db = Database.getConnection();
    const task = await db.fetchOne(
        `SELECT t.*, creator.name as creator_name, assignee.name as assignee_name
         FROM tasks t
         LEFT JOIN users creator ON t.created_by = creator.id
         LEFT JOIN users assignee ON t.assigned_to = assignee.id
         WHERE t.id = :id`,
        { id: req.params.id }
    );

    if (!task) {
        return res.status(404).json({ error: "Task not found" });
    }

    return res.json(task);
}, [authMiddleware]);

/**
 * Create a task
 * @tags Tasks
 * @body {"title": "string", "description": "string", "priority": "string", "assigned_to": "int"
 */
Router.post("/api/tasks", async (req, res) => {
    const db = Database.getConnection();
    const { title, description, priority, assigned_to, due_date } = req.body;

    if (!title) {
        return res.status(400).json({ error: "Title is required" });
    }

    if (priority && !Task.PRIORITIES.includes(priority)) {
        return res.status(400).json({ error: `Invalid priority. Must be one of: ${Task.PRIORITIE
    }

    await db.execute(
        `INSERT INTO tasks (title, description, status, priority, created_by, assigned_to, due_d
         VALUES (:title, :description, 'todo', :priority, :createdBy, :assignedTo, :dueDate)`,
        {
            title,
            description: description ?? "",
            priority: priority ?? "medium",
            createdBy: req.user.user_id,
            assignedTo: assigned_to ?? null,
            dueDate: due_date ?? null
        }
    );

    const task = await db.fetchOne("SELECT * FROM tasks WHERE id = last_insert_rowid()");
```

```
    // Invalidate dashboard cache
    cacheDelete("dashboard:stats");

    // Queue notification if assigned to someone
    if (assigned_to) {
        const notifyQueue = new Queue({ topic: "task-notifications" });
        notifyQueue.push({
            type: "assigned",
            task_id: task.id,
            task_title: title,
            assigned_to,
            assigned_by: req.user.name
        });
    }

    // Push real-time update
    pushTaskUpdate("created", task);

    return res.status(201).json(task);
}, [authMiddleware]);

/**
 * Update a task
 * @tags Tasks
 */
Router.put("/api/tasks/{id:int}", async (req, res) => {
    const db = Database.getConnection();
    const id = req.params.id;

    const existing = await db.fetchOne("SELECT * FROM tasks WHERE id = :id", { id });
    if (!existing) {
        return res.status(404).json({ error: "Task not found" });
    }

    const { title, description, status, priority, assigned_to, due_date } = req.body;

    if (status && !Task.STATUSES.includes(status)) {
        return res.status(400).json({ error: `Invalid status. Must be one of: ${Task.STATUSES.jo
    }

    if (priority && !Task.PRIORITIES.includes(priority)) {
        return res.status(400).json({ error: `Invalid priority. Must be one of: ${Task.PRIORITIE
    }

    const completedAt = status === "done" && existing.status !== "done"
        ? new Date().toISOString()
        : existing.completed_at;

    // Detect reassignment for notification
    const oldAssignee = existing.assigned_to;
    const newAssignee = assigned_to ?? existing.assigned_to;

    await db.execute(
        `UPDATE tasks SET title = :title, description = :description, status = :status,
         priority = :priority, assigned_to = :assignedTo, due_date = :dueDate,
         completed_at = :completedAt, updated_at = CURRENT_TIMESTAMP
         WHERE id = :id`,
        {
            title: title ?? existing.title,
```

```
            description: description ?? existing.description,
            status: status ?? existing.status,
            priority: priority ?? existing.priority,
            assignedTo: newAssignee,
            dueDate: due_date ?? existing.due_date,
            completedAt,
            id
        }
    );

    const task = await db.fetchOne("SELECT * FROM tasks WHERE id = :id", { id });

    // Invalidate dashboard cache
    cacheDelete("dashboard:stats");

    // Notify new assignee if reassigned
    if (newAssignee && newAssignee !== oldAssignee) {
        const notifyQueue = new Queue({ topic: "task-notifications" });
        notifyQueue.push({
            type: "assigned",
            task_id: task.id,
            task_title: task.title,
            assigned_to: newAssignee,
            assigned_by: req.user.name
        });
    }

    // Push real-time update
    pushTaskUpdate("updated", task);

    return res.json(task);
}, [authMiddleware]);

/**
 * Delete a task
 * @tags Tasks
 */
Router.delete("/api/tasks/{id:int}", async (req, res) => {
    const db = Database.getConnection();
    const id = req.params.id;

    const existing = await db.fetchOne("SELECT * FROM tasks WHERE id = :id", { id });
    if (!existing) {
        return res.status(404).json({ error: "Task not found" });
    }

    await db.execute("DELETE FROM tasks WHERE id = :id", { id });

    // Invalidate dashboard cache
    cacheDelete("dashboard:stats");

    // Push real-time update
    pushTaskUpdate("deleted", { id });

    return res.status(204).json(null);
}, [authMiddleware]);
```

Test the Task API

```
# Set your token from the login step
TOKEN="eyJhbGciOiJIUzI1NiIs..."

# Create a task
curl -X POST http://localhost:7148/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:7148/api/tasks \
  -H "Content-Type: application/json" \
  -H "Authorization: Bearer $TOKEN" \
  -d '{"title": "Build API endpoints", "priority": "high"}'

curl -X POST http://localhost:7148/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:7148/api/tasks \
  -H "Authorization: Bearer $TOKEN"

# Update a task status
curl -X PUT http://localhost:7148/api/tasks/1 \
  -H "Content-Type: application/json" \
  -H "Authorization: Bearer $TOKEN" \
  -d '{"status": "done"}'

# Delete a task
curl -X DELETE http://localhost:7148/api/tasks/3 \
  -H "Authorization: Bearer $TOKEN"
```

Every endpoint responds. Create. Read. Update. Delete. The CRUD cycle works.

---

8. Step 6: Dashboard Stats with Caching

Create `src/routes/dashboard.ts`:

```
import { Router, cacheGet, cacheSet } from "tina4-nodejs";
import { Database } from "tina4-nodejs/orm";
import { authMiddleware } from "./middleware";

/**
 * Dashboard statistics
 * @tags Dashboard
```

```
 * @response 200 {"total_tasks": "int", "by_status": "object", "overdue_tasks": "int", "total_us
 */
Router.get("/api/dashboard/stats", async (req, res) => {
    const userId = req.user.user_id;
    const cacheKey = `dashboard:${userId}`;

    const cached = await cacheGet(cacheKey);
    if (cached) {
        return res.json({ ...cached, source: "cache" });
    }

    const db = Database.getConnection();

    const total = await db.fetchOne(
        "SELECT COUNT(*) as count FROM tasks WHERE created_by = :userId OR assigned_to = :userId
        { userId }
    );

    const todo = await db.fetchOne(
        "SELECT COUNT(*) as count FROM tasks WHERE (created_by = :userId OR assigned_to = :userI
        { userId }
    );

    const inProgress = await db.fetchOne(
        "SELECT COUNT(*) as count FROM tasks WHERE (created_by = :userId OR assigned_to = :userI
        { userId }
    );

    const done = await db.fetchOne(
        "SELECT COUNT(*) as count FROM tasks WHERE (created_by = :userId OR assigned_to = :userI
        { userId }
    );

    const overdue = await db.fetchOne(
        "SELECT COUNT(*) as count FROM tasks WHERE (created_by = :userId OR assigned_to = :userI
        { userId }
    );

    const users = await db.fetchOne("SELECT COUNT(*) as count FROM users");

    const recentTasks = await db.fetch(
        `SELECT t.*, u.name as assignee_name FROM tasks t
         LEFT JOIN users u ON t.assigned_to = u.id
         WHERE t.created_by = :userId OR t.assigned_to = :userId
         ORDER BY t.created_at DESC LIMIT 10`,
        { userId }
    );

    const stats = {
        total_tasks: total.count,
        todo: todo.count,
        in_progress: inProgress.count,
        done: done.count,
        overdue_tasks: overdue.count,
        total_users: users.count,
        recent_tasks: recentTasks
    };

    await cacheSet(cacheKey, stats, 30);
```

```
    return res.json({ ...stats, source: "database" });
}, [authMiddleware]);

/**
 * Dashboard HTML page
 * @noauth
 * @tags Dashboard
 */
Router.get("/admin", async (req, res) => {
    return res.html("dashboard.html", {});
});
```

Verify the stats endpoint:

```
curl http://localhost:7148/api/dashboard/stats \
  -H "Authorization: Bearer $TOKEN"
```

```
{
  "total_tasks": 2,
  "todo": 1,
  "in_progress": 0,
  "done": 1,
  "overdue_tasks": 0,
  "total_users": 1,
  "recent_tasks": [...],
  "source": "database"
}
```

Hit it again. The second response comes from cache:

```
curl http://localhost:7148/api/dashboard/stats \
  -H "Authorization: Bearer $TOKEN"
```

```
{
  "total_tasks": 2,
  "todo": 1,
  "done": 1,
  "source": "cache"
}
```

The cache holds for 30 seconds. Creating, updating, or deleting a task invalidates it.

---

9. Step 7: WebSocket for Real-Time Updates

Create `src/routes/ws-tasks.ts`:

```
import { Router } from "tina4-nodejs";

Router.websocket("/ws/tasks", async (connection, event, data) => {
    if (event === "open") {
        connection.send(JSON.stringify({
            type: "connected",
            message: "Listening for task updates"
        }));
    }

    if (event === "message") {
        const message = JSON.parse(data);
```

```
        if (message.type === "ping") {
            connection.send(JSON.stringify({ type: "pong" }));
        }
    }
});

/**
 * Push a task update to all connected WebSocket clients.
 */
export function pushTaskUpdate(action: string, task: any): void {
    Router.pushToWebSocket("/ws/tasks", JSON.stringify({
        type: "task_update",
        action,
        task
    }));
}
```

Any user creates, updates, or deletes a task. All connected dashboard users see the change.

---

10. Step 8: Email Notifications via Queue

Create `src/routes/queue-consumers.ts`:

```
import { Queue, Messenger } from "tina4-nodejs";
import { Database } from "tina4-nodejs/orm";

const notifyQueue = new Queue({ topic: "task-notifications" });

notifyQueue.process(async (job) => {
    const { type, task_title, task_id, assigned_to, assigned_by } = job.payload as any;

    const db = Database.getConnection();
    const user = await db.fetchOne("SELECT name, email FROM users WHERE id = :id", { id: assigne

    if (!user) {
        job.complete();
        return;
    }

    console.log(`[Notification] Task "${task_title}" assigned to ${user.name} by ${assigned_by}`

    const mailer = new Messenger();
    await mailer.send({
        to: user.email,
        subject: `New Task Assigned: ${task_title}`,
        body: `<h2>New Task Assigned</h2>
               <p>Hi ${user.name},</p>
               <p><strong>${assigned_by}</strong> assigned you a new task:</p>
               <div style="border:1px solid #ddd; padding:16px; margin:16px 0;">
                   <h3>${task_title}</h3>
                   <p><a href="http://localhost:7148/admin#task-${task_id}">View Task</a></p>
               </div>`,
        html: true
    });

    job.complete();
});
```

Create the email template at `src/templates/emails/task-assigned.html`:

```
<!DOCTYPE html>
<html>
<head><meta charset="UTF-8"></head>
<body>
    <div class="container">
        <h2>New Task Assigned</h2>
        <p>Hi {{ assignee_name }},</p>
        <p><strong>{{ creator_name }}</strong> assigned you a new task:</p>

        <div class="card">
            <div class="card-body">
                <h3>{{ task_title }}</h3>
                <p>{{ task_description }}</p>
                <table class="table">
                    <tr><td><strong>Priority:</strong></td><td>{{ task_priority }}</td></tr>
                    <tr><td><strong>Due:</strong></td><td>{{ task_due_date }}</td></tr>
                </table>
            </div>
        </div>

        <p><a href="{{ task_url }}">View Task</a></p>
    </div>
</body>
</html>
```

The queue consumer runs in the background. The task route pushes a job. The consumer picks it up, looks up the assignee, and sends the email. No blocking. No delay in the API response.

---

11. Step 9: Dashboard Template

Create `src/templates/dashboard.html`:

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

{% block title %}TaskFlow Dashboard{% endblock %}

{% block content %}
<h1>Dashboard</h1>

<div id="stats" class="row mb-4">
    <div class="col-md-2"><div class="card"><div class="card-body">
        <h6 class="text-muted">Total</h6><h2 id="stat-total">--</h2>
    </div></div></div>
    <div class="col-md-2"><div class="card"><div class="card-body">
        <h6 class="text-muted">To Do</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 class="col-md-2"><div class="card"><div class="card-body">
        <h6 class="text-muted">Users</h6><h2 id="stat-users">--</h2>
    </div></div></div>
</div>

<div class="row">
    <div class="col-md-8">
        <div class="card">
            <div class="card-header d-flex justify-content-between">
                <span>Recent Tasks</span>
                <button class="btn btn-sm btn-primary" data-toggle="modal" data-target="#newTask
                    New Task
                </button>
            </div>
            <div class="card-body">
                <table class="table table-striped" id="task-table">
                    <thead>
                        <tr><th>Title</th><th>Assignee</th><th>Priority</th><th>Status</th></tr>
                    </thead>
                    <tbody id="task-list"></tbody>
                </table>
            </div>
        </div>
    </div>
    <div class="col-md-4">
        <div class="card">
            <div class="card-header">Live Updates</div>
            <div class="card-body" id="live-updates">
                <p class="text-muted">Connecting...</p>
            </div>
        </div>
    </div>
</div>

<script src="/js/frond.min.js"></script>
<script>
    var token = localStorage.getItem("token");
    if (token) frond.setToken(token);

    // Load dashboard stats
    frond.get("/api/dashboard/stats", function (data) {
        document.getElementById("stat-total").textContent = data.total_tasks;
        document.getElementById("stat-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_tasks;
        document.getElementById("stat-users").textContent = data.total_users;

        var tbody = document.getElementById("task-list");
        tbody.innerHTML = "";
        (data.recent_tasks || []).forEach(function (task) {
            var tr = document.createElement("tr");
            var badgeColor = {todo: "secondary", in_progress: "warning",
                              done: "success", cancelled: "danger"};
            tr.innerHTML = "<td>" + task.title + "</td>" +
                "<td>" + (task.assignee_name || "Unassigned") + "</td>" +
                "<td>" + task.priority + "</td>" +
                "<td><span class='badge bg-" + (badgeColor[task.status] || "secondary") +
```

```
                "'>" + task.status + "</span></td>";
            tbody.appendChild(tr);
        });
    });

    // WebSocket for live updates
    var ws = frond.ws("/ws/tasks");
    var updates = document.getElementById("live-updates");

    ws.on("open", function () {
        updates.innerHTML = "<p class='text-success'>Connected - listening for updates</p>";
    });

    ws.on("message", function (raw) {
        var msg = JSON.parse(raw);
        if (msg.type === "task_update") {
            var div = document.createElement("div");
            div.innerHTML = "<strong>" + msg.action + ":</strong> " + msg.task.title;
            updates.prepend(div);

            // Refresh stats
            frond.get("/api/dashboard/stats", function () {});
        }
    });
</script>
{% endblock %}
```

Verify the dashboard:

```
curl http://localhost:7148/admin
```

The server returns the rendered HTML. Open `http://localhost:7148/admin` in your browser to see the full dashboard with stats, task list, and live update panel.

---

12. Step 10: Tests

Create `tests/TaskFlowTest.ts`:

```
import { tests, assertEqual, assertTrue, assertNotNull, runAllTests, Auth } from "tina4-nodejs";
import { TestClient } from "tina4-nodejs/test";

const client = new TestClient();

function randomEmail(): string {
    const hex = Math.random().toString(16).substring(2, 10);
    return `test-${hex}@example.com`;
}

async function registerAndLogin(): Promise<string> {
    const email = randomEmail();
    await client.post("/api/auth/register", {
        name: "Test User",
        email,
        password: "TestPassword123!"
    });
    const loginResp = await client.post("/api/auth/login", {
        email,
```

```
        password: "TestPassword123!"
    });
    return loginResp.body.token;
}

// Test registration
const testRegister = tests(
    assertEqual([201])
)(async function testRegister(): Promise<number> {
    const resp = await client.post("/api/auth/register", {
        name: "Alice",
        email: randomEmail(),
        password: "SecurePass123!"
    });
    return resp.statusCode;
});

// Test login returns a token
const testLogin = tests(
    assertTrue([true])
)(async function testLogin(): Promise<boolean> {
    const email = randomEmail();
    await client.post("/api/auth/register", {
        name: "Bob",
        email,
        password: "SecurePass123!"
    });
    const resp = await client.post("/api/auth/login", {
        email,
        password: "SecurePass123!"
    });
    return resp.statusCode === 200 && resp.body.token !== undefined;
});

// Test task creation
const testCreateTask = tests(
    assertTrue([true])
)(async function testCreateTask(): Promise<boolean> {
    const token = await registerAndLogin();
    const resp = await client.post("/api/tasks", {
        title: "Test Task",
        description: "A test task",
        priority: "high"
    }, { headers: { Authorization: `Bearer ${token}` } });
    return resp.statusCode === 201 && resp.body.title === "Test Task" && resp.body.priority ===
});

// Test listing tasks
const testListTasks = tests(
    assertTrue([true])
)(async function testListTasks(): Promise<boolean> {
    const token = await registerAndLogin();
    await client.post("/api/tasks", { title: "List Task 1" }, { headers: { Authorization: `Beare
    await client.post("/api/tasks", { title: "List Task 2" }, { headers: { Authorization: `Beare
    const resp = await client.get("/api/tasks", { headers: { Authorization: `Bearer ${token}` }
    return resp.statusCode === 200 && resp.body.tasks.length >= 2;
});

// Test updating task status
```

```
const testUpdateTaskStatus = tests(
    assertTrue([true])
)(async function testUpdateTaskStatus(): Promise<boolean> {
    const token = await registerAndLogin();
    const createResp = await client.post("/api/tasks", { title: "Status Task" },
        { headers: { Authorization: `Bearer ${token}` } });
    const taskId = createResp.body.id;
    const resp = await client.put(`/api/tasks/${taskId}`, { status: "done" },
        { headers: { Authorization: `Bearer ${token}` } });
    return resp.statusCode === 200 && resp.body.status === "done" && resp.body.completed_at !==
});

// Test deleting a task
const testDeleteTask = tests(
    assertEqual([204])
)(async function testDeleteTask(): Promise<number> {
    const token = await registerAndLogin();
    const createResp = await client.post("/api/tasks", { title: "Delete Me" },
        { headers: { Authorization: `Bearer ${token}` } });
    const taskId = createResp.body.id;
    const resp = await client.delete(`/api/tasks/${taskId}`,
        { headers: { Authorization: `Bearer ${token}` } });
    return resp.statusCode;
});

// Test dashboard stats
const testDashboardStats = tests(
    assertTrue([true])
)(async function testDashboardStats(): Promise<boolean> {
    const token = await registerAndLogin();
    await client.post("/api/tasks", { title: "Stats Task" },
        { headers: { Authorization: `Bearer ${token}` } });
    const resp = await client.get("/api/dashboard/stats",
        { headers: { Authorization: `Bearer ${token}` } });
    return resp.statusCode === 200 && resp.body.total_tasks >= 1;
});

// Test unauthorized access
const testUnauthorizedAccess = tests(
    assertEqual([401])
)(async function testUnauthorizedAccess(): Promise<number> {
    const resp = await client.get("/api/tasks");
    return resp.statusCode;
});

// Test that token creation and validation round-trips correctly
const testTokenRoundTrip = tests(
    assertTrue([{ userId: 1, role: "admin" }]),
)(function testTokenRoundTrip(payload: Record<string, unknown>): boolean {
    const secret = process.env.TINA4_SECRET || "test-secret";
    const token = Auth.getToken(payload, secret);
    const decoded = Auth.validToken(token, secret);
    return decoded !== null && decoded.userId === payload.userId;
});

// Test that password hashing and checking works
const testPasswordHash = tests(
    assertTrue(["securePass123"]),
)(function testPasswordHash(password: string): boolean {
```

```
    const hash = Auth.hashPassword(password);
    return Auth.checkPassword(password, hash);
});

runAllTests();
```

Run the tests:

```
npm test
```

```
Running tests...

  TaskFlowTest
    [PASS] testRegister
    [PASS] testLogin
    [PASS] testCreateTask
    [PASS] testListTasks
    [PASS] testUpdateTaskStatus
    [PASS] testDeleteTask
    [PASS] testDashboardStats
    [PASS] testUnauthorizedAccess
    [PASS] testTokenRoundTrip
    [PASS] testPasswordHash

  10 tests, 10 passed, 0 failed (1.12s)
```

All green. The application works.

---

13. Step 11: Docker Deployment

Create `Dockerfile`:

```
FROM node:20-alpine
WORKDIR /app
COPY package*.json ./
RUN npm ci --only=production
COPY dist/ ./dist/
COPY src/templates/ ./src/templates/
COPY src/public/ ./src/public/
COPY migrations/ ./migrations/
RUN mkdir -p data logs
ENV TINA4_DEBUG=false
EXPOSE 7148

HEALTHCHECK --interval=30s --timeout=5s --retries=3 \
  CMD curl -f http://localhost:7148/health || exit 1

CMD ["node", "dist/app.js"]
```

Create `docker-compose.yml`:

```
services:
  app:
    build: .
    ports:
      - "7148:7148"
    environment:
      - TINA4_DEBUG=false
```

```
      - TINA4_SECRET=${JWT_SECRET:-change-me-in-production}
      - TINA4_CACHE_BACKEND=redis
      - TINA4_CACHE_URL=redis
    volumes:
      - taskflow-data:/app/data
      - taskflow-logs:/app/logs
    depends_on:
      - redis
    restart: unless-stopped

  redis:
    image: redis:7-alpine
    restart: unless-stopped

volumes:
  taskflow-data:
  taskflow-logs:
```

Build and deploy:

```
tina4 build
docker compose up -d --build
```

Verify:

```
curl http://localhost:7148/health
```

```
{"status": "ok", "version": "1.0.0", "database": "connected"}
```

TaskFlow runs in production. Authenticated APIs. Real-time WebSocket updates. Email notifications. Cached dashboard stats. Tested. Dockerized.

---

14. The Complete Project Structure

```
taskflow/
■■■ .env
■■■ .env.example
■■■ .gitignore
■■■ package.json
■■■ package-lock.json
■■■ tsconfig.json
■■■ app.ts
■■■ Dockerfile
■■■ docker-compose.yml
■■■ src/
■   ■■■ routes/
■   ■   ■■■ auth.ts               # Registration, login
■   ■   ■■■ tasks.ts              # Task CRUD
■   ■   ■■■ dashboard.ts          # Dashboard stats + page
■   ■   ■■■ middleware.ts         # Auth middleware
■   ■   ■■■ queue-consumers.ts    # Email notification consumer
■   ■   ■■■ ws-tasks.ts           # WebSocket event handlers
■   ■■■ orm/
■   ■   ■■■ User.ts               # User model with auth methods
■   ■   ■■■ Task.ts               # Task model with relationships
■   ■■■ migrations/
■   ■   ■■■ 20260322000100_create_users_table.sql
■   ■   ■■■ 20260322000200_create_tasks_table.sql
■   ■■■ templates/
■   ■   ■■■ base.html             # Base layout
■   ■   ■■■ dashboard.html        # Dashboard page
■   ■   ■■■ emails/
■   ■   ■   ■■■ task-assigned.html # Assignment notification
■   ■   ■■■ errors/
■   ■       ■■■ 404.html
■   ■       ■■■ 500.html
■   ■■■ public/
■   ■   ■■■ css/
■   ■   ■   ■■■ tina4.css
■   ■   ■■■ js/
■   ■       ■■■ tina4.min.js
■   ■       ■■■ frond.min.js
■   ■■■ locales/
■       ■■■ en.json
■■■ data/
■   ■■■ app.db
■■■ dist/                        # Production build output
■■■ logs/
■■■ secrets/
■■■ tests/
    ■■■ TaskFlowTest.ts
```

Every file has a purpose. Every directory follows the convention. A new developer looks at this structure and knows where to find things.

---

15. What You Built

This chapter used every major concept from the book:

Feature	Chapter
Route registration (`Router.get`, `Router.post`, `Router.put`, `Router.delete`)	Chapter 2
Request/response handling	Chapter 3
Twig templates	Chapter 4
Database queries	Chapter 5
ORM models (User, Task)	Chapter 6
JWT authentication	Chapter 8
Auth middleware	Chapter 10
Email notifications (Messenger)	Chapter 16
Cache (dashboard stats)	Chapter 11
Frontend (tina4css dashboard)	Chapter 17
WebSocket (live updates)	Chapter 23
Testing (full test suite)	Chapter 18
Docker deployment	Chapter 33

---

16. What to Build Next

TaskFlow is a solid foundation. Here are ideas for extending it.

Features:

- **Task comments** -- Add a Comment model with a `taskId` foreign key. Display comments on the task detail page.
- **File attachments** -- Let users upload files to tasks. Store them in `data/uploads/` and serve them via a route.
- **Team management** -- Add a Team model. Users belong to teams. Tasks are scoped to teams.
- **Task labels/tags** -- Many-to-many relationship between tasks and labels for categorization.
- **Due date reminders** -- Use the queue system to schedule reminder emails 24 hours before a task's due date.
- **Activity log** -- Record every change to a task (who changed what, when) for audit trails.
- **Search** -- Full-text search across task titles and descriptions.
- **Calendar view** -- Render tasks on a calendar based on their due dates.

- **Mobile API** -- The API already works for mobile apps. Add push notification support via Firebase Cloud Messaging.

Technical improvements:

- **Rate limiting per user** -- Replace the global rate limiter with per-user limits.
- **Database upgrade** -- Switch from SQLite to PostgreSQL for better concurrency.
- **CI/CD pipeline** -- Add GitHub Actions to run tests on every push.
- **API documentation** -- Generate OpenAPI/Swagger docs from your route annotations.
- **Internationalization** -- Add `src/locales/` files for multiple languages.

---

17. Closing Thoughts -- The Tina4 Philosophy

You built a complete application. User auth. CRUD. Real-time updates. Email. Caching. Tests. Deployment. Your project has one dependency: `tina4-nodejs`.

No separate ORM package. No template engine package. No authentication library. No WebSocket server. No caching library. No testing framework. No CLI tool. No CSS framework. No JavaScript helpers. All built in.

One framework. Zero extra dependencies. Everything you need.

The same patterns work in PHP, Python, and Ruby. 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:** `autoMap` now defaults to `true` — ORM models automatically map between camelCase properties and snake_case DB columns. Set `static autoMap = false;` on your model to restore the old behaviour.

• **feat:** `toDict(include, case)` parameter — pass `'snake'` as second arg to get snake_case keys matching DB columns, or `'camel'` (default) for camelCase.

• **feat:** Frond `replace` filter now accepts object args — `{{ v|replace({"T": " ", "-": "/"}) }}` for multiple substitutions in one call.

• **tests:** 13 new parity tests covering `toDict(case)`, `autoMap` default, `replace` filter (object + positional), and `ServiceRunner` registration. 268 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 `findOutsideQuotes()` and `splitOutsideQuotes()` — expressions like `arr[i % 2]` no longer treated as top-level arithmetic.

• **fix:** Frond subscript expression evaluation — bracket content uses `evalExpr()` instead of direct context lookup, enabling `arr[loop.index0 % 2]`.

• **fix:** Frond slice with variable bounds — `items[start:end]` evaluates bounds through `evalExpr()`.

• **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. 2,831 tests passing (268 Frond).

v3.10.92 (2026-04-10)

• **feat:** Add `DevAdmin` methods — `capture()` (5-param), `clearAll()`, `health()`, `unresolvedCount()`, `reset()`, `register()`.

• **feat:** Add `Server.start()` and `Server.stop()` for cross-framework parity.

• **feat:** Add `DatabaseResult.size()` method.

• **feat:** Add `DevReload.start()` and `DevReload.stop()`.

• **feat:** Add `ScssCompiler.compileScss()` method.

• **fix:** `autoCrud.ts` — fix spread syntax on non-iterable, add id in POST response, correct response format to `{data, meta}`, change validation status from 400 to 422.

• **parity:** 44/44 cross-framework features green. 2,752 tests passing.

v3.10.91 (2026-04-10)

• **feat:** Add parity methods — `GraphQLType.parse()`, `CorsMiddleware.isPreflight()`, `RateLimiterMiddleware.check()`.

• **breaking:** Rename `from()` → `fromTable()`, remove `template()` alias — align with Python canonical names.

v3.10.90 (2026-04-09)

• **docs:** Chapter 4 (Templates) — new "Dumping Values for Debugging" section covering both `{{ x|dump }}` and `{{ dump(x) }}` forms, the v3.10.88 `inspectValue()` inspector (circular refs, BigInt, Map/Set, Error, Date, class instances), 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 `renderDump()` helper (which delegates to the v3.10.88 `inspectValue()` inspector) and produce identical output.

• **security:** Dump is now **gated on** `TINA4_DEBUG=true`. In production (env var unset or `false`) both the filter and function silently return an empty `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:** 4 new tests in `frond.test.ts` covering `dump()`/`|dump` parity, debug-mode circular ref handling, production silencing for both forms.

v3.10.88 (2026-04-09)

• **fix:** `{{ value|dump }}` filter now handles complex objects safely. The previous implementation used `JSON.stringify` which crashed on circular references and BigInt, silently dropped functions/Symbols/`undefined`, and serialised `Map`/`Set`/`Error`/class instances as empty `{}`. Replaced with an `inspectValue()` inspector that matches PHP's `var_dump`, Python's `repr`, and Ruby's `inspect`:

• Circular references: `[Circular]`

• BigInt: `123n`

• Date: `Date(2026-04-09T13:00:00.000Z)`

• Map / Set: `Map(2) { "a" => 1, "b" => 2 }` / `Set(3) { 1, 2, 3 }`

• Error: `Error("boom")`

• Class instances: `User { name: "Alice", age: 30 }` (class name preserved)

• Functions: `[Function: name]`

• Depth-capped at 8 levels to prevent runaway graphs

• **test:** 11 new edge-case assertions in `frond.test.ts` (frond.test now 254 passing).

v3.10.87 (2026-04-09)

• **fix:** Dev toolbar no longer vanishes after a hot-reload. The CLI watcher used to call `server.router.clear()` on every file change — including template/CSS/JS asset edits — which left a brief window of 404 responses that bypass the dev toolbar injection. The watcher now reports whether a `.ts/.tsx/.js/.jsx` source file changed; router re-discovery only runs on code changes, and asset edits pass through without touching the router. Matches the PHP v3.10.87 fix.

v3.10.86 (2026-04-09)

• **feat:** `foreignKey` field type on `BaseModel` auto-wires both sides of a foreign key relationship. Declaring `user_id: { type: "foreignKey", references: "User" }` injects a `belongsTo` entry on the declaring model and a `hasMany` entry on the referenced model via a module-level FK registry. New static methods `_processForeignKeys()` and `_applyFkRegistry()` are called lazily before relationship resolution. Optional `relatedName` overrides the has-many key.

• **feat:** Cross-framework parity — same FK auto-wiring semantics now available in Python (`ForeignKeyField`), PHP (`$foreignKeys`), and Ruby (`foreign_key_field`)

• **docs:** Chapter 6 (ORM) updated with a new "foreignKey Field Type — 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 `validToken()` was changed to return bool — any code reading `request.user.sub` etc. would have failed silently or crashed

• **fix:** CSRF middleware was not correctly rejecting invalid tokens (null 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 — `joinRoom`, `leaveRoom`, `broadcastToRoom`, `getRoomConnections`, `roomCount`, `getClientRooms`

• **feat:** Queue signature parity — instance-scoped `push`/`pop`/`retry`, no topic params on public methods

• **feat:** Auth alias cleanup — removed `createToken`/`validateToken`, canonical `getToken`/`validToken`

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 Node.js follows semantic versioning. The major version (3) marks the initial Node.js launch — Tina4 Node.js is new in the v3 line, alongside Tina4 Ruby. The minor version tracks feature additions. The patch version tracks fixes, template engine corrections, and cross-framework parity updates.

This chapter covers every release from v3.0.0 through v3.10.x. Each section groups releases by minor version, lists features added, bugs fixed, and breaking changes with migration code where relevant.

---

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() returns boolean** — `model.load(sql, params)` calls selectOne internally, populates the instance, returns `true`/`false`. Use `findById()` 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** — `req.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** — interactive Node REPL with framework loaded (db, Router, Database, Log)

• **tina4 env** — interactive environment configuration

• **Brand update** — "TINA4 — The Intelligent Native Application 4ramework"

• **Dynamic version** — reads from package.json at runtime

• **Port kill-and-take-over** — default port always reclaimed

• **findAvailablePort** — checks 0.0.0.0 not 127.0.0.1

• **MongoDB adapter** (mongodb npm), **ODBC adapter** (odbc npm)

• **Pagination standardized** — limit/offset primary, merged dual-key response

• **Metrics dependency lines** — basename fix for correct rendering

• **autoMap uppercase** — snakeToCamel lowercases first

---

v3.10.57 (2026-04-02)

• **MongoDB adapter** — `initDatabase({ url: "mongodb://host:port/db" })`, requires `npm install mongodb`

- **ODBC adapter** — `initDatabase({ url: "odbc:///DSN=MyDSN" })`, requires `npm install odbc`
- **Pagination standardized** — limit/offset primary, merged dual-key toPaginate() response
- **Test port at +1000** — user testing port (e.g. 8148) stable, no hot-reload
- **Dynamic version** — read from package.json, no hardcoded constant
- **Metrics dependency lines** — fixed basename parsing
- **autoMap uppercase columns** — snakeToCamel lowercases first
- **ORM TINA4*DATABASE*URL discovery** — auto-connect from env for SQLite
- **108 features at 100% parity**, 2,646 tests

---

v3.10.54 (2026-04-02)

- **Auto AI dev port** — second HTTP server on port+1 with no-reload when TINA4_DEBUG=true
- **TINA4*NO*RELOAD** env var + --no-reload CLI flag
- **SQLite transaction safety** — commit/rollback/startTransaction guarded
- **autoMap uppercase columns** — snakeToCamel lowercases first
- **ORM TINA4*DATABASE*URL discovery** — auto-connect from env for SQLite
- **QueryBuilder docs** — added to ORM chapter

---

v3.10.48 — April 2, 2026

Bug Fixes

Cluster mode requires `TINA4_PRODUCTION=true` — Worker forking no longer auto-triggers when debug is off. Set `TINA4_PRODUCTION=true` env var or use `tina4 serve --production` to enable cluster mode.

---

v3.10.46 — April 1, 2026

Test Coverage

CSRF middleware expanded to 32 tests matching Python reference. Node.js now at 2,546 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 Node.js-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 — Node.js default port set to 7148 (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.

Server handler dispatch regex — Fixed a regex that required whitespace after `async` in handler functions. Transpiled auto-CRUD handlers producing `async(req,res)=>` were called with zero arguments, causing crashes.

Cluster mode in tests — Server-based tests now set `TINA4_DEBUG=true` to prevent cluster mode forking, which was causing ECONNREFUSED errors.

Test Coverage

Massive test expansion — 718 new tests added across Auth (+52), ORM (+30), FakeData (+48), Cache (+23), DevMailbox (+32), Static (+21), Queue (+20), Frond (+57), CLI scaffolding (55), Metrics (69), plus v3.10.44 feature tests and server test fixes. 2,530 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 npm. The overlay now shows a purple "ahead of npm" 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

`Database.getColumns(tableName)` — Returns `[{name, type, nullable, default, primaryKey}]` for each column. Uses `PRAGMA table_info` for SQLite and `information_schema.columns` for PostgreSQL/MySQL/MSSQL.

`Database.executeMany(sql, paramSets)` — Execute a SQL statement with multiple parameter arrays in a single transaction for atomicity and performance.

`BaseModel.create<T>(data)` — Static factory method: instantiates, saves, and returns the new record.

`BaseModel.find()` **and** `BaseModel.load()` — aliases for `findById()` (parity with Python, PHP, Ruby).

`seed` **CLI command** — `tina4nodejs seed` scans `src/seeds/*.ts` and executes them via `tsx`.

`Router.allRoutes()` — alias for `getRoutes()`.

---

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 regex-based TypeScript/JavaScript analysis 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

`npx tina4nodejs 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. Fixed `/__dev/` trailing slash returning 404.

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 (March 28–31, 2026)

The v3.10 line focused on ORM refinements, Frond template engine fixes, and cross-framework parity. Thirty-two patch releases landed in four days.

Features

autoMap for ORM field mapping (v3.10.1)

The ORM gained automatic translation between JavaScript camelCase and database snake_case. Set `autoMap = true` on a model and the framework handles the rest.

```
import { BaseModel } from "tina4-nodejs";

class User extends BaseModel {
  static tableName = "users";
  static autoMap = true;
  static fields = {
    id: { type: "integer" as const, primaryKey: true },
    firstName: { type: "string" as const },   // maps to first_name
    lastName: { type: "string" as const },     // maps to last_name
    createdAt: { type: "datetime" as const },  // maps to created_at
  };
}
```

Explicit `fieldMapping` entries take precedence over auto-generated ones. The two utilities `snakeToCamel()` and `camelToSnake()` are exported for direct use.

WSDL lifecycle hooks and dotted function names (v3.10.6)

WSDL services gained `beforeCall` and `afterCall` hooks. The Frond template engine learned to resolve dotted function names like `{{ utils.format(value) }}`.

ORM auto-commit on write operations (v3.10.13)

The ORM now commits after every `save()` and `delete()` call. Before this change, writes on SQLite would silently succeed in memory but never persist to disk unless you called `commit()` yourself.

get*next*id() for ID pre-generation (v3.10.14)

Models gained a `getNextId()` method. It queries the database engine for the next auto-increment value before the insert happens. Useful when you need the ID for a related record before you save the parent.

```
const nextId = await User.getNextId();
// Use nextId in a related record before saving the User
```

Template filters: to*json*, *tojson*, *j*sescape (v3.10.16)

Three new Frond filters for passing data from templates to JavaScript:

```
<script>
  const config = {{ settings|to_json }};
  const message = "{{ userInput|js_escape }}";
</script>
```

formTokenValue() in Frond templates (v3.10.23)

Templates gained a `formTokenValue()` function that generates a unique CSRF token per form. Each token carries a nonce in the JWT payload, so two forms on the same page get distinct tokens (v3.10.22).

```
<form method="POST" action="/submit">
  <input type="hidden" name="formToken" value="{{ formTokenValue() }}">
  <button type="submit">Send</button>
</form>
```

Arithmetic in set and expressions (v3.10.31)

The Frond engine learned arithmetic. `{% set total = price * quantity %}` and `{{ width + padding }}` now work as expected.

MCP server (v3.10.32)

Tina4 Node.js ships a built-in MCP (Model Context Protocol) server. AI coding tools can connect to your running application and inspect routes, models, and database schema.

Bug Fixes

Frond dict[variable_key] access (v3.10.11)

Variable keys in dictionary access were ignored. The engine treated `dict[myVar]` as a literal string lookup instead of resolving `myVar` first.

```
{# Before fix — broken: always looked up the literal string "myVar" #}
{% set key = "name" %}
{{ user[key] }}  {# returned undefined #}

{# After fix — works: resolves key to "name", then looks up user["name"] #}
{% set key = "name" %}
{{ user[key] }}  {# returns the user's name #}
```

Frond |replace filter backslash escaping (v3.10.15)

The `|replace` filter mangled backslashes. A replacement string containing `\n` would insert a literal newline instead of the two characters `\n`.

Frond variable resolution (v3.10.17)

Nested variable lookups in certain template constructs returned `undefined`. The engine now walks the scope chain correctly.

Frond inline-if with quoted strings (v3.10.19)

Inline conditionals broke when the true/false branches contained quoted strings with spaces. The parser split on whitespace inside the quotes.

Filters in if conditions (v3.10.21)

Filters inside `{% if %}` conditions were silently ignored. The condition evaluated the raw value instead of the filtered one.

```
{# Before fix — broken: |length filter ignored, condition tested the array itself #}
{% if items|length > 0 %}

{# After fix — works: |length runs first, condition compares the number #}
{% if items|length > 0 %}
```

Stale templates in dev mode (v3.10.24)

The dev server cached compiled templates and ignored file changes. Editing a template required a server restart. The fix reads the filesystem on every request in development mode, while production mode keeps the cache.

ORM save/delete transaction safety (v3.10.25)

SQLite threw "cannot commit — no transaction is active" when the ORM called `commit()` outside an explicit transaction. The ORM now wraps every `save()` and `delete()` in a `startTransaction()`/`commit()`/`rollback()` block.

```
// Before fix — threw on SQLite:
const user = new User({ firstName: "Alice" });
await user.save(); // Error: cannot commit — no transaction is active

// After fix — works on all database engines:
const user = new User({ firstName: "Alice" });
await user.save(); // Transaction handled internally
```

Frond macro HTML escaping (v3.10.27)

Macro output was HTML-escaped when used inside `{{ }}` expressions. A macro that generated `<div>` would render as `<div>`. Nested macros double-escaped. Macro output is now treated as safe HTML, matching standard Twig behaviour.

js*escape and to*json auto-escaping (v3.10.17–19)

The `js_escape` and `to_json` filters produced output that Frond then HTML-escaped. A JSON string like `{"key":"value"}` became `{"key":"value"}`. These filters now wrap their output in SafeString to bypass auto-escaping.

Firebird-Specific

Migration runner fixes (v3.10.10)

The migration runner generated SQLite-style `AUTOINCREMENT` and `TEXT` types for Firebird. Firebird needs generators and `VARCHAR`. The runner now emits the correct DDL and generates IDs from a `GEN_TINA4_MIGRATION_ID` sequence.

---

v3.9.x — QueryBuilder, Sessions, Path Injection (March 26–27, 2026)

The v3.9 line delivered three features that changed how developers write routes and query data.

Features

QueryBuilder (v3.9.0)

A fluent SQL builder that integrates with the ORM. Chain methods to build queries without writing raw SQL.

```
import { User } from "./orm/User.js";

// ORM integration
const admins = await User.query()
  .where("role = ?", ["admin"])
  .orderBy("name")
  .limit(10)
  .get();

// Standalone usage
import { QueryBuilder } from "tina4-nodejs";

const results = await QueryBuilder.from("orders")
  .where("total > ?", [100])
  .leftJoin("customers", "orders.customer_id = customers.id")
  .orderBy("total", "DESC")
  .limit(20)
  .get();

// Utility methods
const exists = await User.query().where("email = ?", [email]).exists();
const total = await User.query().where("active = ?", [true]).count();
const first = await User.query().where("id = ?", [1]).first();
```

The builder supports `select`, `where`, `orWhere`, `join`, `leftJoin`, `groupBy`, `having`, `orderBy`, `limit`, `first`, `count`, `exists`, and `toSql`.

Path parameter injection (v3.9.0)

Route handlers receive path parameters as named function arguments. The framework inspects the handler's parameter names and injects matching values.

```
import { Router } from "tina4-nodejs";

// The framework injects id as a typed argument
Router.get("/users/{id:int}", async (id, request, response) => {
  const user = await User.find(id);
  response.json(user);
});
```

Auto-start sessions (v3.9.0)

Every route handler receives a session object on `request.session` with zero configuration. No middleware to register. No setup code.

```
Router.post("/login", async (request, response) => {
```

```
  request.session.set("userId", 42);
  request.session.flash("message", "Welcome back.");
  response.redirect("/dashboard");
});

Router.get("/dashboard", async (request, response) => {
  const userId = request.session.get("userId");
  const flash = request.session.getFlash("message");
  response.render("dashboard", { userId, flash });
});
```

The session API: `get`, `set`, `delete`, `has`, `clear`, `destroy`, `save`, `regenerate`, `flash`, `getFlash`, `all`.

CSRF middleware and secure-by-default (v3.9.1)

POST, PUT, PATCH, and DELETE routes require authentication by default. The framework ships a `CsrfMiddleware` that validates session-bound form tokens.

```
// Routes that modify data are protected out of the box
Router.post("/api/orders", async (request, response) => {
  // Request must include a valid form token or auth header
  // Otherwise the framework returns 403
});
```

Queue parity (v3.9.1)

The queue system gained priority-based push, `size(status)` to count jobs by state, `job.retry()`, and `job.topic` for filtering.

NoSQL QueryBuilder and WebSocket backplane (v3.9.2)

The QueryBuilder gained MongoDB support. WebSocket servers gained a backplane for broadcasting across multiple server instances.

SameSite=Lax cookie default (v3.9.2)

Session cookies now set `SameSite=Lax` by default. This prevents CSRF attacks from cross-origin form submissions without breaking same-site navigation.

Breaking Changes

Secure-by-default for mutation routes (v3.9.1)

All POST, PUT, PATCH, and DELETE routes now require authentication. If your application has public mutation endpoints, mark them as open:

```
// BEFORE (v3.8.x) — all routes were open by default
Router.post("/api/feedback", async (request, response) => {
  // Anyone could call this
});

// AFTER (v3.9.x) — opt out of auth for public endpoints
Router.post("/api/feedback", async (request, response) => {
  // Now requires auth unless you explicitly open it
}).secure(false);
```

session.delete() replaces session.unset() (v3.9.0)

The session method was renamed for cross-framework parity.

```
// BEFORE (v3.8.x)
request.session.unset("userId");

// AFTER (v3.9.x)
request.session.delete("userId");
```

Bug Fixes

ESM compatibility (v3.9.4)

Internal `require()` calls broke on Node 22 with ESM-only configurations. All internal imports now use the ESM `import()` function.

Zero dependencies achieved (v3.9.3)

The `better-sqlite3` native module was the last remaining npm dependency. This release replaced it with Node's built-in `node:sqlite` module. `npm install` no longer needs a C++ compiler or `node-gyp`.

---

v3.8.x — Template Engine, Typed Params, Security (March 25–26, 2026)

The v3.8 line replaced the template engine, added typed route parameters, and introduced production-grade security middleware.

Features

Typed route parameters (v3.8.0)

Route paths gained type annotations. The framework validates and converts parameters before your handler runs.

```
Router.get("/products/{id:int}", async (request, response) => {
  // id is guaranteed to be an integer
  // /products/abc returns 404 automatically
});

Router.get("/prices/{amount:float}", async (request, response) => {
  // amount is a floating-point number
});

Router.get("/docs/{path:path}", async (request, response) => {
  // path captures everything including slashes: /docs/api/v2/users
});
```

Template fallback (v3.8.0)

Requesting `/hello` now serves `src/templates/hello.twig` or `src/templates/hello.html` when no route matches. No explicit route registration needed for static pages.

Connection pooling (v3.8.1)

Set `TINA4_DB_POOL=5` in your `.env` file and the framework creates five database connections. Requests distribute across them in a round-robin pattern.

SecurityHeadersMiddleware (v3.8.1)

A built-in middleware that sets `X-Frame-Options`, `Strict-Transport-Security`, `Content-Security-Policy`, and `X-Content-Type-Options` on every response.

Validator class (v3.8.1)

Input validation with structured error responses:

```
import { Validator } from "tina4-nodejs";

Router.post("/api/users", async (request, response) => {
  const errors = Validator.validate(request.body, {
    email: ["required", "email"],
    name: ["required", "minLength:2"],
    age: ["integer", "min:18"],
  });

  if (errors.length > 0) {
    return response.json({ errors }, 422);
  }
});
```

Upload size limit and Docker support (v3.8.1)

Set `TINA4_MAX_UPLOAD_SIZE=10mb` to cap file uploads. The `tina4 init` scaffolding now includes a multi-stage Alpine Dockerfile.

base64encode and base64decode Frond filters (v3.8.0)

```
{{ sensitiveId|base64encode }}
{{ encodedValue|base64decode }}
```

Production template caching (v3.8.0)

Template lookups are cached in production mode. In development mode, the framework reads the filesystem on every request so changes appear without a restart.

Breaking Changes

Frond replaces Twig dependency (v3.8.4)

The `@tina4/twig` npm package was removed. The framework now uses its built-in Frond engine for all template rendering. Frond supports the same Twig syntax, but if your templates relied on Twig-specific extensions, you need to rewrite them as Frond filters.

```
// BEFORE (v3.7.x) — Twig as a separate dependency
// package.json included "@tina4/twig": "^1.x"
// Templates used Twig-specific extensions

// AFTER (v3.8.x) — Frond is built in, zero dependencies
// Remove @tina4/twig from package.json
// Templates use Frond filters (same Twig syntax, built-in engine)
```

Groundwork for zero dependencies (v3.8.4)

v3.8.4 began migrating from `better-sqlite3` to Node's built-in `node:sqlite` module. The migration completed in v3.9.3.

---

v3.7.x — Template Auto-Serve, Firebird Migrations (March 25, 2026)

A focused release. Two features, no breaking changes.

Features

Template auto-serve at / (v3.7.0)

Place `index.html` or `index.twig` in `src/templates/` and the framework serves it at `/`. User-registered `GET /` routes take priority. When neither exists, the Tina4 landing page appears.

Firebird idempotent migrations (v3.7.0)

`ALTER TABLE ADD` statements on Firebird now check `RDB$RELATION_FIELDS` before executing. If the column exists, the migration logs "already applied" and moves on. Other databases and statement types are unaffected.

---

v3.6.x — Architectural Parity (March 25, 2026)

Features

src/orm/ as primary model directory (v3.6.0)

Models now live in `src/orm/` by default, matching the convention across all Tina4 frameworks. The framework still scans `src/models/` as a fallback.

Bug Fixes

Outdated API references (v3.6.0)

Internal references to deprecated function names (`createToken` instead of `getToken`, `validateToken` instead of `validToken`) and route parameter syntax were updated.

---

v3.5.x — Bundled Frontend, Middleware (March 25, 2026)

Features

Bundled tina4js.min.js (v3.5.0)

The reactive frontend library ships inside the framework. A 13.6 KB file gives your templates reactive signals, client-side routing, and API calls with zero additional installs.

session.clear() (v3.5.0)

Wipe all session data without destroying the session itself. The session ID and cookie persist.

```
// clear() removes data but keeps the session alive
request.session.clear();

// destroy() ends the session entirely
request.session.destroy();
```

Standardized middleware classes (v3.5.0)

Middleware follows a naming convention: `before*` classes run before the handler, `after*` classes run after. Three built-in middleware classes ship with the framework.

---

v3.4.x — Database, Auth, WebSocket, Uploads (March 24, 2026)

The v3.4 line added production-grade features across several subsystems.

Features

Database class wrapper (v3.4.0)

A constructor-based pattern for database connections, replacing bare function calls:

```
import { Database } from "tina4-nodejs";

const db = new Database("sqlite://data.db");
const result = await db.fetch("SELECT * FROM users WHERE active = ?", [true]);
```

DatabaseResult with columnInfo() (v3.4.0)

Query results return a `DatabaseResult` object. Call `columnInfo()` to inspect column names, types, and sizes without a separate schema query.

```
const result = await db.fetch("SELECT * FROM users LIMIT 1");
const columns = result.columnInfo();
// [{ name: "id", type: "INTEGER" }, { name: "email", type: "VARCHAR" }, ...]
```

Auth class wrapper (v3.4.0)

Authentication functions grouped into a class with `getToken()` and `validToken()` as the primary API. The old names `createToken` and `validateToken` remain as aliases.

```
import { Auth } from "tina4-nodejs";

const token = Auth.getToken({ userId: 42, role: "admin" });
const payload = Auth.validToken(token);
```

Redis session handler (v3.4.0)

Sessions can now persist to Redis for multi-server deployments. Set `TINA4_SESSION_HANDLER=redis` in your `.env` file.

Path-scoped WebSocket broadcast (v3.4.0)

Broadcast messages to WebSocket clients subscribed to a specific path:

```
Router.websocket("/chat/{room}", (connection, request) => {
```

```
  connection.onMessage((message) => {
    connection.broadcast(message); // Only reaches clients on the same path
  });
});
```

File uploads with raw Buffer and data_uri (v3.4.0)

Uploaded files include a raw `Buffer` and a `data_uri` template filter for embedding images directly in HTML.

Breaking Changes

WebSocket handler signature changed to 3 arguments (v3.4.0)

WebSocket handlers now receive `(connection, request, params)` instead of `(connection, request)`.

```
// BEFORE (v3.3.x)
Router.websocket("/ws", (connection, request) => {
  // No access to path params
});

// AFTER (v3.4.x)
Router.websocket("/ws/{room}", (connection, request, params) => {
  const room = params.room;
});
```

Auth function rename (v3.4.0)

`getToken()` and `validToken()` are now the primary function names. The old names `createToken` and `validateToken` continue to work as aliases but are deprecated.

```
// BEFORE (v3.3.x)
const token = Auth.createToken({ userId: 42 });
const valid = Auth.validateToken(token);

// AFTER (v3.4.0)
const token = Auth.getToken({ userId: 42 });
const valid = Auth.validToken(token);
```

Queue job file extension (v3.4.0)

Queue jobs on the file backend use `.queue-data` instead of `.json`. Existing `.json` job files need renaming or the queue treats them as new.

File upload format (v3.4.0)

Upload objects now use `{ type, content }` where `content` is base64-encoded. Legacy property names (`filename`, `data`) remain as aliases.

---

v3.3.x — Queue API, Field Mapping, Route Chaining (March 24, 2026)

Features

Queue API (v3.3.0)

Produce jobs, consume them with an async generator, and manage their lifecycle:

```
import { produce, consume, Job } from "tina4-nodejs";

// Produce a job
await produce("email-queue", { to: "user@example.com", subject: "Welcome" });

// Consume jobs
for await (const job of consume("email-queue")) {
  try {
    await sendEmail(job.data);
    job.complete();
  } catch (error) {
    job.fail(error.message);
  }
}
```

Switch between SQLite, RabbitMQ, Kafka, and MongoDB backends with a single `.env` variable: `TINA4_QUEUE_BACKEND=rabbitmq`.

ORM fieldMapping (v3.3.0)

Map JavaScript property names to database column names explicitly:

```
class User extends BaseModel {
  static fieldMapping = {
    firstName: "first_name",
    lastName: "last_name",
  };
}
```

Route chaining with .secure() and .cache() (v3.3.0)

Routes return a `RouteRef` that supports chainable modifiers:

```
Router.get("/api/products", handler)
  .secure()
  .cache(300); // Cache for 5 minutes
```

MongoDB queue backend (v3.3.0)

The queue system gained MongoDB as a backend. Set `TINA4_QUEUE_BACKEND=mongodb` in your `.env` file.

Database session handler (v3.3.0)

Sessions can persist to any supported database. Set `TINA4_SESSION_HANDLER=database`.

Dev admin improvements (v3.3.0)

Routes in the dev admin panel are now clickable links that open in a new tab. The error overlay shows full request details.

---

v3.2.x — Flexible Route Handlers (March 22, 2026)

Features

Zero-param and single-param route handlers (v3.2.0)

Route handlers accept multiple signatures. The framework inspects the function's parameter names and injects the right objects.

```
// All of these work:
Router.get("/health", () => ({ status: "ok" }));
Router.get("/health", (response) => response.json({ status: "ok" }));
Router.get("/health", (request, response) => response.json({ status: "ok" }));
```

Name your single parameter `request` or `req` and the framework passes the request object. Name it anything else and it receives the response.

---

v3.1.x — Response Parity, Routing API (March 21–22, 2026)

Features

Explicit routing methods (v3.1.0)

`Router.get()`, `Router.post()`, `Router.put()`, `Router.delete()`, and `Router.websocket()` replaced generic registration. Each method reads like what it does.

```
import { Router } from "tina4-nodejs";

Router.get("/users", listUsers);
Router.post("/users", createUser);
Router.put("/users/{id:int}", updateUser);
Router.delete("/users/{id:int}", deleteUser);
```

response.file() and response.render() (v3.1.0)

Two new response methods for serving files and rendering templates:

```
Router.get("/download", async (request, response) => {
  response.file("reports/quarterly.pdf");
});

Router.get("/dashboard", async (request, response) => {
  response.render("dashboard", { user: currentUser });
});
```

FetchResult with toPaginate() (v3.1.0)

Database queries return a `FetchResult` object with built-in pagination:

```
const result = await db.fetch("SELECT * FROM products", [], 20, 0);
const paginated = result.toPaginate();
```

```
// { data: [...], total: 156, page: 1, perPage: 20, totalPages: 8 }
```

ORM relationships (v3.1.0)

`hasMany`, `hasOne`, and `belongsTo` with eager loading:

```
class User extends BaseModel {
  static relationships = {
    posts: { type: "hasMany", model: "Post", foreignKey: "user_id" },
    profile: { type: "hasOne", model: "Profile", foreignKey: "user_id" },
  };
}

const user = await User.find(1, { include: ["posts", "profile"] });
```

Unified Cache, Messenger, and Queue (v3.1.0)

Switch between memory, Redis, and file-based caching with a single environment variable. The messenger and queue systems follow the same pattern. No code changes needed.

```
# .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_QUEUE_BACKEND=sqlite
```

tina4 generate command (v3.1.0)

Scaffold models, routes, migrations, and middleware from the command line:

```
tina4 generate model User
tina4 generate route api/products
tina4 generate migration add_email_to_users
tina4 generate middleware AuthCheck
```

Frond pre-compilation (v3.1.0)

The template engine caches compiled tokens. File rendering runs 2.8x faster than v3.0.0.

---

v3.0.0 — Initial Release (March 21, 2026)

The initial Node.js release. No Express. No Fastify. No dependencies.

Features

- **Native node:http** — The server uses Node's built-in HTTP module. Zero framework overhead.
- **TypeScript-first** — Strict mode, ESM only. No separate build step.

- **Database adapters** — SQLite, PostgreSQL, MySQL, MSSQL, and Firebird. Same API across all five.
- **File-based routing** — `src/routes/api/users/[id]/get.ts` maps to `GET /api/users/:id`.
- **Auto-CRUD** — Generate full REST endpoints from a model definition.
- **DevAdmin dashboard** — A built-in developer panel with route inspection and database tools.
- **AI integration** — Auto-detect and configure context for seven AI coding tools.
- **1,311 tests** across 43 test files.
- **Configurable port and host** — Default port 7148, binds to 0.0.0.0 for Docker.

```
import { startServer } from "tina4-nodejs";

startServer({ port: 7148 });
```

One import. One function call. The server starts and your application is live.

Pre-Release (rc.2–rc.5)

Four release candidates preceded v3.0.0. They stabilized the scaffolding, fixed the init command, added the error overlay, refined the landing page, and established the benchmark suite. If you started a project on a release candidate, upgrade to v3.0.0 and run `tina4 init` to regenerate your scaffolding files.

Version Timeline

Version	Date	Headline
v3.0.0	March 21	Initial release — zero dependencies, TypeScript-first
v3.1.0	March 21	Response parity, ORM relationships, unified cache/queue
v3.2.0	March 22	Flexible route handler signatures
v3.3.0	March 24	Queue API, field mapping, route chaining
v3.4.0	March 24	Database class, auth wrapper, Redis sessions, WebSocket broadcast
v3.5.0	March 25	Bundled frontend, standardized middleware
v3.6.0	March 25	src/orm/ as primary model directory
v3.7.0	March 25	Template auto-serve, Firebird idempotent migrations
v3.8.0	March 25	Typed route params, template fallback, Frond replaces Twig
v3.9.0	March 26	QueryBuilder, path injection, auto-start sessions
v3.10.0	March 28	Cached Frond instances, autoMap, ORM transactions, template fixes
v3.10.32	March 31	MCP server, arithmetic expressions, current stable

Forty-two releases in eleven days. Each one a step closer to the framework the code deserves.

Complete Feature List

This chapter catalogs all 45 features in Tina4 for Node.js, grouped by category. Use it as a reference when starting a project or auditing what you need.

---

1. Routing and HTTP

#	Feature	Class / Function	Chapter
1	**HTTP Routing**	`Router.get/post/put/delete/patch/all`	2
2	**Route Parameters**	`req.params.id`, `{id:int}`, `{slug}`	2
3	**Request & Response**	`req.body`, `req.query`, `res.json`, `res.render`, `res.status`	3
4	**Middleware**	Route-level string middleware, `"auth"`, `"ResponseCache:300"`	10
5	**Static Files**	Served from `public/` automatically	1
6	**CORS**	`TINA4_CORS_ORIGINS`, `TINA4_CORS_HEADERS` env vars	10

---

2. Templating

#	Feature	Class / Function	Chapter
7	**Frond Templates**	`res.render("page.frond", data)`	4
8	**Template Variables**	`{{variable}}`, `{{nested.field}}`	4
9	**Template Includes**	`{{> partial.frond}}`	4
10	**Template Helpers**	`{{#if}}`, `{{#each}}`, `{{#unless}}`	4

---

3. Database and ORM

#	Feature	Class / Function	Chapter
11	**Database Connection**	`Database.getConnection()`, `.env` config	5
12	**Raw Queries**	`db.fetchAll`, `db.fetchOne`, `db.execute`	5
13	**Named Parameters**	`:paramName` in SQL, bound from object	5
14	**ORM Models**	`class Product extends Model`	6
15	**ORM CRUD**	`Model.find`, `.save`, `.delete`, `.load`	6
16	**Query Builder**	`db.select().fromTable().where().limit().fetch()`	7
17	**Migrations**	Auto-migrations from model definitions	5
18	**Sequence / next_id**	`Database.get_next_id("table")`	5
19	**Multi-DB Support**	SQLite, PostgreSQL, MySQL, MSSQL	5

---

4. Authentication and Sessions

#	Feature	Class / Function	Chapter
20	**JWT Authentication**	`auth` middleware, `req.user`	8
21	**Login / Logout**	`POST /api/login`, `POST /api/logout`	8
22	**Password Hashing**	`hashPassword`, `verifyPassword`	8
23	**Sessions**	`req.session.get/set`, file/Redis/DB backends	9
24	**Cookies**	`res.cookie`, `req.cookies`, signed cookies	9
25	**CSRF Protection**	`CSRF` middleware, `csrfToken`	10

---

5. Background Processing

#	Feature	Class / Function	Chapter
26	**Queue System**	`Queue.push`, `.consume`, `.pop`, `.retry`	12
27	**Queue Backends**	File (default), RabbitMQ, Kafka, MongoDB	12
28	**Service Runner**	`Service`, `ServiceRunner`	26
29	**Scheduled Services**	`interval`, `runOnStart`, `restartDelay`	26

---

6. Caching

#	Feature	Class / Function	Chapter
30	**Response Cache**	`"ResponseCache:300"` middleware	11
31	**Cache API**	`cacheGet`, `cacheSet`, `cacheDelete`	11
32	**Cache Backends**	Memory (default), Redis, file	11
33	**Cache Statistics**	`cacheStats()`	11

---

7. Communication

#	Feature	Class / Function	Chapter
34	**Email (SMTP)**	`Messenger` class	16
35	**WebSocket**	`WebSocket` server, `ws.on`, `ws.send`	23
36	**Events System**	`Events.on`, `.emit`, `.once`, `.off`, `.clear`	13
37	**SSE / Streaming**	`response.stream(generator)` for Server-Sent Events	—

---

8. Utilities and Services

#	Feature	Class / Function	Chapter
38	**Structured Logging**	`Log.info/debug/warn/error`, `TINA4_LOG_LEVEL`	15
39	**Localization**	`I18n`, JSON locale files, `t(key, vars)`	14
40	**API Client**	`api` singleton, `get/post/put/delete`	21
41	**DI Container**	`Container`, `register`, `singleton`, `get`, `has`	25
42	**WSDL / SOAP**	`WSDL.load`, `wsdl_operation`, SOAP server	24
43	**GraphQL**	Schema, resolvers, `Router.graphql`	22

---

9. Developer Experience

#	Feature	Class / Function	Chapter
44	**Swagger / OpenAPI**	Auto-generated from JSDoc `@noauth`, `@path`	20
45	**Dev Dashboard**	`/__dev` — routes, queues, cache, logs	29

---

2. Feature Details

Routing and HTTP

HTTP Routing — Define routes with `Router.get("/path", handler)`. Handlers are async functions that receive `(req, res)`. Supports all HTTP methods. Routes are auto-loaded from `src/routes/`.

Route Parameters — Typed parameters in curly braces: `{id}` is a string, `{id:int}` is an integer, `{slug}` matches URL-safe strings. Parameters are available as `req.params.id`.

Request and Response — `req.body` is the parsed JSON request body. `req.query` holds query string values. `res.json(data)` sends JSON. `res.render("template.frond", data)` renders a template. `res.status(404).json({})` chains status and body.

Middleware — Applied as a string in the route definition: `Router.get("/path", handler, "auth,ResponseCache:300")`. Multiple middleware separated by commas. Custom middleware registered with `Router.middleware("name", fn)`.

Static Files — Everything in the `public/` directory is served automatically. `/public/logo.png` is available at `/logo.png`.

CORS — Set `TINA4_CORS_ORIGINS=*` (or a specific domain) and `TINA4_CORS_HEADERS` to control which headers are allowed. Preflight `OPTIONS` requests are handled automatically.

Templating

Frond Templates — Tina4's built-in templating language. Files use the `.frond` extension. No separate template engine to install.

Template Variables — `{{variable}}` renders a string. `{{user.name}}` accesses nested properties. `{{price | currency}}` applies a filter.

Template Includes — `{{> header.frond}}` includes another template. Included templates share the same variable scope.

Template Helpers — `{{#if condition}}...{{/if}}` for conditionals. `{{#each items}}...{{/each}}` for loops. `{{#unless condition}}` for negation.

Database and ORM

Database Connection — Configured via `.env`: `TINA4_DATABASE_URL`. Supports SQLite (`sqlite:./data/app.db`), PostgreSQL (`pgsql:host=...`), MySQL, and MSSQL. Accessed via `Database.getConnection()`.

Raw Queries — `db.fetchAll(sql, params)` returns an array of rows. `db.fetchOne(sql, params)` returns one row or null. `db.execute(sql, params)` runs a non-returning statement.

Named Parameters — Use `:name` in SQL. Pass values as `{ name: value }`. Never interpolate user input directly into SQL -- always use named parameters.

ORM Models — Extend `Model`, declare fields. Tina4 generates the table if it does not exist. `Product.find({ category: "Electronics" })` queries without SQL.

Query Builder — Chainable: `.select("name", "price").fromTable("products").where("active", true).orderBy("name").limit(20).fetch()`. Composes complex queries without raw SQL.

Migrations — Tina4 compares the model definition to the live table schema and adds missing columns on startup. No migration files needed for additive changes.

Sequence / next_id — `Database.get_next_id("invoices")` returns a race-safe, monotonically increasing ID using a sequence table. Safe for concurrent requests.

Multi-DB Support — Switch databases by changing `TINA4_DATABASE_URL`. Application code is unchanged.

Authentication and Sessions

JWT Authentication — `POST /api/login` with `{ username, password }` returns a JWT. Include it as `Authorization: Bearer <token>`. Protected routes use the `"auth"` middleware string.

Login and Logout — Built-in login/logout routes. Customize by overriding the handlers. `POST /api/logout` invalidates the session.

Password Hashing — `hashPassword(plain)` returns a bcrypt hash. `verifyPassword(plain, hash)` validates. Never store plain passwords.

Sessions — `req.session.set("key", value)` and `req.session.get("key")`. Backend: file (default), Redis (`TINA4_SESSION_BACKEND=redis`), or database.

Cookies — `res.cookie("name", "value", { httpOnly: true, secure: true })` sets a cookie. `req.cookies.name` reads it.

CSRF Protection — The `CSRF` middleware validates a token on mutating requests. Include the token from `req.csrfToken()` in forms or AJAX calls.

Background Processing

Queue System — `queue.push(payload)` enqueues a job. `queue.consume("topic")` is a generator that yields jobs. Call `job.complete()` or `job.fail()` after each.

Queue Backends — File-based by default (`TINA4_QUEUE_BACKEND=file`). Switch to RabbitMQ, Kafka, or MongoDB with one env var change. No code changes.

Service Runner — `ServiceRunner` manages multiple `Service` instances. Each runs its `run()` method on an interval. Crashes restart automatically.

Scheduled Services — Control timing with `interval` (ms), `runOnStart` (boolean), `restartDelay` (ms), and `maxRestarts`.

Caching

Response Cache — `"ResponseCache:300"` as middleware caches the entire HTTP response for 300 seconds. Cache key includes the full URL with query parameters.

Cache API — `cacheSet(key, value, ttlSeconds)`, `cacheGet(key)`, `cacheDelete(key)`. Use for custom caching logic: cache-aside, write-through, event-based invalidation.

Cache Backends — Memory (default), Redis (`TINA4_CACHE_BACKEND=redis`), file (`TINA4_CACHE_BACKEND=file`). Backend is transparent to application code.

Cache Statistics — `cacheStats()` returns hits, misses, hit rate, entry count, and memory usage. Visible in the dev dashboard.

Communication

Email — `Messenger` reads SMTP config from `.env`. `mailer.send({ to, subject, html, text })`. Attachments supported. Dev mode intercepts emails and shows them in the dashboard.

WebSocket — Real-time bidirectional communication. `Router.ws("/ws/chat", handler)`. `ws.on("message", fn)`, `ws.send(data)`, `ws.broadcast(data)`.

Events System — In-process publish/subscribe. `Events.on("event", handler, priority)`. `Events.emit("event", payload)`. `Events.once` for one-shot listeners. `Events.off` and `Events.clear` for cleanup.

Utilities and Services

Structured Logging — `Log.info/debug/warn/error(message, context)`. Outputs JSON. Level controlled by `TINA4_LOG_LEVEL` env var. Context fields merged into every log entry.

Localization — `I18n` loads JSON locale files from `locales/`. `i18n.t("key", { variable: "value" })` with interpolation. Automatic fallback to default locale for missing keys.

API Client — `api` singleton from `@tina4/core`. `api.configure({ baseUrl, headers, timeout })`. Methods: `api.get`, `api.post`, `api.put`, `api.patch`, `api.delete`. Consistent response: `{ ok, status, data, error }`.

DI Container — `Container` with `register(name, factory)` (transient) and `singleton(name, factory)`. Resolve with `get<Type>(name)`. `has(name)` for existence checks. `reset()` for test cleanup.

WSDL / SOAP — Consume SOAP with `WSDL.load(url)` and `client.call(operation, params)`. Publish SOAP with `wsdl_operation` decorator and `WSDL` server class. Auto WSDL document generation.

GraphQL — Schema-first or code-first. Resolvers registered with `Router.graphql`. Introspection enabled in development. Playground available at `/graphql`.

Developer Experience

Swagger / OpenAPI — Routes annotated with JSDoc comments generate an OpenAPI 3.0 spec automatically. Swagger UI available at `/swagger`. `@noauth` marks public endpoints. `@path`, `@query`, and `@body` document parameters.

Dev Dashboard — Available at `/__dev` in development mode. Shows: registered routes, active queue jobs, cache entries and hit rates, recent log output, registered services and their status, environment variable summary.

---

3. Environment Variables Reference

```
# Server
TINA4_PORT=7148
TINA4_ENV=development

# Database
TINA4_DATABASE_URL=sqlite:./data/app.db

# Cache
TINA4_CACHE_BACKEND=memory
TINA4_CACHE_URL=localhost
TINA4_CACHE_URL=6379

# Queue
TINA4_QUEUE_BACKEND=file
TINA4_QUEUE_PATH=./data/queue

# Session
TINA4_SESSION_BACKEND=file
TINA4_SECRET=your-secret-here

# Email
TINA4_MAIL_HOST=smtp.example.com
TINA4_MAIL_PORT=587
TINA4_MAIL_USERNAME=noreply@example.com
TINA4_MAIL_PASSWORD=your-app-password
TINA4_MAIL_ENCRYPTION=tls

# Auth
TINA4_SECRET=your-jwt-secret
TINA4_TOKEN_LIMIT=86400

# Logging
TINA4_LOG_LEVEL=info

# CORS
TINA4_CORS_ORIGINS=*
TINA4_CORS_HEADERS=Content-Type,Authorization
```

---

4. Package Imports Reference

```
// Core framework
import { Tina4, Router } from "tina4-nodejs";

// Database and ORM
import { Database } from "tina4-nodejs/orm";
import { Model } from "tina4-nodejs/orm";

// Auth
import { hashPassword, verifyPassword } from "tina4-nodejs";

// Cache
import { cacheGet, cacheSet, cacheDelete, cacheStats } from "tina4-nodejs";

// Queue
import { Queue } from "tina4-nodejs";

// Events
import { Events } from "tina4-nodejs";

// Logging
import { Log } from "tina4-nodejs";

// Localization
import { I18n } from "tina4-nodejs";

// Email
import { Messenger } from "tina4-nodejs";

// DI Container
import { Container } from "tina4-nodejs";

// Service Runner
import { Service, ServiceRunner } from "tina4-nodejs";

// WSDL / SOAP
import { WSDL, wsdl_operation } from "tina4-nodejs";

// API Client
import { api } from "@tina4/core";
```