Tina4Delphi Developer

FireMonkey, FireDAC, and cross-platform native apps

v3.12.4 • tina4.com

The Intelligent Native Application 4ramework

Table of Contents

Getting Started 24

Your First 10 Minutes . . . 24

1. What Is Tina4 Delphi . . . 24

What It Is Not . . . 24

2. Prerequisites . . . 25

3. Installation . . . 25

Step 1: Clone the Repository . . . 25

Step 2: Open the Project Group . . . 25

Step 3: Build and Install the Runtime Package . . . 25

Step 4: Build and Install the Design-Time Package . . . 25

Step 5: Verify . . . 25

4. SSL Setup . . . 25

Windows . . . 26

Quick Test . . . 26

5. Available Components . . . 26

6. Your First App: API Data in a Grid . . . 27

Step 1: Create the Project . . . 27

Step 2: Drop Components on the Form . . . 27

Step 3: Configure the REST Client . . . 28

Step 4: Configure the REST Request . . . 28

Step 5: Wire the Button . . . 28

Step 6: Run . . . 29

What Just Happened . . . 29

7. Quick Wins with Tina4Core . . . 29

Fetch JSON from an API . . . 29
Database Query to JSON . . . 29
JSON to MemTable . . . 30
Upload a File . . . 30
Convert Between Naming Conventions . . . 30
8. Exercise: Build a Weather Dashboard . . . 30
Requirements . . . 30
Hints . . . 30
Solution . . . 31
9. Common Gotchas . . . 33
SSL DLLs Missing . . . 33
Wrong DLL Bitness . . . 33
Design-Time Package Not Installed . . . 33
Library Path Missing . . . 33
TJSONObject Memory Leaks . . . 33
10. What Just Happened . . . 33
Summary . . . 34
REST APIs **35**
Two Ways to Talk to the Outside World . . . 35
1. TTina4REST -- Base Configuration . . . 35
Design-Time Setup . . . 35
Runtime Configuration . . . 35
Bearer Token Authentication . . . 35
One Component Per API . . . 36
2. Direct REST Calls . . . 36
GET . . . 36
POST . . . 37

PATCH (Partial Update) . . . 37
PUT (Full Replace) . . . 37
DELETE . . . 38
Method Reference . . . 39
3. Authentication Patterns . . . 39
Basic Auth . . . 39
Bearer Token (Static) . . . 40
Bearer Token (Login Flow) . . . 40
Custom Headers . . . 40
4. TTina4RESTRequest -- Declarative REST . . . 41
Basic GET with Auto MemTable Population . . . 41
POST with RequestBody . . . 41
PUT / PATCH / DELETE . . . 41
5. Master/Detail with Parameter Injection . . . 42
Setup . . . 42
How It Works . . . 42
Multiple Placeholders . . . 42
6. POST from SourceMemTable . . . 43
7. Async Execution . . . 43
Thread Safety Rules . . . 43
8. Events . . . 44
OnExecuteDone . . . 44
OnAddRecord . . . 44
9. Complete Example: Customer Management Panel . . . 44
Form Design . . . 44
Implementation . . . 46
10. Exercise: Product Catalog . . . 49

Requirements . . . 49
Solution . . . 50
11. Common Gotchas . . . 52
Forgetting to Free TJSONObject . . . 52
Not Checking StatusCode . . . 52
Async Thread Safety . . . 53
DataKey Mismatch . . . 53
BaseUrl Trailing Slash . . . 53
Summary . . . 54
JSON & Data Binding **55**
The Bridge Between APIs and Grids . . . 55
1. JSON Parsing Utilities . . . 55
StrToJSONObject . . . 55
StrToJSONArray . . . 55
StrToJSONValue . . . 56
BytesToJSONObject . . . 56
GetJSONFieldName . . . 56
2. TTina4JSONAdapter -- Static JSON to MemTable . . . 56
From Static JSON . . . 57
From MasterSource . . . 57
Sync Mode . . . 57
3. Database to JSON . . . 58
GetJSONFromDB . . . 58
With Parameters . . . 58
GetJSONFromTable . . . 59
4. JSON to MemTable . . . 59
GetFieldDefsFromJSONObject . . . 59

PopulateMemTableFromJSON . . . 59
Clear Mode (Default) . . . 60
Sync Mode . . . 60
PopulateTableFromJSON . . . 60
5. Naming Conventions . . . 60
CamelCase . . . 61
SnakeCase . . . 61
6. Complete Example: Data Import/Export Tool . . . 61
7. Complete Example: Master-Detail Pattern . . . 63
8. Exercise: JSON Viewer . . . 66
Requirements . . . 66
Solution . . . 67
9. Common Gotchas . . . 69
TJSONObject Memory Management . . . 69
Nested JSON Becoming ftMemo Fields . . . 70
Sync Mode Without IndexFieldNames . . . 70
DataKey Does Not Exist . . . 70
Date Fields Not Parsing . . . 70
Summary . . . 71
HTML Rendering **72**
A Web Browser Inside Your Desktop App . . . 72
1. Basic Usage . . . 72
Updating Content . . . 72
2. Supported HTML Elements . . . 72
Block Elements . . . 72
Inline Elements . . . 73
Lists . . . 73

Tables . . . 73
Images . . . 73
3. CSS Support . . . 74
External Stylesheets . . . 74
Style Blocks . . . 74
Inline Styles . . . 74
Selector Support . . . 74
Custom Properties (CSS Variables) . . . 75
Supported CSS Properties . . . 75
4. Form Controls . . . 75
5. Events . . . 76
OnFormSubmit . . . 76
OnFormControlChange . . . 77
OnFormControlClick . . . 77
OnLinkClick . . . 77
Event Reference . . . 78
6. onclick and RTTI -- Calling Pascal from HTML . . . 78
Step 1: Register Your Object . . . 78
Step 2: Write the Target Method . . . 78
Step 3: Call from HTML . . . 79
Dynamic Parameter Expressions . . . 79
7. DOM Manipulation . . . 79
Get and Set Values . . . 80
Enable/Disable Controls . . . 80
Show/Hide Elements . . . 80
Change Text Content . . . 80
Change Styles . . . 80

Set Attributes . . . 80
Force Refresh . . . 80
DOM Method Reference . . . 81
8. Image Loading and Caching . . . 81
9. Complete Example: Login Form with Validation . . . 81
10. Complete Example: Interactive Dashboard . . . 84
11. Exercise: Contact Form . . . 87
Requirements . . . 87
Solution . . . 88
12. Common Gotchas . . . 90
Forgetting to Set Cache Directory for Images . . . 90
RTTI Method Not Found . . . 91
Form Control Name Matching . . . 91
Escaped Quotes in HTML Strings . . . 91
Summary . . . 92
Page Navigation **93**
A Single-Page App Inside Your Desktop App . . . 93
1. TTina4HTMLPages Basics . . . 93
Setup . . . 93
2. Creating Pages at Design Time . . . 93
3. Creating Pages at Runtime . . . 94
4. Setting the Default Page . . . 95
5. Navigation via Anchor Links . . . 95
6. Programmatic Navigation . . . 95
Reading the Active Page . . . 96
7. Page Properties . . . 96
8. Component Properties . . . 96

9. Events . . . 97
OnBeforeNavigate . . . 97
OnAfterNavigate . . . 97
10. Using Twig Templates in Pages . . . 97
File-Based Templates . . . 98
11. Complete Example: Multi-Page Admin App . . . 99
12. Complete Example: Navigation Guards . . . 103
13. Exercise: Wizard / Step-by-Step Form . . . 106
Requirements . . . 106
Solution . . . 108
14. Common Gotchas . . . 114
Page Name Case Sensitivity . . . 114
Circular Navigation in OnBeforeNavigate . . . 114
TwigContent vs HTMLContent Priority . . . 114
Styles Not Persisting Between Pages . . . 114
Form Data Lost on Navigation . . . 115
Summary . . . 115
Twig Templates **116**
HTML Without the String Concatenation Hell . . . 116
1. TTina4Twig Standalone Usage . . . 116
Create and Render . . . 116
Render from a String . . . 117
Passing Complex Data . . . 117
2. Variables . . . 117
Simple Variables . . . 117
Nested Properties . . . 117

Setting Variables Inside Templates . . . 117

3. Control Structures . . . 118

if / elseif / else . . . 118

for Loops . . . 118

with Blocks . . . 119

4. Template Inheritance . . . 119

Base Template (base.html) . . . 119

Child Template (dashboard.html) . . . 120

include . . . 120

Setting the Template Path in Pascal . . . 120

5. Macros . . . 120

Define a Macro . . . 121

Call a Macro . . . 121

A More Complex Macro . . . 121

6. Filters Reference . . . 121

String Filters . . . 122

Number Filters . . . 123

Array Filters . . . 124

Date Filters . . . 125

Encoding Filters . . . 126

Other Filters . . . 126

7. Functions . . . 127

8. Operators . . . 127

9. Integration with TTina4HTMLRender . . . 128

Setting Variables and Rendering . . . 128

File-Based Templates with the Renderer . . . 128

Combining with REST Data . . . 128

10. Complete Example: Email Template System . . . 129
File Structure . . . 129
base.html . . . 130
notification.html . . . 130
order-confirm.html . . . 131
Pascal Code to Render . . . 132
11. Complete Example: Report Generator . . . 132
report-catalog.html . . . 133
Pascal Code . . . 135
Exercise: Invoice Template . . . 135
Solution . . . 136
Common Gotchas . . . 138
WebSockets **140**
Real-Time Without Polling . . . 140
1. TTina4WebSocketClient Overview . . . 140
Component Setup . . . 140
2. Basic Connection . . . 141
Design-Time Configuration . . . 141
Runtime Connection . . . 141
Event Handlers . . . 142
3. Sending Messages . . . 142
Send Text . . . 142
Send JSON . . . 143
Check Connection Before Sending . . . 143
4. Auto-Reconnect Behavior . . . 143
Configuration . . . 143
Reconnect Flow . . . 143

Handling Reconnection in Code . . . 144
5. Ping/Pong Keepalive . . . 144
6. Binary vs Text Messages . . . 144
Text Messages . . . 144
Binary Messages . . . 145
7. Disconnecting . . . 145
Graceful Close . . . 145
Cleanup on Form Destroy . . . 145
8. Complete Example: Real-Time Chat Client . . . 146
Form Layout . . . 146
Implementation . . . 147
9. Complete Example: Live Data Feed . . . 151
Implementation . . . 152
10. Exercise: Notification System . . . 156
Solution . . . 157
Common Gotchas . . . 162
Socket Server **164**
Raw TCP Without the Ceremony . . . 164
1. TTina4SocketServer Overview . . . 164
Component Setup . . . 164
2. Properties . . . 164
Event Signature . . . 165
3. Basic TCP Server . . . 165
Design-Time Setup . . . 165
Handling Messages . . . 166
Starting and Stopping . . . 166

4. Practical Examples . . . 166
Echo Server . . . 166
JSON Command Server . . . 166
Telemetry Receiver . . . 167
5. Thread Safety . . . 168
6. Lifecycle . . . 168
7. When to Use Socket Server vs WebSocket Client . . . 169
8. Gotchas . . . 169
Core Utilities **170**
The Swiss Army Knife You Already Have . . . 170
1. String Helpers . . . 170
CamelCase . . . 170
SnakeCase . . . 170
When You Need Both . . . 171
2. GUID Generation . . . 171
GetGUID . . . 171
3. Date Utilities . . . 171
IsDate . . . 171
GetJSONDate . . . 172
JSONDateToDateTime . . . 172
4. Encoding . . . 172
DecodeBase64 . . . 172
FileToBase64 . . . 173
BitmapToBase64EncodedString . . . 173
BitmapToSkiaWepPEncodedString . . . 174
5. JSON Parsing . . . 174
StrToJSONObject . . . 174

StrToJSONArray . . . 174
StrToJSONValue . . . 174
BytesToJSONObject . . . 175
GetJSONFieldName . . . 175
6. Database to JSON . . . 175
GetJSONFromDB . . . 175
With Parameters . . . 176
Serving JSON from a Database . . . 176
GetJSONFromTable . . . 176
7. JSON to MemTable . . . 177
GetFieldDefsFromJSONObject . . . 177
PopulateMemTableFromJSON . . . 177
PopulateTableFromJSON . . . 177
8. HTTP Requests . . . 178
SendHttpRequest . . . 178
POST with JSON Body . . . 179
With Basic Auth . . . 179
PATCH and DELETE . . . 179
SendMultipartFormData . . . 179
9. Shell Commands . . . 180
ExecuteShellCommand . . . 180
Cross-Platform Commands . . . 181
10. Complete Example: File Upload Utility . . . 181
11. Complete Example: Database Sync Tool . . . 183
12. Exercise: CLI Wrapper . . . 187
Requirements . . . 187
Solution . . . 188

Common Gotchas . . . 190

Building a CRUD Application 192

A Contact Manager From Scratch . . . 192

1. What We Are Building . . . 192

2. The REST API . . . 193

Endpoints . . . 193

Response Format . . . 193

3. Project Setup . . . 194

New FMX Project . . . 194

Form Components . . . 194

Layout Structure . . . 195

4. REST Configuration . . . 195

5. Grid Setup . . . 196

6. List Contacts . . . 197

Fetching and Displaying . . . 197

7. View Contact Detail . . . 197

Grid Click Handler . . . 198

Contact Detail Template (templates/contact-detail.html) . . . 198

Show Detail . . . 199

8. Create Contact . . . 200

Contact Form Template (templates/contact-form.html) . . . 200

Show Create Form . . . 201

Handle Form Submission . . . 202

9. Update Contact . . . 203

Edit Button Handler . . . 203

10. Delete Contact . . . 204

Delete with Confirmation . . . 204
11. Search and Filter . . . 204
Option 1: Filter MemTable Locally . . . 204
Option 2: Search via API . . . 205
Clear Search . . . 206
12. Polish: Status, Loading, and Error Handling . . . 206
Status Bar . . . 206
Cancel Form Navigation . . . 206
Error Handling Wrapper . . . 207
13. Full Source Code . . . 207
14. Template Files . . . 214
Exercise: Extend the Contact Manager . . . 214
Solution: CSV Export . . . 215
Solution: Category Filter . . . 215
Real-World Integration **217**
Making the Components Talk . . . 217
1. Philosophy: Components as a Pipeline . . . 217
2. Pattern 1: REST to MemTable to Grid . . . 217
Implementation . . . 218
With Periodic Refresh . . . 218
3. Pattern 2: REST to JSON Adapter to Multiple MemTables . . . 218
Scenario . . . 218
Implementation . . . 219
4. Pattern 3: HTML Render + Twig + REST Data . . . 219
Implementation . . . 220
The Template (product-card.html) . . . 220
5. Pattern 4: Master-Detail Chains . . . 221

Implementation . . . 221
Displaying the Chain . . . 222
6. Pattern 5: Two-Way Sync . . . 222
Implementation . . . 223
Using SourceMemTable for Bulk POST . . . 223
7. Pattern 6: HTML Pages + REST . . . 224
Implementation . . . 225
Navigation Template . . . 226
8. Pattern 7: WebSocket + HTML Render . . . 226
Implementation . . . 227
Status Panel Template . . . 228
9. Complete Example: Product Management Dashboard . . . 228
Architecture . . . 228
Form Components . . . 229
Implementation . . . 230
Template Files . . . 238
10. Integration with Tina4 Backend . . . 240
Tina4 Python Backend . . . 240
Tina4 PHP Backend . . . 240
Tina4 Node.js Backend . . . 240
Exercise: Monitoring Dashboard . . . 240
Requirements . . . 241
Solution Outline . . . 242
Claude Code Integration **244**
Let AI Write Your Delphi . . . 244
1. What Is MCP . . . 244

2. The Pascal MCP Server . . . 244
What the Server Provides . . . 245
Installation . . . 245
3. Configuration . . . 246
Project-Level Configuration . . . 246
Global Configuration . . . 246
Verifying the Setup . . . 246
4. Specifying a Compiler . . . 246
5. Preview Bridge Setup . . . 247
6. What Claude Can Do . . . 247
Compile Single Files . . . 247
Compile Full Delphi Projects . . . 248
Generate Project Templates . . . 248
Launch GUI Applications . . . 248
Interact with Running Applications . . . 249
Parse Form Files . . . 249
7. Walkthrough: Building a Weather App . . . 250
Step 1: Describe What You Want . . . 250
Step 2: Claude Generates the Project . . . 250
Step 3: Claude Compiles and Runs . . . 252
Step 4: Claude Tests the App . . . 252
Step 5: Iterate . . . 252
8. Walkthrough: Modifying an Existing Form . . . 252
Step 1: Let Claude Understand the Project . . . 252
Step 2: Request a Change . . . 253
Step 3: Claude Modifies the Files . . . 253
Step 4: Claude Compiles and Tests . . . 254

9. Tips for Effective AI-Assisted Delphi Development . . . 254
Be Specific About Components . . . 254
Reference Existing Code . . . 254
Let Claude Fix Its Own Mistakes . . . 254
Use the Preview Bridge for Visual Feedback . . . 254
Keep Your CLAUDE.md Updated . . . 254
10. Exercise: Build a Calculator from Scratch . . . 255
Requirements . . . 255
Suggested Prompts . . . 255
Solution . . . 255
11. Common MCP Issues . . . 256
Summary . . . 256
Building a Complete Application **257**
The Admin Dashboard . . . 257
1. What We Are Building . . . 257
2. Project Setup . . . 257
File Structure . . . 258
The Main Form . . . 258
Project File . . . 258
3. Main Unit -- Declarations . . . 259
4. Initialization . . . 261
5. Authentication . . . 262
6. Page Definitions . . . 263
7. Dashboard Page with Stats . . . 264
8. Users Page -- CRUD Operations . . . 266
Creating a User . . . 268
Updating a User . . . 269

Deleting a User 269

9. Settings Page 270

10. RTTI Event Handlers 271

11. Notifications 273

12. WebSocket Notifications 273

13. Error Handling 275

14. Using Twig Templates Instead of String Concatenation 276

15. Project Organization 277

Summary 279

Design Patterns & Best Practices **280**

What We Learned the Hard Way 280

1. Memory Management 280

The try/finally Pattern 280

Parsing JSON Safely 280

Component Ownership 281

MemTable Lifecycle 281

2. Async Patterns 281

The Problem with Synchronous Calls 281

The Async Solution 282

Loading Indicators 282

Multiple Concurrent Requests 282

Cancellation and Timeouts 283

3. Error Handling Patterns 283

Status Code Checking 283

JSON Parse Failure 284

Network Error Handling 284

User-Friendly Error Messages in HTML . . . 285
4. Design-Time vs Runtime . . . 286
What to Configure in Object Inspector . . . 286
What to Configure in Code . . . 286
When to Create Components Dynamically . . . 286
5. Data Flow Patterns . . . 287
Unidirectional: API to MemTable to UI . . . 287
Bidirectional: UI to API to MemTable to UI . . . 288
Event-Driven: WebSocket to UI . . . 288
6. Performance . . . 289
MemTable SyncMode for Incremental Updates . . . 289
Minimizing HTML Re-renders . . . 289
Image Caching . . . 289
Lazy Loading with Pagination . . . 290
7. Security . . . 290
Never Hardcode Credentials . . . 290
Token Refresh Pattern . . . 290
HTTPS Only . . . 291
8. Project Organization . . . 291
Separate Data Modules . . . 291
Template File Organization . . . 292
Resource Management . . . 292
9. Complete Example: Refactoring a Poorly Written App . . . 293
10. Exercise: Code Review . . . 295
Solution . . . 296
Summary . . . 297

When Things Go Wrong . . . 298
1. SSL Errors . . . 298
"Could not load SSL library" . . . 298
Certificate Verification Failures . . . 299
Self-Signed Certificate Handling . . . 299
2. REST Issues . . . 299
401 Unauthorized . . . 299
404 Not Found . . . 300
500 Internal Server Error . . . 300
Timeout Errors . . . 301
CORS Issues When Calling Web APIs . . . 301
3. JSON Issues . . . 301
Access Violation Parsing Malformed JSON . . . 301
Field Names Not Matching . . . 302
Nested Objects Becoming ftMemo . . . 302
Large JSON Causing Out of Memory . . . 302
4. MemTable Issues . . . 303
"Field not found" . . . 303
Duplicate Key Violations in Sync Mode . . . 303
IndexFieldNames Not Set for Sync Mode . . . 304
Data Type Mismatches . . . 304
5. HTML Renderer Issues . . . 304
Elements Not Displaying . . . 304
CSS Not Applying . . . 304
Form Controls Not Appearing . . . 305
onclick Not Firing . . . 305
Images Not Loading . . . 306

6. Page Navigation Issues . . . 306
Default Page Not Showing . . . 306
Links Not Navigating . . . 306
OnBeforeNavigate Not Cancelling . . . 307
7. Twig Template Issues . . . 307
Template Not Rendering . . . 307
Variables Empty . . . 307
Includes Failing . . . 308
8. WebSocket Issues . . . 308
Connection Refused . . . 308
Messages Not Receiving . . . 308
Auto-Reconnect Not Working . . . 309
9. Diagnostic Checklists . . . 309
TTina4REST Checklist . . . 309
TTina4HTMLRender Checklist . . . 309
TTina4HTMLPages Checklist . . . 310
TTina4RESTRequest Checklist . . . 310
10. Quick Reference: Error to Fix . . . 311
Summary . . . 311

Getting Started

Your First 10 Minutes

A Delphi IDE. Two packages installed. One form. In ten minutes you will have a running FMX application that fetches live data from a REST API and displays it in a grid. The data will appear before you understand the plumbing. That is the point -- you ship first, then you learn.

---

1. What Is Tina4 Delphi

Tina4 Delphi is a design-time component library for Delphi 10.4+ (FireMonkey / FMX). Nine components, each solving one problem:

- **TTina4REST** for REST client configuration -- base URL, auth, headers
- **TTina4RESTRequest** for declarative REST calls -- link an endpoint, a MemTable, and execute
- **TTina4JSONAdapter** for static JSON to MemTable binding -- no HTTP required
- **TTina4HTMLRender** for rendering HTML with CSS on an FMX canvas -- forms, tables, images, events
- **TTina4HTMLPages** for SPA-style page navigation inside your desktop app
- **TTina4WebSocketClient** for real-time WebSocket communication with auto-reconnect and ping/pong keepalive
- **TTina4SocketServer** for raw TCP socket server functionality
- **TTina4WebServer** for hosting an embedded HTTP web server
- **TTina4Route** for declarative URL routing

Plus a core utility unit (`Tina4Core.pas`) with standalone functions for HTTP, JSON, database, encoding, and shell commands. There is also `TTina4Twig`, a plain `TObject` class (not a design-time component) for Twig-compatible template rendering.

What It Is Not

Tina4 Delphi is not a framework. It does not take over your application. It does not impose an architecture. It does not require you to restructure your project. Drop components on a form. Set properties. Call methods. Your existing FireDAC connections, your existing business logic, your existing UI -- everything stays exactly where it is.

Tina4 Delphi is not VCL. It is FireMonkey only. If you need VCL support, you can still use `Tina4Core.pas` directly -- the utility functions have no FMX dependency.

---

2. Prerequisites

You need three things. Nothing else.

- **Delphi 10.4 or later** -- any edition that includes FireMonkey and FireDAC.
- **OpenSSL DLLs** -- required for HTTPS. Without them, every REST call to an HTTPS endpoint will fail silently or raise an exception.
- **The Tina4 Delphi source** -- cloned from GitHub.

---

3. Installation

Step 1: Clone the Repository

```
git clone https://github.com/tina4stack/tina4delphi.git
```

Step 2: Open the Project Group

In the Delphi IDE, open the **Tina4DelphiProject** project group file. You will see two projects:

- **Tina4Delphi** -- the runtime package
- **Tina4DelphiDesign** -- the design-time package

Step 3: Build and Install the Runtime Package

Right-click **Tina4Delphi** in the Project Manager and select **Build**. This compiles the runtime units but does not register anything in the IDE.

Step 4: Build and Install the Design-Time Package

Right-click **Tina4DelphiDesign** and select **Build**, then **Install**. You should see a confirmation dialog:

> *Package Tina4DelphiDesign has been installed. The following new component(s) have been registered: TTina4REST, TTina4RESTRequest, TTina4JSONAdapter, TTina4HTMLRender, TTina4HTMLPages, TTina4WebSocketClient, TTina4SocketServer, TTina4WebServer, TTina4Route.*

Step 5: Verify

Open the Tool Palette. Search for "Tina4". All nine components should appear under the **Tina4Delphi** category. If they do not, check that the output directories for both packages are on your IDE's library path.

---

4. SSL Setup

HTTPS calls require OpenSSL DLLs. Delphi's `TNetHTTPClient` (which Tina4 uses internally) will fail without them. The error is often cryptic -- an empty response, a 0 status code, or an access violation.

Windows

Download the OpenSSL binaries for your Delphi version (typically OpenSSL 1.1.x for Delphi 10.4/11, or OpenSSL 3.x for Delphi 12+). You need two sets:

- **32-bit DLLs** (`libssl-1_1.dll`, `libcrypto-1_1.dll`) -- copy to `C:\Windows\SysWOW64\`. The IDE runs as a 32-bit process and needs these to make HTTPS calls at design time and during debugging.
- **64-bit DLLs** (`libssl-1_1-x64.dll`, `libcrypto-1_1-x64.dll`) -- copy to `C:\Windows\System32\`. Your compiled 64-bit application uses these at runtime.

Quick Test

After placing the DLLs, create a blank FMX project and add this to a button click:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  StatusCode: Integer;
  Response: TBytes;
begin
  Response := SendHttpRequest(StatusCode, 'https://jsonplaceholder.typicode.com', '/posts/1');
  ShowMessage('Status: ' + StatusCode.ToString + ' / ' + TEncoding.UTF8.GetString(Response));
end;
```

Add `Tina4Core` to your uses clause. If you see a JSON response with a status of 200, SSL is working. If you get a status of 0 or an exception, your DLLs are missing or the wrong bitness.

---

5. Available Components

Here is the full inventory. Each gets its own chapter, but knowing the landscape helps you plan.

Component	What It Does	Typical Use Case
`TTina4REST`	Holds base URL, credentials, bearer token	One per API endpoint
`TTina4RESTRequest`	Executes a REST call, populates a MemTable	One per endpoint/action
`TTina4JSONAdapter`	Binds static JSON to a MemTable	Offline data, config files
`TTina4HTMLRender`	Renders HTML + CSS on an FMX canvas	Reports, dashboards, forms
`TTina4HTMLPages`	SPA navigation between pages	Multi-page desktop apps
`TTina4WebSocketClient`	WebSocket client with auto-reconnect and keepalive	Real-time data feeds, chat
`TTina4SocketServer`	Raw TCP socket server	Custom protocol servers

`TTina4WebServer`	Embedded HTTP web server	Local dashboards, APIs
`TTina4Route`	Declarative URL routing	REST endpoint definitions

Additionally, `TTina4Twig` is a plain `TObject` class (not a design-time component) that provides a Twig-compatible template engine for dynamic HTML generation.

And the standalone utility functions in `Tina4Core.pas`:

Function	What It Does
`SendHttpRequest`	Low-level HTTP with auth, headers, timeouts
`BytesToJSONObject`	Parse raw HTTP response bytes to `TJSONObject`
`GetJSONFromDB`	SQL query to JSON with camelCase, ISO dates, Base64 blobs
`PopulateMemTableFromJSON`	JSON to MemTable with Clear or Sync mode
`SendMultipartFormData`	File upload with form fields
`CamelCase` / `SnakeCase`	Name conversion between database and JSON
`GetGUID`	Generate a GUID string
`ExecuteShellCommand`	Run a shell command and capture output

---

6. Your First App: API Data in a Grid

Time to build something real. You will fetch a list of posts from a public API and display them in a grid. No dummy data. No mocking. Live HTTP on your first try.

Step 1: Create the Project

File > New > Multi-Device Application > Blank Application. Save it as `FirstTina4App`.

Step 2: Drop Components on the Form

From the Tool Palette, add these components:

- **TTina4REST** -- name it `Tina4REST1`
- **TTina4RESTRequest** -- name it `Tina4RESTRequest1`
- **TFDMemTable** -- name it `FDMemTable1` (from the FireDAC palette)
- **TStringGrid** -- name it `StringGrid1` (from the Grids palette)
- **TButton** -- name it `btnFetch`, set `Text` to `Fetch Posts`

Step 3: Configure the REST Client

Select `Tina4REST1` and set these properties in the Object Inspector:

Property	Value
`BaseUrl`	`https://jsonplaceholder.typicode.com`

No username, no password, no bearer token. This is a public API.

Step 4: Configure the REST Request

Select `Tina4RESTRequest1` and set:

Property	Value
`Tina4REST`	`Tina4REST1`
`EndPoint`	`/posts`
`RequestType`	`Get`
`MemTable`	`FDMemTable1`
`SyncMode`	`Clear`

The `DataKey` property can be left empty. When the API returns a JSON array at the root level (as jsonplaceholder does), the component handles it automatically.

Step 5: Wire the Button

Double-click `btnFetch` and add:

```
procedure TForm1.btnFetchClick(Sender: TObject);
begin
  Tina4RESTRequest1.ExecuteRESTCall;

  // Populate the grid from the MemTable
  StringGrid1.RowCount := FDMemTable1.RecordCount;

  // Clear existing columns and create from field definitions
  StringGrid1.ClearColumns;
  for var I := 0 to FDMemTable1.FieldCount - 1 do
  begin
    var Col := TStringColumn.Create(StringGrid1);
    Col.Header := FDMemTable1.Fields[I].FieldName;
    StringGrid1.AddObject(Col);
  end;

  // Populate rows
  FDMemTable1.First;
  var Row := 0;
  while not FDMemTable1.Eof do
  begin
    for var C := 0 to FDMemTable1.FieldCount - 1 do
      StringGrid1.Cells[C, Row] := FDMemTable1.Fields[C].AsString;
```

```
    Inc(Row);
    FDMemTable1.Next;
  end;
end;
```

Add `Tina4REST, Tina4RESTRequest, FMX.Grid.Style` to your uses clause.

Step 6: Run

Press **F9**. Click **Fetch Posts**. The grid fills with 100 posts from the API -- id, userId, title, and body columns. If it does not work, check the SSL setup from Section 4.

What Just Happened

One component configured the base URL. Another component made the HTTP call, parsed the JSON response, created field definitions from the JSON structure, and populated a MemTable. You wrote zero HTTP code. Zero JSON parsing code. Zero field-definition code. The component chain handled all of it.

---

7. Quick Wins with Tina4Core

You do not always need components. `Tina4Core.pas` gives you standalone functions you can call from anywhere. Here are one-liners that solve common problems:

Fetch JSON from an API

```
uses Tina4Core;

var
  StatusCode: Integer;
  Response: TBytes;
  JSON: TJSONObject;
begin
  Response := SendHttpRequest(StatusCode, 'https://api.example.com', '/users');
  JSON := BytesToJSONObject(Response);
  try
    ShowMessage(JSON.ToString);
  finally
    JSON.Free;
  end;
end;
```

Database Query to JSON

```
var JSON := GetJSONFromDB(FDConnection1, 'SELECT * FROM customers WHERE active = 1');
try
  Memo1.Lines.Text := JSON.Format;
finally
  JSON.Free;
end;
// Output: {"records": [{"id": "1", "firstName": "Alice", ...}, ...]}
// Note: field names auto-convert from snake_case to camelCase
```

JSON to MemTable

```
var JSONStr := '{"users": [{"id": 1, "name": "Alice"}, {"id": 2, "name": "Bob"}]}';
PopulateMemTableFromJSON(FDMemTable1, 'users', JSONStr);
// FDMemTable1 now has 2 rows with id and name columns
```

Upload a File

```
var
  StatusCode: Integer;
begin
  SendMultipartFormData(
    StatusCode,
    'https://api.example.com',
    'upload/document',
    ['userId', '42', 'description', 'Q4 Report'],  // form fields
    ['file', 'C:\reports\q4.pdf'],                   // file field
    '', 'admin', 'secret');                           // auth
end;
```

Convert Between Naming Conventions

```
CamelCase('first_name');    // 'firstName'
CamelCase('user_email');    // 'userEmail'
SnakeCase('firstName');     // 'first_name'
SnakeCase('userEmail');     // 'user_email'
```

---

8. Exercise: Build a Weather Dashboard

Build an FMX application that fetches weather data from a public API and displays it in a MemTable-backed grid.

Requirements

- Use `TTina4REST` configured with `https://api.open-meteo.com` (no API key needed)
- Use `TTina4RESTRequest` to call `/v1/forecast?latitude=52.52&longitude=13.41&hourly=temperature_2m`
- Display the hourly temperatures in a `TStringGrid`
- Add a `TEdit` for latitude and a `TEdit` for longitude so the user can change the location
- Add a "Refresh" button that re-fetches with the new coordinates

Hints

- The Open-Meteo API returns JSON with nested structure. The hourly data is under the `hourly` key.
- Set `DataKey` to `hourly` on the `TTina4RESTRequest` -- but note this API returns parallel arrays (`time` and `temperature_2m`), not an array of objects. You may need to use `Tina4Core.SendHttpRequest` directly and parse manually.
- Use `BytesToJSONObject` to parse the response, then extract the arrays yourself.

Solution

```
unit WeatherForm;

interface

uses
  System.SysUtils, System.Types, System.Classes, System.JSON,
  FMX.Types, FMX.Controls, FMX.Forms, FMX.StdCtrls, FMX.Edit,
  FMX.Grid, FMX.Grid.Style, FMX.ScrollBox,
  FireDAC.Comp.Client,
  Tina4Core;

type
  TfrmWeather = class(TForm)
    edtLatitude: TEdit;
    edtLongitude: TEdit;
    btnRefresh: TButton;
    StringGrid1: TStringGrid;
    lblLatitude: TLabel;
    lblLongitude: TLabel;
    procedure FormCreate(Sender: TObject);
    procedure btnRefreshClick(Sender: TObject);
  private
    FMemTable: TFDMemTable;
    procedure FetchWeather;
  end;

var
  frmWeather: TfrmWeather;

implementation

{$R *.fmx}

procedure TfrmWeather.FormCreate(Sender: TObject);
begin
  edtLatitude.Text := '52.52';
  edtLongitude.Text := '13.41';

  FMemTable := TFDMemTable.Create(Self);
  FMemTable.FieldDefs.Add('Time', ftString, 25);
  FMemTable.FieldDefs.Add('Temperature', ftFloat);
  FMemTable.CreateDataSet;
end;

procedure TfrmWeather.btnRefreshClick(Sender: TObject);
begin
  FetchWeather;
end;

procedure TfrmWeather.FetchWeather;
var
  StatusCode: Integer;
  Response: TBytes;
  JSON: TJSONObject;
  Times, Temps: TJSONArray;
  I: Integer;
begin
```

```
Response := SendHttpRequest(StatusCode,
  'https://api.open-meteo.com',
  '/v1/forecast',
  Format('latitude=%s&longitude=%s&hourly=temperature_2m',
    [edtLatitude.Text, edtLongitude.Text]));

if StatusCode <> 200 then
begin
  ShowMessage('API returned status: ' + StatusCode.ToString);
  Exit;
end;

JSON := BytesToJSONObject(Response);
try
  if not Assigned(JSON) then
  begin
    ShowMessage('Invalid JSON response');
    Exit;
  end;

  var Hourly := JSON.GetValue<TJSONObject>('hourly');
  Times := Hourly.GetValue<TJSONArray>('time');
  Temps := Hourly.GetValue<TJSONArray>('temperature_2m');

  FMemTable.EmptyDataSet;
  for I := 0 to Times.Count - 1 do
  begin
    FMemTable.Append;
    FMemTable.FieldByName('Time').AsString := Times.Items[I].Value;
    FMemTable.FieldByName('Temperature').AsFloat := Temps.Items[I].AsType<Double>;
    FMemTable.Post;
  end;

  // Populate grid
  StringGrid1.RowCount := FMemTable.RecordCount;
  StringGrid1.ClearColumns;

  var ColTime := TStringColumn.Create(StringGrid1);
  ColTime.Header := 'Time';
  ColTime.Width := 200;
  StringGrid1.AddObject(ColTime);

  var ColTemp := TStringColumn.Create(StringGrid1);
  ColTemp.Header := 'Temperature (C)';
  ColTemp.Width := 150;
  StringGrid1.AddObject(ColTemp);

  FMemTable.First;
  var Row := 0;
  while not FMemTable.Eof do
  begin
    StringGrid1.Cells[0, Row] := FMemTable.FieldByName('Time').AsString;
    StringGrid1.Cells[1, Row] := FormatFloat('0.0', FMemTable.FieldByName('Temperature').AsFlo
    Inc(Row);
    FMemTable.Next;
  end;
finally
  JSON.Free;
end;
```

```
end;

end.
```

---

9. Common Gotchas

SSL DLLs Missing

Symptom: Status code 0, empty response, or `ENetHTTPClientException`.

Fix: Place the correct OpenSSL DLLs in the right directories. 32-bit in `SysWOW64` (for the IDE), 64-bit in `System32` (for your compiled app). Check the DLL version matches your Delphi version.

Wrong DLL Bitness

Symptom: Works in the IDE (32-bit debugger) but fails in a 64-bit release build, or vice versa.

Fix: You need both sets. The IDE is 32-bit. Your release build is (usually) 64-bit. Both paths need the correct DLLs.

Design-Time Package Not Installed

Symptom: Components do not appear in the Tool Palette.

Fix: Build the runtime package first, then install the design-time package. The design-time package depends on the runtime package. If you skip the runtime build, the install will fail silently or with cryptic linker errors.

Library Path Missing

Symptom: Compiling your project gives "File not found" errors for Tina4 units.

Fix: Add the Tina4 source directory to your project's search path, or to the IDE's global library path under **Tools > Options > Delphi Options > Library > Library Path**.

TJSONObject Memory Leaks

Symptom: Growing memory usage over time.

Fix: Every `TJSONObject` returned by `Get`, `Post`, `BytesToJSONObject`, `GetJSONFromDB`, etc. must be freed by the caller. Always use `try..finally` blocks. Delphi does not have garbage collection.

---

10. What Just Happened

Ten minutes. Two packages installed. One form built. And you covered:

- Installing the Tina4 component library
- Setting up SSL for HTTPS
- Configuring a REST client with `TTina4REST`

- Fetching data with `TTina4RESTRequest`
- Automatic JSON-to-MemTable population
- Standalone utility functions from `Tina4Core`
- A complete working exercise with solution

The rest of this book goes deep on each component. But you already have a working app. You already have data flowing from an API to your UI. Everything from here is precision and power.

Summary

What	How
Install runtime	Build **Tina4Delphi** package
Install design-time	Build and install **Tina4DelphiDesign** package
SSL (IDE, 32-bit)	Copy 32-bit DLLs to `SysWOW64`
SSL (app, 64-bit)	Copy 64-bit DLLs to `System32`
REST base config	`TTina4REST` -- set `BaseUrl`, auth
Fetch + populate	`TTina4RESTRequest` -- set endpoint, MemTable, call `ExecuteRESTCall`
Raw HTTP	`SendHttpRequest(StatusCode, BaseUrl, Endpoint)`
Parse response	`BytesToJSONObject(ResponseBytes)`
DB to JSON	`GetJSONFromDB(Connection, SQL)`
JSON to MemTable	`PopulateMemTableFromJSON(MemTable, Key, JSON)`
File upload	`SendMultipartFormData(...)`
Name conversion	`CamelCase(snake)` / `SnakeCase(camel)`

REST APIs

Two Ways to Talk to the Outside World

Your application needs data from somewhere. A customer database behind an API. A payment gateway. A weather service. A machine learning endpoint. Between your Delphi form and that data sits HTTP -- and two very different ways to make the call.

The first way is component-based. Drop `TTina4REST` and `TTina4RESTRequest` on your form. Set properties. Execute. The MemTable fills. You write no HTTP code, no JSON parsing, no threading boilerplate.

The second way is direct. Call `Tina4REST1.Get()` or `Tina4REST1.Post()` and get a `TJSONObject` back. Full control. Full responsibility -- including freeing the object.

This chapter covers both.

---

1. TTina4REST -- Base Configuration

Every REST call needs a server. `TTina4REST` holds that configuration so you set it once and every `TTina4RESTRequest` linked to it inherits the connection details.

Design-Time Setup

Drop a `TTina4REST` on your form. In the Object Inspector:

Property	Description	Example
`BaseUrl`	The root URL for all endpoints	`https://api.example.com/v1`
`Username`	HTTP Basic Auth username	`admin`
`Password`	HTTP Basic Auth password	`secret`

Runtime Configuration

```
Tina4REST1.BaseUrl := 'https://api.example.com/v1';
Tina4REST1.Username := 'admin';
Tina4REST1.Password := 'secret';
```

Bearer Token Authentication

Most modern APIs use Bearer tokens instead of Basic Auth. Call `SetBearer` after obtaining your token:

```
Tina4REST1.SetBearer('eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...');
```

This adds an `Authorization: Bearer <token>` header to every request made through this component. If you also set `Username` and `Password`, Basic Auth is used instead -- Bearer and Basic Auth are mutually exclusive.

One Component Per API

If your application talks to multiple APIs, use multiple `TTina4REST` components:

```
// API 1: Your backend
Tina4RESTBackend.BaseUrl := 'https://api.myapp.com/v1';
Tina4RESTBackend.SetBearer(AuthToken);

// API 2: Payment gateway
Tina4RESTPayments.BaseUrl := 'https://payments.stripe.com';
Tina4RESTPayments.SetBearer(StripeKey);

// API 3: Public data
Tina4RESTPublic.BaseUrl := 'https://api.open-meteo.com';
// No auth needed
```

---

2. Direct REST Calls

When you need full control over the request and response, call methods directly on `TTina4REST`. All five HTTP methods are supported. All return a `TJSONObject`. All require you to free the result.

GET

```
var
  StatusCode: Integer;
  Response: TJSONObject;
begin
  Response := Tina4REST1.Get(StatusCode, '/users', 'page=1&limit=10');
  try
    if StatusCode = 200 then
      Memo1.Lines.Text := Response.Format
    else
      ShowMessage('Error: ' + StatusCode.ToString);
  finally
    Response.Free;
  end;
end;
```

The three parameters are: `StatusCode` (out), `EndPoint`, and `QueryParams`. The endpoint is appended to the `BaseUrl`. Query params are appended after a `?`.

POST

```
var
  StatusCode: Integer;
  Response: TJSONObject;
begin
  Response := Tina4REST1.Post(StatusCode, '/users', '',
    '{"name": "Alice", "email": "alice@example.com"}');
  try
    if StatusCode = 201 then
      ShowMessage('User created: ' + Response.GetValue<String>('id'))
    else
      ShowMessage('Failed: ' + Response.ToString);
  finally
    Response.Free;
  end;
end;
```

PATCH (Partial Update)

```
var
  StatusCode: Integer;
  Response: TJSONObject;
begin
  Response := Tina4REST1.Patch(StatusCode, '/users/42', '',
    '{"role": "admin"}');
  try
    if StatusCode = 200 then
      ShowMessage('User updated')
    else
      ShowMessage('Failed: ' + StatusCode.ToString);
  finally
    Response.Free;
  end;
end;
```

PUT (Full Replace)

```
var
  StatusCode: Integer;
  Response: TJSONObject;
begin
  Response := Tina4REST1.Put(StatusCode, '/users/42', '',
    '{"name": "Alice", "email": "alice@new.com", "role": "admin"}');
  try
    // handle response
  finally
    Response.Free;
  end;
end;
```

DELETE

```
var
  StatusCode: Integer;
  Response: TJSONObject;
begin
  Response := Tina4REST1.Delete(StatusCode, '/users/42');
  try
    if StatusCode = 204 then
      ShowMessage('Deleted')
    else
      ShowMessage('Failed: ' + StatusCode.ToString);
  finally
    Response.Free;
  end;
end;
```

Method Reference

Method	Signature	HTTP Verb
`Get`	`Get(var StatusCode: Integer; EndPoint: string; QueryParams: string = ''): TJSONObject`	GET
`Post`	`Post(var StatusCode: Integer; EndPoint: string; QueryParams: string = ''; Body: string = ''): TJSONObject`	POST
`Patch`	`Patch(var StatusCode: Integer; EndPoint: string; QueryParams: string = ''; Body: string = ''): TJSONObject`	PATCH
`Put`	`Put(var StatusCode: Integer; EndPoint: string; QueryParams: string = ''; Body: string = ''): TJSONObject`	PUT
`Delete`	`Delete(var StatusCode: Integer; EndPoint: string; QueryParams: string = ''): TJSONObject`	DELETE

---

3. Authentication Patterns

Basic Auth

Set `Username` and `Password` on `TTina4REST`. Every request includes an `Authorization: Basic` header automatically:

```
Tina4REST1.BaseUrl := 'https://api.example.com';
Tina4REST1.Username := 'apiuser';
Tina4REST1.Password := 'apipassword';
```

Bearer Token (Static)

If you have a long-lived API key or token:

```
Tina4REST1.BaseUrl := 'https://api.example.com';
Tina4REST1.SetBearer('your-api-key-here');
```

Bearer Token (Login Flow)

Most apps require a login step that returns a short-lived token:

```
procedure TForm1.Login;
var
  StatusCode: Integer;
  Response: TJSONObject;
begin
  // Use a temporary REST component with no auth for the login call
  Tina4REST1.BaseUrl := 'https://api.example.com';

  Response := Tina4REST1.Post(StatusCode, '/auth/login', '',
    Format('{"email": "%s", "password": "%s"}',
      [edtEmail.Text, edtPassword.Text]));
  try
    if StatusCode = 200 then
    begin
      var Token := Response.GetValue<String>('token');
      Tina4REST1.SetBearer(Token);
      ShowMessage('Logged in successfully');
    end
    else
      ShowMessage('Login failed: ' + Response.GetValue<String>('message'));
  finally
    Response.Free;
  end;
end;
```

Custom Headers

For APIs that require custom headers (API keys in headers, tenant IDs, etc.), use `SendHttpRequest` from `Tina4Core` directly:

```
uses Tina4Core;

var
  StatusCode: Integer;
  Headers: TNetHeaders;
  Response: TBytes;
begin
  SetLength(Headers, 2);
  Headers[0] := TNameValuePair.Create('X-API-Key', 'my-key-123');
  Headers[1] := TNameValuePair.Create('X-Tenant-Id', 'acme-corp');

  Response := SendHttpRequest(StatusCode,
    'https://api.example.com', '/data', '', '',
    'application/json', 'utf-8', '', '', Headers);
end;
```

---

4. TTina4RESTRequest -- Declarative REST

Direct calls give you control. `TTina4RESTRequest` gives you convenience. Link it to a `TTina4REST`, set properties, and execute. The component handles the HTTP call, parses the JSON response, creates MemTable field definitions, and populates the table. One method call.

Basic GET with Auto MemTable Population

Drop these on your form:

- `TTina4REST` (name: `Tina4REST1`)
- `TTina4RESTRequest` (name: `Tina4RESTRequest1`)
- `TFDMemTable` (name: `FDMemTable1`)

Configure `Tina4RESTRequest1`:

Property	Value
`Tina4REST`	`Tina4REST1`
`EndPoint`	`/users`
`RequestType`	`Get`
`DataKey`	`records`
`MemTable`	`FDMemTable1`
`SyncMode`	`Clear`

Execute:

```
Tina4RESTRequest1.ExecuteRESTCall;
// FDMemTable1 now contains all users from the "records" array
```

The `DataKey` tells the component which JSON key contains the array of records. If your API returns `{"records": [...]}`, set `DataKey` to `records`. If the response is a bare JSON array `[...]`, leave `DataKey` empty.

POST with RequestBody

```
Tina4RESTRequest1.RequestType := TTina4RequestType.Post;
Tina4RESTRequest1.EndPoint := '/users';
Tina4RESTRequest1.RequestBody.Text :=
  '{"name": "Alice", "email": "alice@example.com", "role": "editor"}';
Tina4RESTRequest1.ExecuteRESTCall;
```

The `RequestBody` is a `TStringList`. Set it with `.Text` for single-line JSON, or use `.Add` for multiline construction.

PUT / PATCH / DELETE

Change the `RequestType` property:

```
// Update
Tina4RESTRequest1.RequestType := TTina4RequestType.Put;
```

```
Tina4RESTRequest1.EndPoint := '/users/42';
Tina4RESTRequest1.RequestBody.Text := '{"name": "Alice Updated"}';
Tina4RESTRequest1.ExecuteRESTCall;

// Delete
Tina4RESTRequest1.RequestType := TTina4RequestType.Delete;
Tina4RESTRequest1.EndPoint := '/users/42';
Tina4RESTRequest1.ExecuteRESTCall;
```

---

5. Master/Detail with Parameter Injection

This is where `TTina4RESTRequest` earns its keep. Set a `MasterSource` and the detail request injects field values from the master's MemTable into the endpoint, request body, and query params using `{fieldName}` placeholders.

Setup

```
// Master: fetches all customers
Tina4RESTRequest1.Tina4REST := Tina4REST1;
Tina4RESTRequest1.EndPoint := '/customers';
Tina4RESTRequest1.DataKey := 'records';
Tina4RESTRequest1.MemTable := FDMemTableCustomers;
Tina4RESTRequest1.RequestType := TTina4RequestType.Get;

// Detail: fetches orders for the selected customer
Tina4RESTRequest2.Tina4REST := Tina4REST1;
Tina4RESTRequest2.MasterSource := Tina4RESTRequest1;
Tina4RESTRequest2.EndPoint := '/customers/{id}/orders';
Tina4RESTRequest2.DataKey := 'records';
Tina4RESTRequest2.MemTable := FDMemTableOrders;
Tina4RESTRequest2.RequestType := TTina4RequestType.Get;
```

When the master executes and the user navigates to a customer with `id = 5`, the detail's endpoint becomes `/customers/5/orders`. The `{id}` placeholder is replaced with the current value of the `id` field from `FDMemTableCustomers`.

How It Works

- The master request executes and populates `FDMemTableCustomers`.
- When you scroll to a different row in `FDMemTableCustomers`, the detail request re-executes automatically.
- The detail request scans its `EndPoint`, `RequestBody`, and `QueryParams` for `{fieldName}` patterns.
- Each pattern is replaced with the current field value from the master's MemTable.

Multiple Placeholders

You can use multiple placeholders:

```
Tina4RESTRequest2.EndPoint := '/customers/{customerId}/orders';
Tina4RESTRequest2.RequestBody.Text :=
  '{"customerId": "{customerId}", "status": "active"}';
```

---

6. POST from SourceMemTable

Sometimes you need to send data that already exists in a MemTable -- an import batch, a modified dataset, user edits. Instead of manually serializing rows to JSON, link a `SourceMemTable`:

```
Tina4RESTRequest1.RequestType := TTina4RequestType.Post;
Tina4RESTRequest1.EndPoint := '/import/products';
Tina4RESTRequest1.SourceMemTable := FDMemTableProducts;
Tina4RESTRequest1.SourceIgnoreFields := 'internal_id,temp_flag';
Tina4RESTRequest1.SourceIgnoreBlanks := True;
Tina4RESTRequest1.ExecuteRESTCall;
```

The component serializes all rows from `FDMemTableProducts` to a JSON array and sends it as the POST body. Fields listed in `SourceIgnoreFields` are excluded. If `SourceIgnoreBlanks` is `True`, fields with empty values are omitted from each row.

---

7. Async Execution

REST calls block the main thread. For a quick local API, that is fine. For a slow endpoint or a large response, your UI freezes. `ExecuteRESTCallAsync` runs the request in a background thread.

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  Tina4RESTRequest1.OnExecuteDone := HandleRequestDone;
end;

procedure TForm1.HandleRequestDone(Sender: TObject);
begin
  TThread.Synchronize(nil, procedure
  begin
    ShowMessage('Loaded ' + FDMemTable1.RecordCount.ToString + ' records');
    // Update your grid or UI here -- you are now on the main thread
  end);
end;

procedure TForm1.btnFetchClick(Sender: TObject);
begin
  btnFetch.Enabled := False;
  Tina4RESTRequest1.ExecuteRESTCallAsync;
end;
```

Thread Safety Rules

- **Never access UI controls from the background thread.** The `OnExecuteDone` event fires on the background thread. Wrap all UI updates in `TThread.Synchronize`.
- **The MemTable is populated before `OnExecuteDone` fires.** You can read the MemTable inside the synchronized block.
- **Disable buttons while the request is in flight.** Re-enable them in `OnExecuteDone`.

---

8. Events

OnExecuteDone

Fires after the REST call completes and the MemTable is populated (if configured). Use it for post-processing, UI updates, or chaining requests:

```
procedure TForm1.Request1ExecuteDone(Sender: TObject);
begin
  TThread.Synchronize(nil, procedure
  begin
    lblCount.Text := Format('%d records loaded', [FDMemTable1.RecordCount]);

    // Chain: now fetch details for the first record
    if FDMemTable1.RecordCount > 0 then
    begin
      FDMemTable1.First;
      Tina4RESTRequest2.ExecuteRESTCall;
    end;
  end);
end;
```

OnAddRecord

Fires for each record added to the MemTable during population. Use it for custom field transformations, filtering, or logging:

```
procedure TForm1.Request1AddRecord(Sender: TObject);
begin
  // Access the MemTable -- the cursor is on the newly added record
  var Status := FDMemTable1.FieldByName('status').AsString;
  if Status = 'inactive' then
    FDMemTable1.Delete;  // Remove inactive records during import
end;
```

---

9. Complete Example: Customer Management Panel

A real-world scenario. List customers. View details. Create new ones. Update existing ones. Four operations, four REST calls, one form.

Form Design

- `TTina4REST` (name: `restAPI`, BaseUrl: `https://api.example.com/v1`)
- `TTina4RESTRequest` (name: `reqListCustomers`)
- `TTina4RESTRequest` (name: `reqCreateCustomer`)
- `TTina4RESTRequest` (name: `reqUpdateCustomer`)
- `TFDMemTable` (name: `mtCustomers`)
- `TStringGrid` (name: `gridCustomers`)
- `TEdit` (name: `edtName`)
- `TEdit` (name: `edtEmail`)

- `TButton` (name: `btnLoad`, Text: `Load`)
- `TButton` (name: `btnSave`, Text: `Save`)
- `TLabel` (name: `lblStatus`)

Implementation

```
unit CustomerPanel;

interface

uses
  System.SysUtils, System.Classes, System.JSON,
  FMX.Types, FMX.Controls, FMX.Forms, FMX.StdCtrls, FMX.Edit,
  FMX.Grid, FMX.Grid.Style, FMX.ScrollBox, FMX.Layouts,
  FireDAC.Comp.Client,
  Tina4REST, Tina4RESTRequest;

type
  TfrmCustomers = class(TForm)
    restAPI: TTina4REST;
    reqListCustomers: TTina4RESTRequest;
    reqCreateCustomer: TTina4RESTRequest;
    reqUpdateCustomer: TTina4RESTRequest;
    mtCustomers: TFDMemTable;
    gridCustomers: TStringGrid;
    edtName: TEdit;
    edtEmail: TEdit;
    btnLoad: TButton;
    btnSave: TButton;
    lblStatus: TLabel;
    procedure FormCreate(Sender: TObject);
    procedure btnLoadClick(Sender: TObject);
    procedure btnSaveClick(Sender: TObject);
    procedure gridCustomersSelectCell(Sender: TObject; const ACol, ARow: Integer;
      var CanSelect: Boolean);
  private
    FSelectedId: string;
    procedure SetupRequests;
    procedure RefreshGrid;
    procedure SetStatus(const Msg: string);
  end;

var
  frmCustomers: TfrmCustomers;

implementation

{$R *.fmx}

procedure TfrmCustomers.FormCreate(Sender: TObject);
begin
  restAPI.BaseUrl := 'https://api.example.com/v1';
  restAPI.SetBearer('your-token-here');
  FSelectedId := '';
  SetupRequests;
end;

procedure TfrmCustomers.SetupRequests;
begin
  // List customers
  reqListCustomers.Tina4REST := restAPI;
  reqListCustomers.EndPoint := '/customers';
  reqListCustomers.RequestType := TTina4RequestType.Get;
```

```
  reqListCustomers.DataKey := 'records';
  reqListCustomers.MemTable := mtCustomers;
  reqListCustomers.SyncMode := TTina4RestSyncMode.Clear;

  // Create customer
  reqCreateCustomer.Tina4REST := restAPI;
  reqCreateCustomer.EndPoint := '/customers';
  reqCreateCustomer.RequestType := TTina4RequestType.Post;

  // Update customer
  reqUpdateCustomer.Tina4REST := restAPI;
  reqUpdateCustomer.RequestType := TTina4RequestType.Put;
end;

procedure TfrmCustomers.btnLoadClick(Sender: TObject);
begin
  reqListCustomers.ExecuteRESTCall;
  RefreshGrid;
  SetStatus(Format('Loaded %d customers', [mtCustomers.RecordCount]));
end;

procedure TfrmCustomers.btnSaveClick(Sender: TObject);
var
  Body: string;
  StatusCode: Integer;
  Response: TJSONObject;
begin
  Body := Format('{"name": "%s", "email": "%s"}',
    [edtName.Text, edtEmail.Text]);

  if FSelectedId <> '' then
  begin
    // Update existing customer
    Response := restAPI.Put(StatusCode,
      '/customers/' + FSelectedId, '', Body);
    try
      if StatusCode = 200 then
        SetStatus('Customer updated')
      else
        SetStatus('Update failed: ' + StatusCode.ToString);
    finally
      Response.Free;
    end;
  end
  else
  begin
    // Create new customer
    Response := restAPI.Post(StatusCode, '/customers', '', Body);
    try
      if StatusCode = 201 then
        SetStatus('Customer created')
      else
        SetStatus('Create failed: ' + StatusCode.ToString);
    finally
      Response.Free;
    end;
  end;

  // Refresh the list
```

```
  FSelectedId := '';
  edtName.Text := '';
  edtEmail.Text := '';
  btnLoadClick(nil);
end;

procedure TfrmCustomers.gridCustomersSelectCell(Sender: TObject;
  const ACol, ARow: Integer; var CanSelect: Boolean);
begin
  if ARow < mtCustomers.RecordCount then
  begin
    mtCustomers.RecNo := ARow + 1;
    FSelectedId := mtCustomers.FieldByName('id').AsString;
    edtName.Text := mtCustomers.FieldByName('name').AsString;
    edtEmail.Text := mtCustomers.FieldByName('email').AsString;
  end;
end;

procedure TfrmCustomers.RefreshGrid;
begin
  gridCustomers.RowCount := mtCustomers.RecordCount;
  gridCustomers.ClearColumns;

  var ColId := TStringColumn.Create(gridCustomers);
  ColId.Header := 'ID';
  ColId.Width := 50;
  gridCustomers.AddObject(ColId);

  var ColName := TStringColumn.Create(gridCustomers);
  ColName.Header := 'Name';
  ColName.Width := 200;
  gridCustomers.AddObject(ColName);

  var ColEmail := TStringColumn.Create(gridCustomers);
  ColEmail.Header := 'Email';
  ColEmail.Width := 250;
  gridCustomers.AddObject(ColEmail);

  mtCustomers.First;
  var Row := 0;
  while not mtCustomers.Eof do
  begin
    gridCustomers.Cells[0, Row] := mtCustomers.FieldByName('id').AsString;
    gridCustomers.Cells[1, Row] := mtCustomers.FieldByName('name').AsString;
    gridCustomers.Cells[2, Row] := mtCustomers.FieldByName('email').AsString;
    Inc(Row);
    mtCustomers.Next;
  end;
end;

procedure TfrmCustomers.SetStatus(const Msg: string);
begin
  lblStatus.Text := Msg;
end;

end.
```

---

10. Exercise: Product Catalog

Build a product management application with the following features:

Requirements

- Fetch products from `GET /products` (use jsonplaceholder or your own API)
- Display products in a `TStringGrid`
- Add a search `TEdit` that filters products by title (client-side filtering on the MemTable)
- Add a form to create new products via `POST /products`
- Use async execution for the initial load with a loading indicator

Solution

```
unit ProductCatalog;

interface

uses
  System.SysUtils, System.Classes, System.JSON,
  FMX.Types, FMX.Controls, FMX.Forms, FMX.StdCtrls, FMX.Edit,
  FMX.Grid, FMX.Grid.Style, FMX.ScrollBox, FMX.Layouts,
  FireDAC.Comp.Client,
  Tina4REST, Tina4RESTRequest;

type
  TfrmProducts = class(TForm)
    restAPI: TTina4REST;
    reqProducts: TTina4RESTRequest;
    mtProducts: TFDMemTable;
    gridProducts: TStringGrid;
    edtSearch: TEdit;
    edtTitle: TEdit;
    edtPrice: TEdit;
    btnCreate: TButton;
    lblLoading: TLabel;
    procedure FormCreate(Sender: TObject);
    procedure edtSearchChangeTracking(Sender: TObject);
    procedure btnCreateClick(Sender: TObject);
  private
    procedure OnProductsLoaded(Sender: TObject);
    procedure RefreshGrid;
    procedure FilterGrid(const SearchText: string);
  end;

var
  frmProducts: TfrmProducts;

implementation

{$R *.fmx}

procedure TfrmProducts.FormCreate(Sender: TObject);
begin
  restAPI.BaseUrl := 'https://jsonplaceholder.typicode.com';

  reqProducts.Tina4REST := restAPI;
  reqProducts.EndPoint := '/posts'; // Using posts as stand-in for products
  reqProducts.RequestType := TTina4RequestType.Get;
  reqProducts.MemTable := mtProducts;
  reqProducts.SyncMode := TTina4RestSyncMode.Clear;

  reqProducts.OnExecuteDone := OnProductsLoaded;

  lblLoading.Text := 'Loading products...';
  lblLoading.Visible := True;
  reqProducts.ExecuteRESTCallAsync;
end;

procedure TfrmProducts.OnProductsLoaded(Sender: TObject);
begin
```

```
  TThread.Synchronize(nil, procedure
  begin
    lblLoading.Visible := False;
    RefreshGrid;
  end);
end;

procedure TfrmProducts.RefreshGrid;
begin
  gridProducts.RowCount := mtProducts.RecordCount;
  gridProducts.ClearColumns;

  for var I := 0 to mtProducts.FieldCount - 1 do
  begin
    var Col := TStringColumn.Create(gridProducts);
    Col.Header := mtProducts.Fields[I].FieldName;
    Col.Width := 150;
    gridProducts.AddObject(Col);
  end;

  mtProducts.First;
  var Row := 0;
  while not mtProducts.Eof do
  begin
    for var C := 0 to mtProducts.FieldCount - 1 do
      gridProducts.Cells[C, Row] := mtProducts.Fields[C].AsString;
    Inc(Row);
    mtProducts.Next;
  end;
end;

procedure TfrmProducts.edtSearchChangeTracking(Sender: TObject);
begin
  FilterGrid(edtSearch.Text);
end;

procedure TfrmProducts.FilterGrid(const SearchText: string);
var
  Row: Integer;
begin
  if SearchText = '' then
  begin
    RefreshGrid;
    Exit;
  end;

  Row := 0;
  gridProducts.RowCount := 0;

  mtProducts.First;
  while not mtProducts.Eof do
  begin
    var Title := mtProducts.FieldByName('title').AsString;
    if Title.ToLower.Contains(SearchText.ToLower) then
    begin
      gridProducts.RowCount := Row + 1;
      for var C := 0 to mtProducts.FieldCount - 1 do
        gridProducts.Cells[C, Row] := mtProducts.Fields[C].AsString;
      Inc(Row);
```

```
    end;
    mtProducts.Next;
  end;
end;

procedure TfrmProducts.btnCreateClick(Sender: TObject);
var
  StatusCode: Integer;
  Response: TJSONObject;
begin
  Response := restAPI.Post(StatusCode, '/posts', '',
    Format('{"title": "%s", "body": "%s", "userId": 1}',
      [edtTitle.Text, edtPrice.Text]));
  try
    if StatusCode = 201 then
    begin
      ShowMessage('Product created with ID: ' + Response.GetValue<String>('id'));
      edtTitle.Text := '';
      edtPrice.Text := '';
    end
    else
      ShowMessage('Failed: ' + StatusCode.ToString);
  finally
    Response.Free;
  end;
end;

end.
```

---

11. Common Gotchas

Forgetting to Free TJSONObject

Symptom: Memory usage grows over time. ReportMemoryLeaksOnShutdown shows leaks.

Fix: Every `Get`, `Post`, `Patch`, `Put`, and `Delete` call returns a `TJSONObject` that you own. Always wrap in `try..finally`:

```
var Response := Tina4REST1.Get(StatusCode, '/data');
try
  // use Response
finally
  Response.Free;  // Always. Every time.
end;
```

Not Checking StatusCode

Symptom: Application crashes when trying to read fields from an error response.

Fix: Always check the status code before accessing response data:

```
Response := Tina4REST1.Get(StatusCode, '/users/999');
try
  if StatusCode = 200 then
    ProcessUser(Response)
  else if StatusCode = 404 then
    ShowMessage('User not found')
```

```
  else
    ShowMessage('Unexpected error: ' + StatusCode.ToString);
finally
  Response.Free;
end;
```

Async Thread Safety

Symptom: Intermittent access violations, garbled UI, or "Canvas does not allow drawing" errors.

Fix: Never touch UI controls from `OnExecuteDone` without `TThread.Synchronize`:

```
// WRONG -- will crash randomly
procedure TForm1.OnDone(Sender: TObject);
begin
  lblStatus.Text := 'Done';  // Main thread violation
end;

// CORRECT
procedure TForm1.OnDone(Sender: TObject);
begin
  TThread.Synchronize(nil, procedure
  begin
    lblStatus.Text := 'Done';  // Safe -- runs on main thread
  end);
end;
```

DataKey Mismatch

Symptom: MemTable is empty after a successful request.

Fix: Check that `DataKey` matches the JSON structure. If the API returns `{"data": [...]}`, set `DataKey` to `data`. If it returns `{"results": [...]}`, set it to `results`. If the response is a bare array `[...]`, leave `DataKey` empty.

BaseUrl Trailing Slash

Symptom: 404 errors on endpoints that work in the browser.

Fix: Do not include a trailing slash on `BaseUrl`. The endpoint already starts with `/`:

```
// WRONG
Tina4REST1.BaseUrl := 'https://api.example.com/v1/';
// Endpoint '/users' becomes 'https://api.example.com/v1//users'

// CORRECT
Tina4REST1.BaseUrl := 'https://api.example.com/v1';
```

Summary

What	How
Base configuration	`TTina4REST` -- set `BaseUrl`, auth
Basic Auth	Set `Username` and `Password`
Bearer token	`SetBearer('token')`
Direct GET	`Tina4REST1.Get(StatusCode, '/endpoint', 'params')`
Direct POST	`Tina4REST1.Post(StatusCode, '/endpoint', '', Body)`
Direct PATCH	`Tina4REST1.Patch(StatusCode, '/endpoint', '', Body)`
Direct PUT	`Tina4REST1.Put(StatusCode, '/endpoint', '', Body)`
Direct DELETE	`Tina4REST1.Delete(StatusCode, '/endpoint')`
Declarative GET	`TTina4RESTRequest` -- set endpoint, MemTable, `ExecuteRESTCall`
Master/Detail	Set `MasterSource`, use `{fieldName}` placeholders
POST from MemTable	Set `SourceMemTable`, call `ExecuteRESTCall`
Async	`ExecuteRESTCallAsync` + `OnExecuteDone` + `TThread.Synchronize`
Memory rule	Every `TJSONObject` returned must be freed by the caller

JSON & Data Binding

The Bridge Between APIs and Grids

Your API returns JSON. Your grid displays MemTable rows. Between these two worlds sits a translation layer -- field names need converting, dates need formatting, nested objects need flattening, and records need matching for updates. Tina4 Delphi handles all of this with a set of utility functions and one component.

This chapter covers the full JSON pipeline: parsing raw strings, converting database queries to JSON, populating MemTables from JSON, syncing changes, and binding data declaratively with `TTina4JSONAdapter`.

---

1. JSON Parsing Utilities

Before you can work with JSON data, you need to parse it. `Tina4Core.pas` provides four parsing functions that handle the common cases.

StrToJSONObject

Parses a JSON string into a `TJSONObject`. Returns `nil` if parsing fails -- always check with `Assigned`.

```
uses Tina4Core;

var Obj := StrToJSONObject('{"name": "Alice", "age": 30, "active": true}');
try
  if Assigned(Obj) then
  begin
    ShowMessage(Obj.GetValue<String>('name'));     // 'Alice'
    ShowMessage(Obj.GetValue<Integer>('age').ToString);  // '30'
    ShowMessage(Obj.GetValue<Boolean>('active').ToString); // 'True'
  end
  else
    ShowMessage('Invalid JSON');
finally
  Obj.Free;
end;
```

StrToJSONArray

Parses a JSON string into a `TJSONArray`. Use this when the root element is an array:

```
var Arr := StrToJSONArray('[{"id": 1}, {"id": 2}, {"id": 3}]');
try
  if Assigned(Arr) then
    for var I := 0 to Arr.Count - 1 do
      ShowMessage((Arr.Items[I] as TJSONObject).GetValue<String>('id'));
finally
  Arr.Free;
end;
```

StrToJSONValue

When you do not know whether the input is an object, array, string, number, or boolean:

```
var Val := StrToJSONValue(SomeInput);
try
  if Val is TJSONObject then
    ProcessObject(Val as TJSONObject)
  else if Val is TJSONArray then
    ProcessArray(Val as TJSONArray)
  else
    ShowMessage('Primitive: ' + Val.Value);
finally
  Val.Free;
end;
```

BytesToJSONObject

Parses raw `TBytes` directly -- the typical output of `SendHttpRequest`:

```
var
  StatusCode: Integer;
  Response: TBytes;
begin
  Response := SendHttpRequest(StatusCode, 'https://api.example.com', '/users');
  var JSON := BytesToJSONObject(Response);
  try
    if Assigned(JSON) then
      Memo1.Lines.Text := JSON.Format;
  finally
    JSON.Free;
  end;
end;
```

GetJSONFieldName

Strips surrounding quotes from a JSON field name. Useful when iterating `TJSONPair` elements:

```
GetJSONFieldName('"firstName"');  // 'firstName'
GetJSONFieldName('age');          // 'age'
```

---

2. TTina4JSONAdapter -- Static JSON to MemTable

`TTina4JSONAdapter` is the declarative way to bind JSON data to a `TFDMemTable`. Drop it on your form, set the JSON, set the data key, and execute. No parsing code. No field definition code. No population loops.

From Static JSON

```
// Design-time or runtime:
Tina4JSONAdapter1.MemTable := FDMemTable1;
Tina4JSONAdapter1.DataKey := 'products';
Tina4JSONAdapter1.JSONData.Text :=
  '{"products": [' +
  '  {"id": "1", "name": "Widget", "price": 9.99},' +
  '  {"id": "2", "name": "Gadget", "price": 24.99},' +
  '  {"id": "3", "name": "Doohickey", "price": 4.50}' +
  ']}';
Tina4JSONAdapter1.Execute;
// FDMemTable1 now has 3 rows with id, name, price columns
```

From MasterSource

Link the adapter to a `TTina4RESTRequest` and it auto-executes whenever the master's `OnExecuteDone` fires:

```
// The REST request fetches data that contains embedded JSON
Tina4RESTRequest1.EndPoint := '/dashboard';
Tina4RESTRequest1.MemTable := FDMemTableDashboard;

// The adapter extracts a nested array from the response
Tina4JSONAdapter1.MasterSource := Tina4RESTRequest1;
Tina4JSONAdapter1.DataKey := 'recentOrders';
Tina4JSONAdapter1.MemTable := FDMemTableOrders;
// When Tina4RESTRequest1 completes, FDMemTableOrders auto-populates
```

This works well for APIs that return complex nested responses. The REST request gets the whole response into one MemTable. The JSON adapter extracts a specific nested array into another MemTable.

Sync Mode

By default, `Execute` clears the MemTable and replaces all data. For incremental updates, use `Sync` mode:

```
Tina4JSONAdapter1.SyncMode := TTina4RestSyncMode.Sync;
Tina4JSONAdapter1.IndexFieldNames := 'id';
```

Sync Mode	Behavior
`Clear` (default)	Empties the table first, then appends all records
`Sync`	Matches records by `IndexFieldNames`, updates existing rows, inserts new ones

`Sync` mode requires `IndexFieldNames` to be set. This is the field (or fields) used to match existing rows against incoming JSON records. Without it, sync mode cannot determine which rows to update.

---

3. Database to JSON

Going the other direction -- from database to JSON -- is equally common. You query a database and need to send the results to a REST API, save to a file, or display in an HTML template.

GetJSONFromDB

Executes a SQL query and returns the results as a `TJSONObject`. Three automatic conversions happen:

- **Field names** convert from `snake_case` to `camelCase` (e.g., `first_name` becomes `firstName`)
- **DateTime fields** format as ISO 8601 (e.g., `2024-06-15T14:30:00.000Z`)
- **Blob fields** encode as Base64

```
// Simple query
var Result := GetJSONFromDB(FDConnection1, 'SELECT * FROM users');
try
  Memo1.Lines.Text := Result.Format;
  // {"records": [
  //   {"id": "1", "firstName": "Alice", "email": "alice@example.com", ...},
  //   {"id": "2", "firstName": "Bob", "email": "bob@example.com", ...}
  // ]}
finally
  Result.Free;
end;
```

The default dataset key is `records`. To use a custom key:

```
var Result := GetJSONFromDB(FDConnection1,
  'SELECT * FROM cats', nil, 'cats');
// {"cats": [{"id": "1", "name": "Whiskers"}, ...]}
```

With Parameters

Use `TFDParams` for parameterized queries to prevent SQL injection:

```
var Params := TFDParams.Create;
try
  Params.Add('status', 'active');
  Params.Add('minAge', 18);

  var Result := GetJSONFromDB(FDConnection1,
    'SELECT * FROM users WHERE status = :status AND age >= :minAge',
    Params);
  try
    Memo1.Lines.Text := Result.Format;
  finally
    Result.Free;
  end;
finally
  Params.Free;
end;
```

GetJSONFromTable

Converts an existing `TFDMemTable` or `TFDTable` to JSON. Useful when you have data already loaded and need to serialize it:

```
// Basic conversion
var JSON := GetJSONFromTable(FDMemTable1);
try
  Memo1.Lines.Text := JSON.Format;
  // {"records": [{"id": "1", "name": "Item1"}, ...]}
finally
  JSON.Free;
end;
```

Ignore specific fields (passwords, internal IDs):

```
var JSON := GetJSONFromTable(FDMemTable1, 'records', 'password,internal_id');
```

Ignore blank values to reduce payload size:

```
var JSON := GetJSONFromTable(FDMemTable1, 'records', '', True);
// Fields with empty string values are omitted from each record
```

---

4. JSON to MemTable

The reverse pipeline. You have JSON data and need it in a `TFDMemTable` for display, editing, or further processing.

GetFieldDefsFromJSONObject

Creates field definitions on a MemTable from a JSON object's structure. You call this once to set up the schema, then populate rows:

```
var JSONObj := StrToJSONObject(
  '{"firstName": "Alice", "age": 30, "address": {"city": "Cape Town"}}');
try
  GetFieldDefsFromJSONObject(JSONObj, FDMemTable1, True);
  // Creates fields:
  //   first_name : ftString  (camelCase converted to snake_case with True flag)
  //   age        : ftString
  //   address    : ftMemo    (nested object becomes ftMemo)
  FDMemTable1.CreateDataSet;
finally
  JSONObj.Free;
end;
```

The third parameter controls snake_case conversion. Pass `True` to convert `firstName` to `first_name`. Pass `False` to keep JSON field names as-is.

Nested objects and arrays become `ftMemo` fields containing the serialized JSON string.

PopulateMemTableFromJSON

The main workhorse. Takes a JSON string, extracts the array at the specified data key, and populates a MemTable. If the MemTable has no field definitions, they are created automatically from the first JSON object.

Clear Mode (Default)

Empties the table and replaces all data:

```
var JSONStr :=
  '{"records": [' +
  '  {"id": "1", "name": "Alice", "email": "alice@example.com"},' +
  '  {"id": "2", "name": "Bob", "email": "bob@example.com"}' +
  ']}';

PopulateMemTableFromJSON(FDMemTable1, 'records', JSONStr);
// FDMemTable1 has 2 rows, any previous data is gone
```

Sync Mode

Matches existing rows by key fields and updates them. New rows are inserted. Existing rows not in the JSON are left unchanged:

```
// Initial load
PopulateMemTableFromJSON(FDMemTable1, 'records',
  '{"records": [{"id": "1", "name": "Alice"}, {"id": "2", "name": "Bob"}]}');

// Later: update Alice, add Charlie, Bob stays unchanged
PopulateMemTableFromJSON(FDMemTable1, 'records',
  '{"records": [{"id": "1", "name": "Alice Updated"}, {"id": "3", "name": "Charlie"}]}',
  'id', TTina4RestSyncMode.Sync);

// Result: 3 rows
//   id=1: Alice Updated (updated)
//   id=2: Bob (unchanged)
//   id=3: Charlie (inserted)
```

The fourth parameter is `IndexFieldNames` -- the field(s) used for matching. For composite keys, separate with semicolons: `'tenantId;userId'`.

PopulateTableFromJSON

Inserts or updates rows directly into a database table (not a MemTable) from JSON. Uses a primary key for upsert logic:

```
var Result := PopulateTableFromJSON(
  FDConnection1,           // database connection
  'users',                 // table name
  '{"response": [{"name": "Alice"}, {"name": "Bob"}]}',
  'response',              // data key
  'id');                   // primary key field for upsert
```

This is useful for bulk imports -- JSON data goes directly to the database without an intermediate MemTable.

---

5. Naming Conventions

Tina4 Delphi automatically converts between naming conventions at every boundary:

Direction	From	To	Example

Database to JSON	`snake_case`	`camelCase`	`first_name` becomes `firstName`
JSON to MemTable	`camelCase`	`snake_case` (optional)	`firstName` becomes `first_name`

CamelCase

```
CamelCase('first_name');     // 'firstName'
CamelCase('id');             // 'id'
CamelCase('user_email');     // 'userEmail'
CamelCase('created_at');     // 'createdAt'
```

SnakeCase

```
SnakeCase('firstName');      // 'first_name'
SnakeCase('userEmail');      // 'user_email'
SnakeCase('createdAt');      // 'created_at'
```

This matters because databases typically use `snake_case` column names while JSON APIs use `camelCase` keys. Tina4 handles the translation transparently when using `GetJSONFromDB` and `GetFieldDefsFromJSONObject`.

---

6. Complete Example: Data Import/Export Tool

A realistic scenario: fetch data from an API, display it in a grid, let the user edit rows, and push changes back to the API.

```
unit ImportExport;

interface

uses
  System.SysUtils, System.Classes, System.JSON,
  FMX.Types, FMX.Controls, FMX.Forms, FMX.StdCtrls, FMX.Edit,
  FMX.Grid, FMX.Grid.Style, FMX.ScrollBox, FMX.Memo, FMX.Layouts,
  FireDAC.Comp.Client,
  Tina4Core, Tina4REST, Tina4RESTRequest;

type
  TfrmImportExport = class(TForm)
    restAPI: TTina4REST;
    mtData: TFDMemTable;
    gridData: TStringGrid;
    btnFetch: TButton;
    btnPushChanges: TButton;
    memoLog: TMemo;
    lblStatus: TLabel;
    procedure FormCreate(Sender: TObject);
    procedure btnFetchClick(Sender: TObject);
    procedure btnPushChangesClick(Sender: TObject);
  private
```

```
    procedure RefreshGrid;
    procedure Log(const Msg: string);
  end;

var
  frmImportExport: TfrmImportExport;

implementation

{$R *.fmx}

procedure TfrmImportExport.FormCreate(Sender: TObject);
begin
  restAPI.BaseUrl := 'https://jsonplaceholder.typicode.com';
end;

procedure TfrmImportExport.btnFetchClick(Sender: TObject);
var
  StatusCode: Integer;
  Response: TJSONObject;
begin
  Log('Fetching users...');

  Response := restAPI.Get(StatusCode, '/users');
  try
    if StatusCode <> 200 then
    begin
      Log('Failed: HTTP ' + StatusCode.ToString);
      Exit;
    end;

    // The response is a JSON array, but Tina4REST wraps it
    // Use PopulateMemTableFromJSON for direct control
    PopulateMemTableFromJSON(mtData, '', Response.ToString);
    RefreshGrid;
    Log(Format('Loaded %d users', [mtData.RecordCount]));
  finally
    Response.Free;
  end;
end;

procedure TfrmImportExport.btnPushChangesClick(Sender: TObject);
var
  StatusCode: Integer;
  Response: TJSONObject;
  JSON: TJSONObject;
begin
  // Serialize the MemTable to JSON
  JSON := GetJSONFromTable(mtData);
  try
    Log('Pushing changes...');
    Log('Payload: ' + JSON.ToString);

    // In a real app, POST this to your API
    Response := restAPI.Post(StatusCode, '/users', '', JSON.ToString);
    try
      if StatusCode in [200, 201] then
        Log('Changes pushed successfully')
      else
```

```
        Log('Push failed: HTTP ' + StatusCode.ToString);
    finally
      Response.Free;
    end;
  finally
    JSON.Free;
  end;
end;

procedure TfrmImportExport.RefreshGrid;
begin
  gridData.RowCount := mtData.RecordCount;
  gridData.ClearColumns;

  for var I := 0 to mtData.FieldCount - 1 do
  begin
    var Col := TStringColumn.Create(gridData);
    Col.Header := mtData.Fields[I].FieldName;
    Col.Width := 150;
    gridData.AddObject(Col);
  end;

  mtData.First;
  var Row := 0;
  while not mtData.Eof do
  begin
    for var C := 0 to mtData.FieldCount - 1 do
      gridData.Cells[C, Row] := mtData.Fields[C].AsString;
    Inc(Row);
    mtData.Next;
  end;
end;

procedure TfrmImportExport.Log(const Msg: string);
begin
  memoLog.Lines.Add(FormatDateTime('hh:nn:ss', Now) + ' ' + Msg);
end;

end.
```

---

7. Complete Example: Master-Detail Pattern

Customers in the top grid. Orders for the selected customer in the bottom grid. The orders grid updates automatically when you select a different customer.

```
unit MasterDetail;

interface

uses
  System.SysUtils, System.Classes, System.JSON,
  FMX.Types, FMX.Controls, FMX.Forms, FMX.StdCtrls,
  FMX.Grid, FMX.Grid.Style, FMX.ScrollBox, FMX.Layouts,
  FireDAC.Comp.Client,
  Tina4Core, Tina4REST, Tina4RESTRequest, Tina4JSONAdapter;

type
```

```
  TfrmMasterDetail = class(TForm)
    restAPI: TTina4REST;
    reqCustomers: TTina4RESTRequest;
    adapterOrders: TTina4JSONAdapter;
    mtCustomers: TFDMemTable;
    mtOrders: TFDMemTable;
    gridCustomers: TStringGrid;
    gridOrders: TStringGrid;
    btnLoad: TButton;
    lblCustomerCount: TLabel;
    lblOrderCount: TLabel;
    procedure FormCreate(Sender: TObject);
    procedure btnLoadClick(Sender: TObject);
    procedure gridCustomersSelectCell(Sender: TObject; const ACol, ARow: Integer;
      var CanSelect: Boolean);
  private
    FOrdersData: TJSONObject;
    procedure RefreshCustomerGrid;
    procedure LoadOrdersForCustomer(CustomerId: string);
    procedure RefreshOrderGrid;
  end;

var
  frmMasterDetail: TfrmMasterDetail;

implementation

{$R *.fmx}

procedure TfrmMasterDetail.FormCreate(Sender: TObject);
begin
  restAPI.BaseUrl := 'https://api.example.com/v1';
  restAPI.SetBearer('your-token-here');

  reqCustomers.Tina4REST := restAPI;
  reqCustomers.EndPoint := '/customers';
  reqCustomers.RequestType := TTina4RequestType.Get;
  reqCustomers.DataKey := 'records';
  reqCustomers.MemTable := mtCustomers;
  reqCustomers.SyncMode := TTina4RestSyncMode.Clear;

  FOrdersData := nil;
end;

procedure TfrmMasterDetail.btnLoadClick(Sender: TObject);
begin
  reqCustomers.ExecuteRESTCall;
  RefreshCustomerGrid;
  lblCustomerCount.Text := Format('%d customers', [mtCustomers.RecordCount]);

  // Auto-select first customer
  if mtCustomers.RecordCount > 0 then
  begin
    mtCustomers.First;
    LoadOrdersForCustomer(mtCustomers.FieldByName('id').AsString);
  end;
end;

procedure TfrmMasterDetail.gridCustomersSelectCell(Sender: TObject;
```

```
  const ACol, ARow: Integer; var CanSelect: Boolean);
begin
  if ARow < mtCustomers.RecordCount then
  begin
    mtCustomers.RecNo := ARow + 1;
    LoadOrdersForCustomer(mtCustomers.FieldByName('id').AsString);
  end;
end;

procedure TfrmMasterDetail.LoadOrdersForCustomer(CustomerId: string);
var
  StatusCode: Integer;
  Response: TJSONObject;
begin
  Response := restAPI.Get(StatusCode,
    '/customers/' + CustomerId + '/orders');
  try
    if StatusCode = 200 then
    begin
      PopulateMemTableFromJSON(mtOrders, 'records', Response.ToString);
      RefreshOrderGrid;
      lblOrderCount.Text := Format('%d orders', [mtOrders.RecordCount]);
    end
    else
    begin
      mtOrders.EmptyDataSet;
      RefreshOrderGrid;
      lblOrderCount.Text := '0 orders';
    end;
  finally
    Response.Free;
  end;
end;

procedure TfrmMasterDetail.RefreshCustomerGrid;
begin
  gridCustomers.RowCount := mtCustomers.RecordCount;
  gridCustomers.ClearColumns;

  var ColId := TStringColumn.Create(gridCustomers);
  ColId.Header := 'ID';
  ColId.Width := 50;
  gridCustomers.AddObject(ColId);

  var ColName := TStringColumn.Create(gridCustomers);
  ColName.Header := 'Name';
  ColName.Width := 200;
  gridCustomers.AddObject(ColName);

  var ColEmail := TStringColumn.Create(gridCustomers);
  ColEmail.Header := 'Email';
  ColEmail.Width := 250;
  gridCustomers.AddObject(ColEmail);

  mtCustomers.First;
  var Row := 0;
  while not mtCustomers.Eof do
  begin
    gridCustomers.Cells[0, Row] := mtCustomers.FieldByName('id').AsString;
```

```
    gridCustomers.Cells[1, Row] := mtCustomers.FieldByName('name').AsString;
    gridCustomers.Cells[2, Row] := mtCustomers.FieldByName('email').AsString;
    Inc(Row);
    mtCustomers.Next;
  end;
end;

procedure TfrmMasterDetail.RefreshOrderGrid;
begin
  gridOrders.RowCount := mtOrders.RecordCount;
  gridOrders.ClearColumns;

  for var I := 0 to mtOrders.FieldCount - 1 do
  begin
    var Col := TStringColumn.Create(gridOrders);
    Col.Header := mtOrders.Fields[I].FieldName;
    Col.Width := 120;
    gridOrders.AddObject(Col);
  end;

  mtOrders.First;
  var Row := 0;
  while not mtOrders.Eof do
  begin
    for var C := 0 to mtOrders.FieldCount - 1 do
      gridOrders.Cells[C, Row] := mtOrders.Fields[C].AsString;
    Inc(Row);
    mtOrders.Next;
  end;
end;

end.
```

---

8. Exercise: JSON Viewer

Build a universal JSON viewer that can load any JSON file, auto-create MemTable fields, display the data in a grid, and allow editing.

Requirements

- An "Open File" button that loads a `.json` file from disk
- A `TEdit` for specifying the data key (default: `records`)
- Auto-detect field definitions from the JSON structure
- Display the data in a `TStringGrid`
- Allow the user to edit cells in the grid
- A "Save" button that writes the modified data back to the JSON file

Solution

```
unit JSONViewer;

interface

uses
  System.SysUtils, System.Classes, System.JSON, System.IOUtils,
  FMX.Types, FMX.Controls, FMX.Forms, FMX.StdCtrls, FMX.Edit,
  FMX.Grid, FMX.Grid.Style, FMX.ScrollBox, FMX.Dialogs, FMX.Layouts,
  FireDAC.Comp.Client,
  Tina4Core;

type
  TfrmJSONViewer = class(TForm)
    btnOpen: TButton;
    btnSave: TButton;
    edtDataKey: TEdit;
    gridData: TStringGrid;
    mtData: TFDMemTable;
    lblStatus: TLabel;
    lblDataKey: TLabel;
    OpenDialog1: TOpenDialog;
    SaveDialog1: TSaveDialog;
    procedure btnOpenClick(Sender: TObject);
    procedure btnSaveClick(Sender: TObject);
    procedure FormCreate(Sender: TObject);
  private
    FCurrentFile: string;
    FOriginalJSON: string;
    procedure LoadJSON(const FileName: string);
    procedure RefreshGrid;
  end;

var
  frmJSONViewer: TfrmJSONViewer;

implementation

{$R *.fmx}

procedure TfrmJSONViewer.FormCreate(Sender: TObject);
begin
  edtDataKey.Text := 'records';
  OpenDialog1.Filter := 'JSON files (*.json)|*.json|All files (*.*)|*.*';
  SaveDialog1.Filter := 'JSON files (*.json)|*.json';
end;

procedure TfrmJSONViewer.btnOpenClick(Sender: TObject);
begin
  if OpenDialog1.Execute then
    LoadJSON(OpenDialog1.FileName);
end;

procedure TfrmJSONViewer.LoadJSON(const FileName: string);
var
  JSONStr: string;
  DataKey: string;
begin
```

```
  FCurrentFile := FileName;
  JSONStr := TFile.ReadAllText(FileName);
  FOriginalJSON := JSONStr;
  DataKey := edtDataKey.Text;

  // Clear existing data
  mtData.Close;
  mtData.FieldDefs.Clear;

  // Try to parse and detect structure
  var JSONVal := StrToJSONValue(JSONStr);
  try
    if JSONVal is TJSONArray then
    begin
      // Root is an array -- wrap it for PopulateMemTableFromJSON
      var Wrapped := Format('{"%s": %s}', [DataKey, JSONStr]);
      PopulateMemTableFromJSON(mtData, DataKey, Wrapped);
    end
    else if JSONVal is TJSONObject then
    begin
      PopulateMemTableFromJSON(mtData, DataKey, JSONStr);
    end
    else
    begin
      lblStatus.Text := 'JSON is neither an object nor an array';
      Exit;
    end;
  finally
    JSONVal.Free;
  end;

  RefreshGrid;
  lblStatus.Text := Format('Loaded %d records from %s',
    [mtData.RecordCount, ExtractFileName(FileName)]);
end;

procedure TfrmJSONViewer.RefreshGrid;
begin
  gridData.RowCount := mtData.RecordCount;
  gridData.ClearColumns;

  for var I := 0 to mtData.FieldCount - 1 do
  begin
    var Col := TStringColumn.Create(gridData);
    Col.Header := mtData.Fields[I].FieldName;
    Col.Width := 150;
    gridData.AddObject(Col);
  end;

  mtData.First;
  var Row := 0;
  while not mtData.Eof do
  begin
    for var C := 0 to mtData.FieldCount - 1 do
      gridData.Cells[C, Row] := mtData.Fields[C].AsString;
    Inc(Row);
    mtData.Next;
  end;
end;
```

```
procedure TfrmJSONViewer.btnSaveClick(Sender: TObject);
var
  JSON: TJSONObject;
  FileName: string;
begin
  // Read grid edits back into MemTable
  mtData.First;
  var Row := 0;
  while not mtData.Eof do
  begin
    mtData.Edit;
    for var C := 0 to mtData.FieldCount - 1 do
      mtData.Fields[C].AsString := gridData.Cells[C, Row];
    mtData.Post;
    Inc(Row);
    mtData.Next;
  end;

  // Serialize to JSON
  JSON := GetJSONFromTable(mtData, edtDataKey.Text);
  try
    if FCurrentFile <> '' then
      FileName := FCurrentFile
    else if SaveDialog1.Execute then
      FileName := SaveDialog1.FileName
    else
      Exit;

    TFile.WriteAllText(FileName, JSON.Format);
    lblStatus.Text := 'Saved to ' + ExtractFileName(FileName);
  finally
    JSON.Free;
  end;
end;

end.
```

---

9. Common Gotchas

TJSONObject Memory Management

Symptom: Memory leaks reported by `ReportMemoryLeaksOnShutdown`.

Fix: Every function that returns a `TJSONObject` -- `StrToJSONObject`, `BytesToJSONObject`, `GetJSONFromDB`, `GetJSONFromTable`, `Get`, `Post`, etc. -- creates an object on the heap. You must free it:

```
// Pattern: always use try..finally
var Obj := StrToJSONObject(SomeString);
try
  // work with Obj
finally
  Obj.Free;
end;
```

Do not free child objects extracted with `GetValue<TJSONObject>` or `GetValue<TJSONArray>` -- they are owned by the parent. Freeing the parent frees all children.

Nested JSON Becoming ftMemo Fields

Symptom: A field contains `{"city": "Cape Town", "zip": "8001"}` instead of the expected flat value.

Explanation: When `GetFieldDefsFromJSONObject` encounters a nested JSON object or array, it creates an `ftMemo` field containing the serialized JSON string. This is by design -- there is no automatic flattening.

Fix: If you need flat fields, pre-process the JSON to flatten it before populating the MemTable. Or use a second `TTina4JSONAdapter` to extract nested data into a separate MemTable.

Sync Mode Without IndexFieldNames

Symptom: Duplicate rows appear in the MemTable after sync.

Fix: When using `TTina4RestSyncMode.Sync`, you must set `IndexFieldNames`. Without it, the sync has no way to match incoming records to existing rows, so it appends everything:

```
// WRONG -- no index, sync inserts duplicates
PopulateMemTableFromJSON(mtData, 'records', JSONStr,
  '', TTina4RestSyncMode.Sync);

// CORRECT -- match by id field
PopulateMemTableFromJSON(mtData, 'records', JSONStr,
  'id', TTina4RestSyncMode.Sync);
```

DataKey Does Not Exist

Symptom: MemTable is empty after `PopulateMemTableFromJSON`, even though the JSON contains data.

Fix: Verify the data key matches the actual JSON structure. Common mismatches:

```
// API returns {"data": [...]}
PopulateMemTableFromJSON(mtData, 'records', JSONStr);  // WRONG: no "records" key
PopulateMemTableFromJSON(mtData, 'data', JSONStr);     // CORRECT

// API returns a bare array [...]
PopulateMemTableFromJSON(mtData, 'records', JSONStr);   // WRONG: no wrapper object
// Wrap it first:
var Wrapped := '{"records": ' + JSONStr + '}';
PopulateMemTableFromJSON(mtData, 'records', Wrapped);   // CORRECT
```

Date Fields Not Parsing

Symptom: Date values appear as raw strings like `2024-06-15T14:30:00.000Z` instead of `TDateTime` values.

Explanation: `PopulateMemTableFromJSON` creates all fields as `ftString` by default (auto-detected from JSON, which has no date type). Dates are stored as strings.

Fix: Use `IsDate` and `JSONDateToDateTime` for explicit conversion:

```
if IsDate(mtData.FieldByName('createdAt').AsString) then
```

```
begin
  var DT := JSONDateToDateTime(mtData.FieldByName('createdAt').AsString);
  // DT is now a TDateTime you can format or compare
end;
```

Summary

What	How
Parse JSON string	`StrToJSONObject(str)` / `StrToJSONArray(str)`
Parse HTTP response	`BytesToJSONObject(bytes)`
JSON adapter	`TTina4JSONAdapter` -- set `MemTable`, `DataKey`, `JSONData`, `Execute`
Adapter from REST	Set `MasterSource` to a `TTina4RESTRequest`
Sync mode	`SyncMode := Sync` + `IndexFieldNames := 'id'`
DB to JSON	`GetJSONFromDB(Connection, SQL)` -- auto camelCase, ISO dates
Table to JSON	`GetJSONFromTable(MemTable)`
JSON to MemTable	`PopulateMemTableFromJSON(MemTable, DataKey, JSON)`
JSON to DB	`PopulateTableFromJSON(Connection, TableName, JSON, DataKey, PK)`
Field defs from JSON	`GetFieldDefsFromJSONObject(JSONObj, MemTable, SnakeCase)`
camelCase convert	`CamelCase('snake_name')`
snake_case convert	`SnakeCase('camelName')`
Date check	`IsDate(Value)`
Date to ISO	`GetJSONDate(DateTime)`
ISO to Date	`JSONDateToDateTime(ISOString)`

HTML Rendering

A Web Browser Inside Your Desktop App

Your FMX application needs a dashboard with styled cards, tables, and action buttons. You could build it with native controls -- dozens of `TLabel`, `TRectangle`, `TPanel`, and `TLayout` components, each positioned and styled by hand. Or you could write HTML.

`TTina4HTMLRender` is an FMX control that parses HTML and CSS and renders them directly on a canvas. It is not a web browser. It does not embed Chromium. It does not spawn a separate process. It is a native FMX control that understands HTML structure, CSS styling, form controls, and interactive events. Drop it on your form, set the `HTML.Text` property, and you have a styled, interactive UI in your desktop application.

---

1. Basic Usage

Drop a `TTina4HTMLRender` on your form. Set its `Align` to `Client` so it fills the form. Then set the HTML:

```
Tina4HTMLRender1.HTML.Text :=
  '<h1>Hello from Tina4</h1>' +
  '<p>This is <b>bold</b> and <i>italic</i> text rendered on an FMX canvas.</p>' +
  '<hr>' +
  '<p style="color: blue; font-size: 18px;">Styled paragraph with inline CSS.</p>';
```

Run the app. You see a rendered heading, a paragraph with bold and italic, a horizontal rule, and a blue styled paragraph. No web view. No Chromium. Just canvas drawing.

Updating Content

Change the HTML at any time and the control re-renders:

```
procedure TForm1.btnRefreshClick(Sender: TObject);
begin
  Tina4HTMLRender1.HTML.Text :=
    '<h1>Updated at ' + FormatDateTime('hh:nn:ss', Now) + '</h1>';
end;
```

---

2. Supported HTML Elements

The renderer supports a practical subset of HTML -- everything you need for dashboards, forms, reports, and documentation displays.

Block Elements

`h1` through `h6`, `p`, `div`, `pre`, `blockquote`, `hr`, `fieldset`

```
Tina4HTMLRender1.HTML.Text :=
  '<div style="padding: 10px; border: 1px solid #ccc;">' +
  '  <h2>Section Title</h2>' +
```

```
  '  <p>Regular paragraph text.</p>' +
  '  <blockquote>A quoted passage with special styling.</blockquote>' +
  '  <pre>Preformatted code block</pre>' +
  '</div>';
```

Inline Elements

`span`, `b`/`strong`, `i`/`em`, `a`, `br`, `small`, `label`, `kbd`, `abbr`, `cite`, `q`, `var`, `samp`, `dfn`, `time`

Lists

`ul`, `ol`, `li` with bullet and number markers. The `list-style-type` CSS property is supported:

```
Tina4HTMLRender1.HTML.Text :=
  '<h3>Features</h3>' +
  '<ul>' +
  '  <li>REST client components</li>' +
  '  <li>JSON data binding</li>' +
  '  <li>HTML rendering with CSS</li>' +
  '</ul>' +
  '<h3>Steps</h3>' +
  '<ol>' +
  '  <li>Install the packages</li>' +
  '  <li>Drop components on form</li>' +
  '  <li>Set properties and run</li>' +
  '</ol>';
```

Tables

`table`, `tr`, `td`, `th`, `thead`, `tbody`, `tfoot` with collapsed borders:

```
Tina4HTMLRender1.HTML.Text :=
  '<table style="width: 100%; border-collapse: collapse;">' +
  '  <thead>' +
  '    <tr>' +
  '      <th style="border: 1px solid #ddd; padding: 8px; background: #f5f5f5;">Name</th>' +
  '      <th style="border: 1px solid #ddd; padding: 8px; background: #f5f5f5;">Email</th>' +
  '      <th style="border: 1px solid #ddd; padding: 8px; background: #f5f5f5;">Status</th>' +
  '    </tr>' +
  '  </thead>' +
  '  <tbody>' +
  '    <tr>' +
  '      <td style="border: 1px solid #ddd; padding: 8px;">Alice</td>' +
  '      <td style="border: 1px solid #ddd; padding: 8px;">alice@example.com</td>' +
  '      <td style="border: 1px solid #ddd; padding: 8px;">Active</td>' +
  '    </tr>' +
  '    <tr>' +
  '      <td style="border: 1px solid #ddd; padding: 8px;">Bob</td>' +
  '      <td style="border: 1px solid #ddd; padding: 8px;">bob@example.com</td>' +
  '      <td style="border: 1px solid #ddd; padding: 8px;">Inactive</td>' +
  '    </tr>' +
  '  </tbody>' +
  '</table>';
```

Images

`img` with HTTP download, async loading, and disk-based caching:

```
Tina4HTMLRender1.CacheEnabled := True;
Tina4HTMLRender1.CacheDir := 'C:\MyApp\cache';
```

```
Tina4HTMLRender1.HTML.Text :=
  '<img src="https://picsum.photos/300/200" width="300" height="200">' +
  '<p>Image loaded from the web and cached to disk.</p>';
```

---

3. CSS Support

The renderer supports a substantial CSS feature set -- enough for professional-looking UIs without reaching for native FMX styling.

External Stylesheets

```
<link rel="stylesheet" href="https://example.com/styles.css">
```

Stylesheets are downloaded via HTTP and cached. This means you can use external CSS frameworks or shared stylesheets.

Style Blocks

```
Tina4HTMLRender1.HTML.Text :=
  '<style>' +
  '  .card { border: 1px solid #e0e0e0; border-radius: 8px; padding: 16px; margin: 8px; }' +
  '  .card h3 { margin: 0 0 8px 0; color: #333; }' +
  '  .card p { color: #666; font-size: 14px; }' +
  '  .status-active { color: green; font-weight: bold; }' +
  '  .status-inactive { color: red; }' +
  '</style>' +
  '<div class="card">' +
  '  <h3>User Dashboard</h3>' +
  '  <p>Status: <span class="status-active">Active</span></p>' +
  '</div>';
```

Inline Styles

```
<div style="background-color: #f9f9f9; padding: 20px; border-radius: 4px;">
  <p style="font-size: 16px; color: #333;">Inline styled content.</p>
</div>
```

Selector Support

- **Tag selectors**: `h1`, `p`, `div`
- **Class selectors**: `.card`, `.btn`
- **ID selectors**: `#header`, `#main`
- **Combined selectors**: `div.card`, `p.highlight`
- **Specificity-based cascade**: more specific selectors override less specific ones

Custom Properties (CSS Variables)

```
Tina4HTMLRender1.HTML.Text :=
  '<style>' +
  '  :root { --primary: #2563eb; --text: #333; --bg: #f8f9fa; }' +
  '  .header { color: var(--primary); background: var(--bg); padding: 16px; }' +
  '  .body { color: var(--text); padding: 16px; }' +
  '</style>' +
  '<div class="header"><h2>Dashboard</h2></div>' +
  '<div class="body"><p>Content styled with CSS variables.</p></div>';
```

Supported CSS Properties

Category	Properties
Box model	`margin`, `padding`, `border`, `border-radius`, `width`, `height`, `min-width`, `max-width`, `min-height`, `max-height`, `box-sizing`, `box-shadow`
Display	`block`, `inline`, `inline-block`, `none`, `table`, `table-row`, `table-cell`, `list-item`
Text	`color`, `font-size`, `font-family`, `font-weight`, `font-style`, `text-align`, `line-height`, `text-decoration`, `text-transform`, `letter-spacing`, `text-indent`, `text-overflow`, `white-space`
Background	`background-color`, `opacity`
Visibility	`visibility`, `overflow`, `display: none`
Bootstrap 5	`.btn` variants, `.form-control`, `.form-check`, `.text-muted` -- fallback styles are built in

---

4. Form Controls

HTML form elements create native FMX controls overlaid on the rendered content. These are real editable controls -- text inputs, checkboxes, radio buttons, dropdowns, and buttons.

```
Tina4HTMLRender1.HTML.Text :=
  '<style>' +
  '  .form-group { margin-bottom: 12px; }' +
  '  label { display: block; margin-bottom: 4px; font-weight: bold; }' +
  '  input, select, textarea { width: 300px; padding: 6px; border: 1px solid #ccc; }' +
  '  .btn { padding: 8px 16px; background: #2563eb; color: white; border: none; cursor: pointer;
  '</style>' +
  '<form name="userForm">' +
```

```
  '  <div class="form-group">' +
  '    <label>Name</label>' +
  '    <input type="text" name="username" id="username" placeholder="Enter your name">' +
  '  </div>' +
  '  <div class="form-group">' +
  '    <label>Email</label>' +
  '    <input type="email" name="email" id="email" placeholder="you@example.com">' +
  '  </div>' +
  '  <div class="form-group">' +
  '    <label>Password</label>' +
  '    <input type="password" name="password" id="password">' +
  '  </div>' +
  '  <div class="form-group">' +
  '    <label>Role</label>' +
  '    <select name="role" id="role">' +
  '      <option value="user">User</option>' +
  '      <option value="admin">Admin</option>' +
  '      <option value="editor">Editor</option>' +
  '    </select>' +
  '  </div>' +
  '  <div class="form-group">' +
  '    <label>Bio</label>' +
  '    <textarea name="bio" id="bio" rows="4"></textarea>' +
  '  </div>' +
  '  <div class="form-group">' +
  '    <input type="checkbox" name="terms" id="terms">' +
  '    <label style="display: inline;">I agree to the terms</label>' +
  '  </div>' +
  '  <button type="submit" class="btn">Submit</button>' +
  '</form>';
```

Supported input types: `text`, `password`, `email`, `radio`, `checkbox`, `submit`, `button`, `reset`, `file`. Plus `textarea`, `select`/`option`, and `button`.

---

5. Events

The renderer fires events for form interactions, element clicks, and link clicks.

OnFormSubmit

Fires when a submit button is clicked. Collects all form data as name=value pairs:

```
procedure TForm1.HTMLRender1FormSubmit(Sender: TObject;
  const FormName: string; FormData: TStrings);
var
  Username, Email, Role: string;
begin
  Username := FormData.Values['username'];
  Email := FormData.Values['email'];
  Role := FormData.Values['role'];

  ShowMessage(Format('Form "%s" submitted. User: %s, Email: %s, Role: %s',
    [FormName, Username, Email, Role]));
end;
```

OnFormControlChange

Fires when any form control's value changes:

```
procedure TForm1.HTMLRender1FormControlChange(Sender: TObject;
  const Name, Value: string);
begin
  // React to real-time changes
  if Name = 'role' then
  begin
    if Value = 'admin' then
      Tina4HTMLRender1.SetElementVisible('adminPanel', True)
    else
      Tina4HTMLRender1.SetElementVisible('adminPanel', False);
  end;
end;
```

OnFormControlClick

Fires when a form control is clicked (useful for buttons that are not submit buttons):

```
procedure TForm1.HTMLRender1FormControlClick(Sender: TObject;
  const Name, Value: string);
begin
  if Name = 'cancelBtn' then
    ClearForm;
end;
```

OnLinkClick

Fires when an anchor tag is clicked. Set `Handled := True` to prevent default navigation:

```
procedure TForm1.HTMLRender1LinkClick(Sender: TObject;
  const AURL: string; var Handled: Boolean);
begin
  if AURL.StartsWith('http') then
  begin
    // Open in system browser instead of navigating
    ShellExecute(0, 'open', PChar(AURL), nil, nil, SW_SHOWNORMAL);
    Handled := True;
  end;
end;
```

Event Reference

Event	Signature	When
`OnFormControlChange`	`(Sender; Name, Value: string)`	Form control value changes
`OnFormControlClick`	`(Sender; Name, Value: string)`	Form control clicked
`OnFormControlEnter`	`(Sender; Name, Value: string)`	Form control gains focus
`OnFormControlExit`	`(Sender; Name, Value: string)`	Form control loses focus
`OnFormSubmit`	`(Sender; FormName: string; FormData: TStrings)`	Submit button clicked
`OnElementClick`	`(Sender; ObjectName, MethodName: string; Params: TStrings)`	onclick RTTI element clicked
`OnLinkClick`	`(Sender; URL: string; var Handled: Boolean)`	Anchor href clicked

---

6. onclick and RTTI -- Calling Pascal from HTML

Any HTML element can call a Pascal method directly using the `onclick` attribute with a special syntax: `onclick="ObjectName:MethodName(params)"`. This bridges HTML events to Delphi code without writing event handlers.

Step 1: Register Your Object

In your form's `OnCreate`, register the Delphi object that will receive calls:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  Tina4HTMLRender1.RegisterObject('App', Self);
end;
```

Step 2: Write the Target Method

The method must be `published` or use `{$M+}` RTTI. Parameters are passed as strings:

```
procedure TForm1.ShowAlert(Message: String);
begin
  ShowMessage(Message);
end;

procedure TForm1.HandleAction(Action: String; ItemId: String);
begin
```

```
  if Action = 'delete' then
    DeleteItem(ItemId)
  else if Action = 'edit' then
    EditItem(ItemId);
end;
```

Step 3: Call from HTML

```
Tina4HTMLRender1.HTML.Text :=
  '<button onclick="App:ShowAlert(''Hello from HTML!'')">Say Hello</button>' +
  '<button onclick="App:HandleAction(''edit'', ''42'')">Edit Item 42</button>' +
  '<button onclick="App:HandleAction(''delete'', ''42'')">Delete Item 42</button>';
```

Dynamic Parameter Expressions

The onclick handler supports dynamic expressions, not just string literals:

Expression	Resolves To
`'literal'` or `"literal"`	String literal
`123`	Numeric literal
`this.value`	Value of the clicked element
`this.id`	ID of the clicked element
`document.getElementById('id').value`	Value of element by ID
`document.getElementById('id').<attr>`	Any attribute of element by ID

Example with dynamic values:

```
Tina4HTMLRender1.HTML.Text :=
  '<input type="text" id="nameInput" placeholder="Type your name">' +
  '<button onclick="App:Greet(document.getElementById(''nameInput'').value)">' +
  '  Greet</button>';

procedure TForm1.Greet(Name: String);
begin
  ShowMessage('Hello, ' + Name + '!');
end;
```

---

7. DOM Manipulation

Modify rendered HTML elements from Delphi code at runtime. Update text, change styles, show/hide elements, enable/disable controls -- all without re-rendering the entire HTML.

Get and Set Values

```
// Set a form input's value
Tina4HTMLRender1.SetElementValue('emailInput', 'user@example.com');

// Read a form input's value
var Email := Tina4HTMLRender1.GetElementValue('emailInput');
```

Enable/Disable Controls

```
// Disable the submit button until the form is valid
Tina4HTMLRender1.SetElementEnabled('submitBtn', False);

// Enable it when validation passes
Tina4HTMLRender1.SetElementEnabled('submitBtn', True);
```

Show/Hide Elements

```
// Show an error message
Tina4HTMLRender1.SetElementVisible('errorMsg', True);

// Hide the loading spinner
Tina4HTMLRender1.SetElementVisible('spinner', False);
```

Change Text Content

```
Tina4HTMLRender1.SetElementText('statusLabel', 'Processing...');
Tina4HTMLRender1.SetElementText('recordCount', IntToStr(Count) + ' records');
```

Change Styles

```
Tina4HTMLRender1.SetElementStyle('statusLabel', 'color', 'green');
Tina4HTMLRender1.SetElementStyle('alertBox', 'background-color', '#fee2e2');
Tina4HTMLRender1.SetElementStyle('alertBox', 'border', '1px solid #ef4444');
```

Set Attributes

```
Tina4HTMLRender1.SetElementAttribute('myImage', 'src', 'https://example.com/new-photo.jpg');
Tina4HTMLRender1.SetElementAttribute('myLink', 'href', '/new-page');

// Changing class or style triggers relayout
Tina4HTMLRender1.SetElementAttribute('myDiv', 'class', 'card highlighted');
```

Force Refresh

```
// After multiple DOM changes, force a full re-layout
Tina4HTMLRender1.RefreshElement('mainContent');
```

DOM Method Reference

Method	Description
`GetElementById(Id)`	Returns the `THTMLTag` for the element
`GetElementValue(Id)`	Gets the live value from a native control or DOM attribute
`SetElementValue(Id, Value)`	Sets the value on native controls and DOM
`SetElementAttribute(Id, Attr, Value)`	Sets any attribute; triggers relayout for `class`/`style`
`SetElementEnabled(Id, Enabled)`	Enables/disables native controls
`SetElementVisible(Id, Visible)`	Shows/hides elements via `display:none`
`SetElementText(Id, Text)`	Updates inner text content
`SetElementStyle(Id, Prop, Value)`	Sets an inline style property
`RefreshElement(Id)`	Forces a full re-layout and repaint

---

8. Image Loading and Caching

Images referenced in `<img>` tags are downloaded asynchronously via HTTP. Once downloaded, they are cached to disk so subsequent renders are instant.

```
// Enable caching and set the cache directory
Tina4HTMLRender1.CacheEnabled := True;
Tina4HTMLRender1.CacheDir := 'C:\MyApp\cache';

// Images load in the background and appear when ready
Tina4HTMLRender1.HTML.Text :=
  '<div style="display: inline-block; margin: 8px;">' +
  '  <img src="https://picsum.photos/200/150" width="200" height="150">' +
  '  <p style="text-align: center;">Random photo</p>' +
  '</div>';
```

The first load downloads the image. Subsequent loads read from `C:\MyApp\cache`. Without a cache directory, images are re-downloaded every time.

---

9. Complete Example: Login Form with Validation

A login form with username and password fields, client-side validation, error display, and a submit handler that calls a REST API.

```
unit LoginForm;

interface
```

```
uses
  System.SysUtils, System.Classes, System.JSON,
  FMX.Types, FMX.Controls, FMX.Forms, FMX.StdCtrls,
  Tina4HTMLRender, Tina4REST;

type
  TfrmLogin = class(TForm)
    HTMLRender1: TTina4HTMLRender;
    restAPI: TTina4REST;
    procedure FormCreate(Sender: TObject);
    procedure HTMLRender1FormSubmit(Sender: TObject;
      const FormName: string; FormData: TStrings);
  private
    procedure RenderLoginPage;
    procedure ShowError(const Msg: string);
    procedure ShowSuccess;
  published
    procedure ForgotPassword(Action: String);
  end;

var
  frmLogin: TfrmLogin;

implementation

{$R *.fmx}

procedure TfrmLogin.FormCreate(Sender: TObject);
begin
  restAPI.BaseUrl := 'https://api.example.com';
  HTMLRender1.RegisterObject('Login', Self);
  RenderLoginPage;
end;

procedure TfrmLogin.RenderLoginPage;
begin
  HTMLRender1.HTML.Text :=
    '<style>' +
    '  body { font-family: Arial, sans-serif; background: #f5f5f5; }' +
    '  .login-card { max-width: 400px; margin: 40px auto; padding: 32px;' +
    '    background: white; border-radius: 8px; box-shadow: 0 2px 8px rgba(0,0,0,0.1); }' +
    '  .login-card h2 { margin: 0 0 24px 0; text-align: center; color: #333; }' +
    '  .form-group { margin-bottom: 16px; }' +
    '  .form-group label { display: block; margin-bottom: 4px; font-weight: bold;' +
    '    color: #555; font-size: 14px; }' +
    '  .form-group input { width: 100%; padding: 10px; border: 1px solid #ddd;' +
    '    border-radius: 4px; font-size: 14px; }' +
    '  .btn-login { width: 100%; padding: 12px; background: #2563eb; color: white;' +
    '    border: none; border-radius: 4px; font-size: 16px; font-weight: bold; }' +
    '  .error { background: #fee2e2; color: #dc2626; padding: 10px; border-radius: 4px;' +
    '    margin-bottom: 16px; display: none; }' +
    '  .success { background: #dcfce7; color: #16a34a; padding: 10px; border-radius: 4px;' +
    '    margin-bottom: 16px; display: none; }' +
    '  .forgot { text-align: center; margin-top: 16px; }' +
    '  .forgot a { color: #2563eb; font-size: 14px; }' +
    '</style>' +
    '<div class="login-card">' +
    '  <h2>Sign In</h2>' +
    '  <div class="error" id="errorBox">Error message here</div>' +
```

```
    '  <div class="success" id="successBox">Login successful!</div>' +
    '  <form name="loginForm">' +
    '    <div class="form-group">' +
    '      <label>Email</label>' +
    '      <input type="email" name="email" id="email" placeholder="you@example.com">' +
    '    </div>' +
    '    <div class="form-group">' +
    '      <label>Password</label>' +
    '      <input type="password" name="password" id="password" placeholder="Enter password">' +
    '    </div>' +
    '    <button type="submit" class="btn-login" id="btnSubmit">Sign In</button>' +
    '  </form>' +
    '  <div class="forgot">' +
    '    <span onclick="Login:ForgotPassword(''reset'')" ' +
    '      style="color: #2563eb; cursor: pointer;">Forgot your password?</span>' +
    '  </div>' +
    '</div>';
end;

procedure TfrmLogin.HTMLRender1FormSubmit(Sender: TObject;
  const FormName: string; FormData: TStrings);
var
  Email, Password: string;
  StatusCode: Integer;
  Response: TJSONObject;
begin
  if FormName <> 'loginForm' then Exit;

  Email := FormData.Values['email'];
  Password := FormData.Values['password'];

  // Client-side validation
  if Email.Trim = '' then
  begin
    ShowError('Email is required');
    Exit;
  end;
  if Password.Trim = '' then
  begin
    ShowError('Password is required');
    Exit;
  end;
  if not Email.Contains('@') then
  begin
    ShowError('Please enter a valid email address');
    Exit;
  end;

  // Disable the button while processing
  HTMLRender1.SetElementEnabled('btnSubmit', False);
  HTMLRender1.SetElementText('btnSubmit', 'Signing in...');

  // Call the API
  Response := restAPI.Post(StatusCode, '/auth/login', '',
    Format('{"email": "%s", "password": "%s"}', [Email, Password]));
  try
    if StatusCode = 200 then
    begin
      var Token := Response.GetValue<String>('token');
```

```
      restAPI.SetBearer(Token);
      ShowSuccess;
    end
    else
    begin
      ShowError('Invalid email or password');
    end;
  finally
    Response.Free;
    HTMLRender1.SetElementEnabled('btnSubmit', True);
    HTMLRender1.SetElementText('btnSubmit', 'Sign In');
  end;
end;

procedure TfrmLogin.ShowError(const Msg: string);
begin
  HTMLRender1.SetElementVisible('successBox', False);
  HTMLRender1.SetElementText('errorBox', Msg);
  HTMLRender1.SetElementVisible('errorBox', True);
end;

procedure TfrmLogin.ShowSuccess;
begin
  HTMLRender1.SetElementVisible('errorBox', False);
  HTMLRender1.SetElementVisible('successBox', True);
end;

procedure TfrmLogin.ForgotPassword(Action: String);
begin
  ShowMessage('Forgot password flow: ' + Action);
end;

end.
```

---

10. Complete Example: Interactive Dashboard

A dashboard with stats cards, a data table, and action buttons that call Pascal methods. This demonstrates combining styled HTML, dynamic data, and RTTI-based event handling.

```
unit Dashboard;

interface

uses
  System.SysUtils, System.Classes, System.JSON,
  FMX.Types, FMX.Controls, FMX.Forms, FMX.StdCtrls,
  FireDAC.Comp.Client,
  Tina4HTMLRender, Tina4REST, Tina4Core;

type
  TfrmDashboard = class(TForm)
    HTMLRender1: TTina4HTMLRender;
    restAPI: TTina4REST;
    procedure FormCreate(Sender: TObject);
  private
    procedure RenderDashboard;
```

```
    function BuildStatsCards: string;
    function BuildUserTable: string;
  published
    procedure ViewUser(UserId: String);
    procedure DeleteUser(UserId: String);
    procedure RefreshData(Action: String);
  end;

var
  frmDashboard: TfrmDashboard;

implementation

{$R *.fmx}

procedure TfrmDashboard.FormCreate(Sender: TObject);
begin
  restAPI.BaseUrl := 'https://api.example.com/v1';
  HTMLRender1.RegisterObject('Dashboard', Self);
  RenderDashboard;
end;

procedure TfrmDashboard.RenderDashboard;
begin
  HTMLRender1.HTML.Text :=
    '<style>' +
    '  * { box-sizing: border-box; }' +
    '  body { font-family: Arial, sans-serif; padding: 20px; background: #f0f2f5; }' +
    '  h1 { color: #1a1a2e; margin-bottom: 24px; }' +
    '  .stats { display: inline-block; width: 100%; margin-bottom: 24px; }' +
    '  .stat-card { display: inline-block; width: 22%; margin-right: 2%;' +
    '    background: white; border-radius: 8px; padding: 20px;' +
    '    box-shadow: 0 1px 3px rgba(0,0,0,0.1); }' +
    '  .stat-card h3 { margin: 0; color: #888; font-size: 12px; text-transform: uppercase; }' +
    '  .stat-card .value { font-size: 28px; font-weight: bold; color: #1a1a2e; margin: 8px 0; }'
    '  .stat-card .change { font-size: 12px; color: #16a34a; }' +
    '  table { width: 100%; border-collapse: collapse; background: white;' +
    '    border-radius: 8px; overflow: hidden; box-shadow: 0 1px 3px rgba(0,0,0,0.1); }' +
    '  th { background: #f8f9fa; padding: 12px; text-align: left;' +
    '    font-size: 12px; color: #888; text-transform: uppercase; border-bottom: 2px solid #e0e0
    '  td { padding: 12px; border-bottom: 1px solid #f0f0f0; color: #333; }' +
    '  .btn-sm { padding: 4px 12px; border: none; border-radius: 4px;' +
    '    font-size: 12px; cursor: pointer; margin-right: 4px; }' +
    '  .btn-view { background: #dbeafe; color: #2563eb; }' +
    '  .btn-delete { background: #fee2e2; color: #dc2626; }' +
    '  .btn-refresh { padding: 8px 16px; background: #2563eb; color: white;' +
    '    border: none; border-radius: 4px; margin-bottom: 16px; }' +
    '  .toolbar { margin-bottom: 16px; }' +
    '</style>' +
    '<h1>Admin Dashboard</h1>' +
    '<div class="toolbar">' +
    '  <button class="btn-refresh" ' +
    '    onclick="Dashboard:RefreshData(''all'')">Refresh Data</button>' +
    '</div>' +
    BuildStatsCards +
    '<h2 style="color: #1a1a2e; margin: 24px 0 16px;">Recent Users</h2>' +
    BuildUserTable;
end;
```

```
function TfrmDashboard.BuildStatsCards: string;
begin
  Result :=
    '<div class="stats">' +
    '  <div class="stat-card">' +
    '    <h3>Total Users</h3>' +
    '    <div class="value" id="totalUsers">1,234</div>' +
    '    <div class="change">+12% this month</div>' +
    '  </div>' +
    '  <div class="stat-card">' +
    '    <h3>Active Sessions</h3>' +
    '    <div class="value" id="activeSessions">56</div>' +
    '    <div class="change">+3% this hour</div>' +
    '  </div>' +
    '  <div class="stat-card">' +
    '    <h3>Revenue</h3>' +
    '    <div class="value" id="revenue">$48,290</div>' +
    '    <div class="change">+8% this week</div>' +
    '  </div>' +
    '  <div class="stat-card">' +
    '    <h3>Orders</h3>' +
    '    <div class="value" id="orders">389</div>' +
    '    <div class="change">+5% today</div>' +
    '  </div>' +
    '</div>';
end;

function TfrmDashboard.BuildUserTable: string;
begin
  Result :=
    '<table>' +
    '  <thead>' +
    '    <tr><th>ID</th><th>Name</th><th>Email</th><th>Status</th><th>Actions</th></tr>' +
    '  </thead>' +
    '  <tbody>' +
    '    <tr>' +
    '      <td>1</td><td>Alice Smith</td><td>alice@example.com</td>' +
    '      <td style="color: green;">Active</td>' +
    '      <td>' +
    '        <button class="btn-sm btn-view" onclick="Dashboard:ViewUser(''1'')">View</button>'
    '        <button class="btn-sm btn-delete" onclick="Dashboard:DeleteUser(''1'')">Delete</but
    '      </td>' +
    '    </tr>' +
    '    <tr>' +
    '      <td>2</td><td>Bob Johnson</td><td>bob@example.com</td>' +
    '      <td style="color: green;">Active</td>' +
    '      <td>' +
    '        <button class="btn-sm btn-view" onclick="Dashboard:ViewUser(''2'')">View</button>'
    '        <button class="btn-sm btn-delete" onclick="Dashboard:DeleteUser(''2'')">Delete</but
    '      </td>' +
    '    </tr>' +
    '    <tr>' +
    '      <td>3</td><td>Carol Williams</td><td>carol@example.com</td>' +
    '      <td style="color: red;">Inactive</td>' +
    '      <td>' +
    '        <button class="btn-sm btn-view" onclick="Dashboard:ViewUser(''3'')">View</button>'
    '        <button class="btn-sm btn-delete" onclick="Dashboard:DeleteUser(''3'')">Delete</but
    '      </td>' +
    '    </tr>' +
```

```
    '  </tbody>' +
    '</table>';
end;

procedure TfrmDashboard.ViewUser(UserId: String);
begin
  ShowMessage('Viewing user ' + UserId);
  // In a real app: navigate to user detail page or show a modal
end;

procedure TfrmDashboard.DeleteUser(UserId: String);
begin
  ShowMessage('Delete user ' + UserId + '?');
  // In a real app: confirm then call DELETE /users/{id}
end;

procedure TfrmDashboard.RefreshData(Action: String);
begin
  // Refresh stats via DOM manipulation -- no full re-render needed
  HTMLRender1.SetElementText('totalUsers', '1,256');
  HTMLRender1.SetElementText('activeSessions', '61');
  HTMLRender1.SetElementText('revenue', '$49,100');
  HTMLRender1.SetElementText('orders', '402');
  ShowMessage('Dashboard data refreshed');
end;

end.
```

---

11. Exercise: Contact Form

Build a contact form with name, email, and message fields. Validate all fields before submission. Submit the data to a REST API.

Requirements

- Drop a `TTina4HTMLRender` on a form
- Create an HTML form with: name (text), email (email), subject (select dropdown), message (textarea)
- Add validation: all fields required, email must contain `@`, message minimum 10 characters
- Show validation errors inline (red text below each field)
- On successful validation, POST the form data to `/contact` as JSON
- Show a success message after submission

Solution

```
unit ContactForm;

interface

uses
  System.SysUtils, System.Classes, System.JSON,
  FMX.Types, FMX.Controls, FMX.Forms,
  Tina4HTMLRender, Tina4REST;

type
  TfrmContact = class(TForm)
    HTMLRender1: TTina4HTMLRender;
    restAPI: TTina4REST;
    procedure FormCreate(Sender: TObject);
    procedure HTMLRender1FormSubmit(Sender: TObject;
      const FormName: string; FormData: TStrings);
  private
    procedure RenderForm;
    function Validate(FormData: TStrings): Boolean;
  end;

var
  frmContact: TfrmContact;

implementation

{$R *.fmx}

procedure TfrmContact.FormCreate(Sender: TObject);
begin
  restAPI.BaseUrl := 'https://api.example.com';
  RenderForm;
end;

procedure TfrmContact.RenderForm;
begin
  HTMLRender1.HTML.Text :=
    '<style>' +
    '  .container { max-width: 500px; margin: 20px auto; padding: 24px;' +
    '    background: white; border-radius: 8px; box-shadow: 0 2px 4px rgba(0,0,0,0.1); }' +
    '  h2 { margin: 0 0 20px 0; color: #333; }' +
    '  .field { margin-bottom: 16px; }' +
    '  .field label { display: block; margin-bottom: 4px; font-weight: bold; font-size: 14px; }'
    '  .field input, .field select, .field textarea ' +
    '    { width: 100%; padding: 8px; border: 1px solid #ccc; border-radius: 4px; }' +
    '  .field textarea { height: 100px; }' +
    '  .error-text { color: #dc2626; font-size: 12px; display: none; margin-top: 4px; }' +
    '  .btn-submit { padding: 10px 24px; background: #2563eb; color: white;' +
    '    border: none; border-radius: 4px; font-size: 14px; }' +
    '  .success-msg { background: #dcfce7; color: #16a34a; padding: 12px;' +
    '    border-radius: 4px; display: none; margin-bottom: 16px; }' +
    '</style>' +
    '<div class="container">' +
    '  <h2>Contact Us</h2>' +
    '  <div class="success-msg" id="successMsg">Thank you! Your message has been sent.</div>' +
    '  <form name="contactForm">' +
    '    <div class="field">' +
```

```
    '      <label>Name</label>' +
    '      <input type="text" name="name" id="name" placeholder="Your full name">' +
    '      <div class="error-text" id="nameError">Name is required</div>' +
    '    </div>' +
    '    <div class="field">' +
    '      <label>Email</label>' +
    '      <input type="email" name="email" id="email" placeholder="you@example.com">' +
    '      <div class="error-text" id="emailError">Valid email is required</div>' +
    '    </div>' +
    '    <div class="field">' +
    '      <label>Subject</label>' +
    '      <select name="subject" id="subject">' +
    '        <option value="">Select a subject</option>' +
    '        <option value="support">Technical Support</option>' +
    '        <option value="sales">Sales Inquiry</option>' +
    '        <option value="feedback">Feedback</option>' +
    '      </select>' +
    '      <div class="error-text" id="subjectError">Please select a subject</div>' +
    '    </div>' +
    '    <div class="field">' +
    '      <label>Message</label>' +
    '      <textarea name="message" id="message" placeholder="Your message (min 10 chars)"></tex
    '      <div class="error-text" id="messageError">Message must be at least 10 characters</div
    '    </div>' +
    '    <button type="submit" class="btn-submit" id="btnSubmit">Send Message</button>' +
    '  </form>' +
    '</div>';
end;

procedure TfrmContact.HTMLRender1FormSubmit(Sender: TObject;
  const FormName: string; FormData: TStrings);
var
  StatusCode: Integer;
  Response: TJSONObject;
begin
  if FormName <> 'contactForm' then Exit;

  // Hide previous errors
  HTMLRender1.SetElementVisible('nameError', False);
  HTMLRender1.SetElementVisible('emailError', False);
  HTMLRender1.SetElementVisible('subjectError', False);
  HTMLRender1.SetElementVisible('messageError', False);
  HTMLRender1.SetElementVisible('successMsg', False);

  if not Validate(FormData) then Exit;

  // Submit to API
  HTMLRender1.SetElementEnabled('btnSubmit', False);
  HTMLRender1.SetElementText('btnSubmit', 'Sending...');

  Response := restAPI.Post(StatusCode, '/contact', '',
    Format('{"name": "%s", "email": "%s", "subject": "%s", "message": "%s"}',
      [FormData.Values['name'], FormData.Values['email'],
       FormData.Values['subject'], FormData.Values['message']]));
  try
    if StatusCode in [200, 201] then
    begin
      HTMLRender1.SetElementVisible('successMsg', True);
      // Clear the form
```

```
      HTMLRender1.SetElementValue('name', '');
      HTMLRender1.SetElementValue('email', '');
      HTMLRender1.SetElementValue('message', '');
    end
    else
      ShowMessage('Submission failed: HTTP ' + StatusCode.ToString);
  finally
    Response.Free;
    HTMLRender1.SetElementEnabled('btnSubmit', True);
    HTMLRender1.SetElementText('btnSubmit', 'Send Message');
  end;
end;

function TfrmContact.Validate(FormData: TStrings): Boolean;
begin
  Result := True;

  if FormData.Values['name'].Trim = '' then
  begin
    HTMLRender1.SetElementVisible('nameError', True);
    Result := False;
  end;

  var Email := FormData.Values['email'].Trim;
  if (Email = '') or (not Email.Contains('@')) then
  begin
    HTMLRender1.SetElementVisible('emailError', True);
    Result := False;
  end;

  if FormData.Values['subject'].Trim = '' then
  begin
    HTMLRender1.SetElementVisible('subjectError', True);
    Result := False;
  end;

  if FormData.Values['message'].Trim.Length < 10 then
  begin
    HTMLRender1.SetElementVisible('messageError', True);
    Result := False;
  end;
end;

end.
```

---

12. Common Gotchas

Forgetting to Set Cache Directory for Images

Symptom: Images load the first time, but every subsequent launch re-downloads them. Or images do not appear at all.

Fix: Set `CacheEnabled := True` and `CacheDir` to a writable directory before setting the HTML:

```
HTMLRender1.CacheEnabled := True;
```

```
HTMLRender1.CacheDir := TPath.Combine(TPath.GetDocumentsPath, 'AppCache');
ForceDirectories(HTMLRender1.CacheDir);
```

RTTI Method Not Found

Symptom: Clicking an `onclick` element does nothing, or raises an access violation.

Fix: Ensure the target method is `published` (or the class has `{$M+}` RTTI). Ensure `RegisterObject` was called with the correct object name. Ensure the `onclick` format is exactly `ObjectName:MethodName(params)`:

```
// Registration
HTMLRender1.RegisterObject('MyApp', Self);

// HTML must match the registered name
onclick="MyApp:DoSomething('param')"  // Correct
onclick="Form1:DoSomething('param')"  // Wrong name -- will not find the object
```

Form Control Name Matching

Symptom: `FormData.Values['username']` returns empty string even though the user typed in the field.

Fix: The `name` attribute in the HTML must match exactly. Case matters:

```
<input type="text" name="userName">   <!-- FormData.Values['userName'] -->
<input type="text" name="username">   <!-- FormData.Values['username'] -->
```

Escaped Quotes in HTML Strings

Symptom: Compilation error or garbled HTML.

Fix: In Delphi string literals, use doubled single quotes `''` for apostrophes inside HTML attributes:

```
// WRONG -- compilation error
HTML.Text := '<button onclick="App:Do('param')">Click</button>';

// CORRECT -- doubled single quotes
HTML.Text := '<button onclick="App:Do(''param'')">Click</button>';
```

Summary

What	How
Basic rendering	`HTMLRender1.HTML.Text := '<h1>Hello</h1>'`
External CSS	`<link rel="stylesheet" href="...">`
Style blocks	`<style>.card { ... }</style>`
Inline styles	`style="color: blue;"`
CSS variables	`var(--primary)` with `:root`
Form controls	`<input>`, `<select>`, `<textarea>`, `<button>`
Form submit	`OnFormSubmit` event -- `FormData.Values['name']`
RTTI onclick	`onclick="ObjName:Method(params)"` + `RegisterObject`
DOM: set value	`SetElementValue('id', 'value')`
DOM: show/hide	`SetElementVisible('id', True/False)`
DOM: enable	`SetElementEnabled('id', True/False)`
DOM: text	`SetElementText('id', 'text')`
DOM: style	`SetElementStyle('id', 'prop', 'value')`
Image caching	`CacheEnabled := True` + `CacheDir := '...'`

Page Navigation

A Single-Page App Inside Your Desktop App

Your application has a dashboard, a user management page, a settings page, and a reports page. In a traditional Delphi app, you create four forms and show/hide them. With `TTina4HTMLPages`, you create four pages inside one HTML renderer and navigate between them with anchor links -- the same way a single-page web app works.

No form switching. No component visibility toggling. No frame loading. Just set the HTML for each page, link them to a renderer, and navigate with `<a href="#pagename">` or programmatic calls.

---

1. TTina4HTMLPages Basics

`TTina4HTMLPages` manages a collection of pages and renders them through a `TTina4HTMLRender`. Each page has a name, HTML content (or Twig template content), and an optional "is default" flag.

Setup

Drop two components on your form:

- **TTina4HTMLRender** (name: `HTMLRender1`) -- set `Align` to `Client`
- **TTina4HTMLPages** (name: `HTMLPages1`)

Link them:

```
HTMLPages1.Renderer := HTMLRender1;
```

That is the entire setup. The pages component controls what the renderer displays.

---

2. Creating Pages at Design Time

Double-click `HTMLPages1` in the form designer to open the collection editor. Click "Add" to create pages:

Page 1:

- `PageName`: `home`
- `IsDefault`: `True`
- `HTMLContent`: `<h1>Home</h1><p>Welcome to the app.</p><a href="#settings">Go to Settings</a>`

Page 2:

- `PageName`: `settings`

- `IsDefault`: `False`
- `HTMLContent`: `<h1>Settings</h1><p>Configure your preferences.</p><a href="#home">Back to Home</a>`

Run the app. The home page appears (it has `IsDefault = True`). Click "Go to Settings". The settings page appears. Click "Back to Home". You are back. No code written.

---

3. Creating Pages at Runtime

For dynamic apps where pages are built from data or configuration:

```
procedure TForm1.FormCreate(Sender: TObject);
var
  Page: TTina4Page;
begin
  HTMLPages1.Renderer := HTMLRender1;

  // Home page
  Page := HTMLPages1.Pages.Add;
  Page.PageName := 'home';
  Page.IsDefault := True;
  Page.HTMLContent.Text :=
    '<style>' +
    '  .nav { background: #1a1a2e; padding: 12px 20px; }' +
    '  .nav a { color: white; margin-right: 16px; text-decoration: none; }' +
    '  .content { padding: 20px; }' +
    '</style>' +
    '<div class="nav">' +
    '  <a href="#home">Home</a>' +
    '  <a href="#users">Users</a>' +
    '  <a href="#settings">Settings</a>' +
    '</div>' +
    '<div class="content">' +
    '  <h1>Dashboard</h1>' +
    '  <p>Welcome back. Select a section from the navigation above.</p>' +
    '</div>';

  // Users page
  Page := HTMLPages1.Pages.Add;
  Page.PageName := 'users';
  Page.HTMLContent.Text :=
    '<div class="nav">' +
    '  <a href="#home">Home</a>' +
    '  <a href="#users">Users</a>' +
    '  <a href="#settings">Settings</a>' +
    '</div>' +
    '<div class="content">' +
    '  <h1>Users</h1>' +
    '  <p>User management goes here.</p>' +
    '</div>';

  // Settings page
  Page := HTMLPages1.Pages.Add;
  Page.PageName := 'settings';
  Page.HTMLContent.Text :=
```

```
    '<div class="nav">' +
    '  <a href="#home">Home</a>' +
    '  <a href="#users">Users</a>' +
    '  <a href="#settings">Settings</a>' +
    '</div>' +
    '<div class="content">' +
    '  <h1>Settings</h1>' +
    '  <p>Application settings go here.</p>' +
    '</div>';
end;
```

---

4. Setting the Default Page

The page with `IsDefault := True` renders automatically when the component initializes. If no page has `IsDefault` set, nothing renders until you navigate programmatically.

```
Page.IsDefault := True;
```

Only one page should have `IsDefault = True`. If multiple pages have it set, the first one in the collection wins.

---

5. Navigation via Anchor Links

The most natural way to navigate is with HTML anchor tags. The `href` value maps to a page name:

value	Maps to PageName
`#dashboard`	`dashboard`
`/settings`	`settings`
`about`	`about`

The renderer intercepts link clicks and delegates to the pages component. The leading `#` or `/` is stripped to match the `PageName`.

```
<a href="#home">Home</a>
<a href="#users">Users</a>
<a href="/settings">Settings</a>
```

All three formats work. The `#` prefix is conventional for SPA-style navigation. The `/` prefix works the same way. A bare name also works.

---

6. Programmatic Navigation

Navigate from Delphi code without user interaction:

```
HTMLPages1.NavigateTo('settings');
```

This is useful for:

- Redirecting after a successful action (e.g., login redirects to dashboard)
- Responding to button clicks that are not anchor tags
- Implementing back/forward functionality
- Conditional navigation based on business logic

```
procedure TForm1.OnLoginSuccess;
begin
  HTMLPages1.NavigateTo('dashboard');
end;

procedure TForm1.OnLogout;
begin
  HTMLPages1.NavigateTo('login');
end;
```

Reading the Active Page

```
var CurrentPage := HTMLPages1.ActivePage;
if CurrentPage = 'settings' then
  ShowMessage('You are on the settings page');
```

---

7. Page Properties

Each `TTina4Page` in the collection has these properties:

Property	Type	Description
`PageName`	`string`	Unique name used as navigation target
`TwigContent`	`TStringList`	Twig template source (rendered via TTina4Twig)
`HTMLContent`	`TStringList`	Raw HTML (used when TwigContent is empty)
`IsDefault`	`Boolean`	If `True`, this page is shown on startup

Priority rule: If `TwigContent` is not empty, it is rendered through the Twig engine and the result replaces `HTMLContent`. If `TwigContent` is empty, `HTMLContent` is used directly.

---

8. Component Properties

`TTina4HTMLPages` itself has these properties:

Property	Type	Description

`Pages`	`TTina4PageCollection`	Collection of pages (design-time editable)
`Renderer`	`TTina4HTMLRender`	The HTML renderer that displays the active page
`ActivePage`	`string`	Name of the currently displayed page (read/write)
`TwigTemplatePath`	`string`	Base path for Twig `{% include %}` / `{% extends %}`

---

9. Events

OnBeforeNavigate

Fires before navigation occurs. Set `Allow := False` to cancel the navigation:

```
procedure TForm1.HTMLPages1BeforeNavigate(Sender: TObject;
  const FromPage, ToPage: string; var Allow: Boolean);
begin
  // Prevent navigation to admin page if not authenticated
  if (ToPage = 'admin') and (not FIsAuthenticated) then
  begin
    Allow := False;
    ShowMessage('You must log in to access the admin page.');
    HTMLPages1.NavigateTo('login');
  end;
end;
```

OnAfterNavigate

Fires after the new page has been rendered. Use it for post-navigation setup like loading data:

```
procedure TForm1.HTMLPages1AfterNavigate(Sender: TObject);
begin
  if HTMLPages1.ActivePage = 'users' then
    LoadUserData;
  if HTMLPages1.ActivePage = 'dashboard' then
    RefreshStats;
end;
```

---

10. Using Twig Templates in Pages

Pages can use Twig templates for dynamic content. Set variables with `SetTwigVariable` and use Twig syntax in `TwigContent`:

```
HTMLPages1.SetTwigVariable('userName', 'Alice');
HTMLPages1.SetTwigVariable('userRole', 'Admin');
HTMLPages1.SetTwigVariable('notificationCount', '3');
```

```
var Page := HTMLPages1.Pages.Add;
Page.PageName := 'dashboard';
Page.TwigContent.Text :=
  '<div class="header">' +
  '  <h1>Welcome, {{ userName }}</h1>' +
  '  <span>Role: {{ userRole }}</span>' +
  '  {% if notificationCount > 0 %}' +
  '    <span class="badge">{{ notificationCount }} new</span>' +
  '  {% endif %}' +
  '</div>';
```

File-Based Templates

For complex pages, use files with `{% include %}` and `{% extends %}`:

```
HTMLPages1.TwigTemplatePath := 'C:\MyApp\templates';

var Page := HTMLPages1.Pages.Add;
Page.PageName := 'report';
Page.TwigContent.LoadFromFile('C:\MyApp\templates\report.html');
```

report.html:

```
{% extends 'layout.html' %}

{% block title %}Monthly Report{% endblock %}

{% block content %}
  <h1>Report for {{ month }}</h1>
  <table>
    {% for row in data %}
    <tr>
      <td>{{ row.name }}</td>
      <td>{{ row.value }}</td>
    </tr>
    {% endfor %}
  </table>
{% endblock %}
```

layout.html:

```
<html>
<head>
  <style>
    body { font-family: Arial, sans-serif; }
    .nav { background: #333; padding: 12px; }
    .nav a { color: white; margin-right: 16px; }
  </style>
</head>
<body>
  <div class="nav">
    <a href="#home">Home</a>
    <a href="#report">Report</a>
    <a href="#settings">Settings</a>
  </div>
  <div style="padding: 20px;">
    <title>{% block title %}{% endblock %}</title>
    {% block content %}{% endblock %}
  </div>
</body>
```

```
</html>
```

---

11. Complete Example: Multi-Page Admin App

A full admin application with a sidebar menu, dashboard page, users page with a data table, and settings page with a form.

```
unit AdminApp;

interface

uses
  System.SysUtils, System.Classes, System.JSON,
  FMX.Types, FMX.Controls, FMX.Forms, FMX.StdCtrls,
  Tina4HTMLRender, Tina4HTMLPages, Tina4REST;

type
  TfrmAdmin = class(TForm)
    HTMLRender1: TTina4HTMLRender;
    HTMLPages1: TTina4HTMLPages;
    restAPI: TTina4REST;
    procedure FormCreate(Sender: TObject);
    procedure HTMLPages1AfterNavigate(Sender: TObject);
    procedure HTMLRender1FormSubmit(Sender: TObject;
      const FormName: string; FormData: TStrings);
  private
    function BuildLayout(const ActivePage, Content: string): string;
    function GetSidebar(const ActivePage: string): string;
    function GetStyles: string;
    procedure SetupPages;
  published
    procedure NavTo(PageName: String);
    procedure EditUser(UserId: String);
    procedure SaveSettings(Action: String);
  end;

var
  frmAdmin: TfrmAdmin;

implementation

{$R *.fmx}

procedure TfrmAdmin.FormCreate(Sender: TObject);
begin
  restAPI.BaseUrl := 'https://api.example.com/v1';
  HTMLPages1.Renderer := HTMLRender1;
  HTMLRender1.RegisterObject('Admin', Self);
  SetupPages;
end;

function TfrmAdmin.GetStyles: string;
begin
  Result :=
    '<style>' +
    '  * { box-sizing: border-box; margin: 0; padding: 0; }' +
```

```
    '  body { font-family: Arial, sans-serif; display: inline-block; width: 100%; }' +
    '  .sidebar { display: inline-block; width: 200px; background: #1a1a2e;' +
    '    min-height: 600px; padding: 20px 0; vertical-align: top; }' +
    '  .sidebar h3 { color: #7c8db5; padding: 0 20px 16px; font-size: 12px;' +
    '    text-transform: uppercase; letter-spacing: 1px; }' +
    '  .sidebar a { display: block; color: #a0aec0; padding: 10px 20px;' +
    '    text-decoration: none; font-size: 14px; }' +
    '  .sidebar a:hover, .sidebar a.active { background: #16213e; color: white; }' +
    '  .main { display: inline-block; width: calc(100% - 220px); padding: 24px;' +
    '    vertical-align: top; }' +
    '  .page-title { font-size: 24px; color: #1a1a2e; margin-bottom: 20px; }' +
    '  .card { background: white; border: 1px solid #e2e8f0; border-radius: 8px;' +
    '    padding: 20px; margin-bottom: 16px; }' +
    '  .stat-row { display: inline-block; width: 100%; margin-bottom: 20px; }' +
    '  .stat { display: inline-block; width: 30%; margin-right: 3%;' +
    '    background: #f8fafc; border-radius: 8px; padding: 16px; text-align: center; }' +
    '  .stat .number { font-size: 32px; font-weight: bold; color: #2563eb; }' +
    '  .stat .label { font-size: 12px; color: #888; margin-top: 4px; }' +
    '  table { width: 100%; border-collapse: collapse; }' +
    '  th { text-align: left; padding: 10px; border-bottom: 2px solid #e2e8f0;' +
    '    font-size: 12px; color: #888; text-transform: uppercase; }' +
    '  td { padding: 10px; border-bottom: 1px solid #f0f0f0; }' +
    '  .btn { padding: 6px 14px; border: none; border-radius: 4px; font-size: 13px; }' +
    '  .btn-primary { background: #2563eb; color: white; }' +
    '  .btn-sm { padding: 4px 10px; font-size: 12px; }' +
    '  .form-group { margin-bottom: 16px; }' +
    '  .form-group label { display: block; margin-bottom: 4px; font-weight: bold;' +
    '    font-size: 14px; color: #555; }' +
    '  .form-group input, .form-group select { width: 100%; padding: 8px;' +
    '    border: 1px solid #ddd; border-radius: 4px; }' +
    '</style>';
end;

function TfrmAdmin.GetSidebar(const ActivePage: string): string;

  function ActiveClass(const Page: string): string;
  begin
    if Page = ActivePage then
      Result := ' class="active"'
    else
      Result := '';
  end;

begin
  Result :=
    '<div class="sidebar">' +
    '  <h3>Navigation</h3>' +
    '  <a href="#dashboard"' + ActiveClass('dashboard') + '>Dashboard</a>' +
    '  <a href="#users"' + ActiveClass('users') + '>Users</a>' +
    '  <a href="#settings"' + ActiveClass('settings') + '>Settings</a>' +
    '</div>';
end;

function TfrmAdmin.BuildLayout(const ActivePage, Content: string): string;
begin
  Result := GetStyles +
    '<div>' +
    GetSidebar(ActivePage) +
    '<div class="main">' + Content + '</div>' +
```

```
    '</div>';
end;

procedure TfrmAdmin.SetupPages;
var
  Page: TTina4Page;
begin
  // Dashboard
  Page := HTMLPages1.Pages.Add;
  Page.PageName := 'dashboard';
  Page.IsDefault := True;
  Page.HTMLContent.Text := BuildLayout('dashboard',
    '<h1 class="page-title">Dashboard</h1>' +
    '<div class="stat-row">' +
    '  <div class="stat">' +
    '    <div class="number" id="userCount">248</div>' +
    '    <div class="label">Total Users</div>' +
    '  </div>' +
    '  <div class="stat">' +
    '    <div class="number" id="activeCount">189</div>' +
    '    <div class="label">Active</div>' +
    '  </div>' +
    '  <div class="stat">' +
    '    <div class="number" id="newCount">12</div>' +
    '    <div class="label">New Today</div>' +
    '  </div>' +
    '</div>' +
    '<div class="card">' +
    '  <h3>Recent Activity</h3>' +
    '  <table>' +
    '    <tr><td>Alice logged in</td><td style="color:#888;">2 min ago</td></tr>' +
    '    <tr><td>Bob updated profile</td><td style="color:#888;">15 min ago</td></tr>' +
    '    <tr><td>Carol created a report</td><td style="color:#888;">1 hour ago</td></tr>' +
    '  </table>' +
    '</div>');

  // Users
  Page := HTMLPages1.Pages.Add;
  Page.PageName := 'users';
  Page.HTMLContent.Text := BuildLayout('users',
    '<h1 class="page-title">User Management</h1>' +
    '<div class="card">' +
    '  <table>' +
    '    <thead>' +
    '      <tr><th>Name</th><th>Email</th><th>Role</th><th>Status</th><th>Actions</th></tr>' +
    '    </thead>' +
    '    <tbody>' +
    '      <tr>' +
    '        <td>Alice Smith</td><td>alice@example.com</td><td>Admin</td>' +
    '        <td style="color:green;">Active</td>' +
    '        <td><button class="btn btn-sm btn-primary" ' +
    '          onclick="Admin:EditUser(''1'')">Edit</button></td>' +
    '      </tr>' +
    '      <tr>' +
    '        <td>Bob Johnson</td><td>bob@example.com</td><td>Editor</td>' +
    '        <td style="color:green;">Active</td>' +
    '        <td><button class="btn btn-sm btn-primary" ' +
    '          onclick="Admin:EditUser(''2'')">Edit</button></td>' +
    '      </tr>' +
```

```
    '      <tr>' +
    '        <td>Carol Williams</td><td>carol@example.com</td><td>Viewer</td>' +
    '        <td style="color:red;">Inactive</td>' +
    '        <td><button class="btn btn-sm btn-primary" ' +
    '          onclick="Admin:EditUser(''3'')">Edit</button></td>' +
    '      </tr>' +
    '    </tbody>' +
    '  </table>' +
    '</div>');

  // Settings
  Page := HTMLPages1.Pages.Add;
  Page.PageName := 'settings';
  Page.HTMLContent.Text := BuildLayout('settings',
    '<h1 class="page-title">Settings</h1>' +
    '<div class="card">' +
    '  <form name="settingsForm">' +
    '    <div class="form-group">' +
    '      <label>Application Name</label>' +
    '      <input type="text" name="appName" id="appName" value="My Admin App">' +
    '    </div>' +
    '    <div class="form-group">' +
    '      <label>Default Language</label>' +
    '      <select name="language" id="language">' +
    '        <option value="en">English</option>' +
    '        <option value="fr">French</option>' +
    '        <option value="de">German</option>' +
    '      </select>' +
    '    </div>' +
    '    <div class="form-group">' +
    '      <label>Items Per Page</label>' +
    '      <input type="text" name="pageSize" id="pageSize" value="25">' +
    '    </div>' +
    '    <button type="submit" class="btn btn-primary">Save Settings</button>' +
    '  </form>' +
    '</div>');
end;

procedure TfrmAdmin.HTMLPages1AfterNavigate(Sender: TObject);
begin
  // Load data when navigating to specific pages
  if HTMLPages1.ActivePage = 'dashboard' then
  begin
    // Could refresh stats from API here
  end;
end;

procedure TfrmAdmin.HTMLRender1FormSubmit(Sender: TObject;
  const FormName: string; FormData: TStrings);
begin
  if FormName = 'settingsForm' then
  begin
    ShowMessage(Format('Settings saved: App=%s, Lang=%s, PageSize=%s',
      [FormData.Values['appName'],
       FormData.Values['language'],
       FormData.Values['pageSize']]));
  end;
end;
```

```
procedure TfrmAdmin.NavTo(PageName: String);
begin
  HTMLPages1.NavigateTo(PageName);
end;

procedure TfrmAdmin.EditUser(UserId: String);
begin
  ShowMessage('Edit user: ' + UserId);
  // In a real app: navigate to an edit page with the user's data pre-filled
end;

procedure TfrmAdmin.SaveSettings(Action: String);
begin
  ShowMessage('Settings action: ' + Action);
end;

end.
```

---

12. Complete Example: Navigation Guards

A login-protected app where users must authenticate before accessing protected pages.

```
unit GuardedApp;

interface

uses
  System.SysUtils, System.Classes, System.JSON,
  FMX.Types, FMX.Controls, FMX.Forms,
  Tina4HTMLRender, Tina4HTMLPages, Tina4REST;

type
  TfrmGuarded = class(TForm)
    HTMLRender1: TTina4HTMLRender;
    HTMLPages1: TTina4HTMLPages;
    restAPI: TTina4REST;
    procedure FormCreate(Sender: TObject);
    procedure HTMLPages1BeforeNavigate(Sender: TObject;
      const FromPage, ToPage: string; var Allow: Boolean);
    procedure HTMLRender1FormSubmit(Sender: TObject;
      const FormName: string; FormData: TStrings);
  private
    FIsLoggedIn: Boolean;
    FUserName: string;
    procedure SetupPages;
  published
    procedure Logout(Action: String);
  end;

var
  frmGuarded: TfrmGuarded;

implementation

{$R *.fmx}

procedure TfrmGuarded.FormCreate(Sender: TObject);
```

```
begin
  FIsLoggedIn := False;
  FUserName := '';
  restAPI.BaseUrl := 'https://api.example.com';
  HTMLPages1.Renderer := HTMLRender1;
  HTMLRender1.RegisterObject('App', Self);
  SetupPages;
end;

procedure TfrmGuarded.SetupPages;
var
  Page: TTina4Page;
begin
  // Login page (the default -- shown first)
  Page := HTMLPages1.Pages.Add;
  Page.PageName := 'login';
  Page.IsDefault := True;
  Page.HTMLContent.Text :=
    '<style>' +
    '  .login-box { max-width: 360px; margin: 80px auto; padding: 32px;' +
    '    background: white; border-radius: 8px; box-shadow: 0 4px 12px rgba(0,0,0,0.15); }' +
    '  .login-box h2 { text-align: center; margin-bottom: 24px; color: #333; }' +
    '  .field { margin-bottom: 16px; }' +
    '  .field label { display: block; margin-bottom: 4px; font-size: 14px; color: #555; }' +
    '  .field input { width: 100%; padding: 10px; border: 1px solid #ddd; border-radius: 4px; }'
    '  .btn { width: 100%; padding: 12px; background: #2563eb; color: white;' +
    '    border: none; border-radius: 4px; font-size: 16px; }' +
    '  .error { color: #dc2626; font-size: 13px; text-align: center;' +
    '    margin-bottom: 12px; display: none; }' +
    '</style>' +
    '<div class="login-box">' +
    '  <h2>Login Required</h2>' +
    '  <div class="error" id="loginError">Invalid credentials</div>' +
    '  <form name="loginForm">' +
    '    <div class="field">' +
    '      <label>Username</label>' +
    '      <input type="text" name="username" id="username" placeholder="Enter username">' +
    '    </div>' +
    '    <div class="field">' +
    '      <label>Password</label>' +
    '      <input type="password" name="password" id="password" placeholder="Enter password">' +
    '    </div>' +
    '    <button type="submit" class="btn">Sign In</button>' +
    '  </form>' +
    '</div>';

  // Protected: Dashboard
  Page := HTMLPages1.Pages.Add;
  Page.PageName := 'dashboard';
  Page.HTMLContent.Text :=
    '<style>' +
    '  .topbar { background: #1a1a2e; color: white; padding: 12px 20px;' +
    '    display: inline-block; width: 100%; }' +
    '  .topbar span { float: left; }' +
    '  .topbar .user { float: right; }' +
    '  .topbar a { color: #93c5fd; margin-left: 16px; }' +
    '  .content { padding: 24px; }' +
    '</style>' +
    '<div class="topbar">' +
```

```
    '  <span><b>Admin Panel</b></span>' +
    '  <span class="user">' +
    '    <a href="#profile">Profile</a>' +
    '    <a href="#settings">Settings</a>' +
    '    <span onclick="App:Logout(''now'')" style="color: #fca5a5; cursor: pointer;' +
    '      margin-left: 16px;">Logout</span>' +
    '  </span>' +
    '</div>' +
    '<div class="content">' +
    '  <h1>Dashboard</h1>' +
    '  <p>You are logged in. This is the protected dashboard.</p>' +
    '  <p><a href="#profile">View Profile</a> | <a href="#settings">Settings</a></p>' +
    '</div>';

  // Protected: Profile
  Page := HTMLPages1.Pages.Add;
  Page.PageName := 'profile';
  Page.HTMLContent.Text :=
    '<div class="topbar">' +
    '  <span><b>Admin Panel</b></span>' +
    '  <span class="user">' +
    '    <a href="#dashboard">Dashboard</a>' +
    '    <span onclick="App:Logout(''now'')" style="color: #fca5a5; cursor: pointer;' +
    '      margin-left: 16px;">Logout</span>' +
    '  </span>' +
    '</div>' +
    '<div class="content">' +
    '  <h1>Profile</h1>' +
    '  <p>User profile page. Only accessible when logged in.</p>' +
    '  <p><a href="#dashboard">Back to Dashboard</a></p>' +
    '</div>';

  // Protected: Settings
  Page := HTMLPages1.Pages.Add;
  Page.PageName := 'settings';
  Page.HTMLContent.Text :=
    '<div class="topbar">' +
    '  <span><b>Admin Panel</b></span>' +
    '  <span class="user">' +
    '    <a href="#dashboard">Dashboard</a>' +
    '    <span onclick="App:Logout(''now'')" style="color: #fca5a5; cursor: pointer;' +
    '      margin-left: 16px;">Logout</span>' +
    '  </span>' +
    '</div>' +
    '<div class="content">' +
    '  <h1>Settings</h1>' +
    '  <p>App settings. Only accessible when logged in.</p>' +
    '  <p><a href="#dashboard">Back to Dashboard</a></p>' +
    '</div>';
end;

procedure TfrmGuarded.HTMLPages1BeforeNavigate(Sender: TObject;
  const FromPage, ToPage: string; var Allow: Boolean);
begin
  // The login page is always accessible
  if ToPage = 'login' then
  begin
    Allow := True;
    Exit;
```

```
  end;

  // All other pages require authentication
  if not FIsLoggedIn then
  begin
    Allow := False;
    HTMLPages1.NavigateTo('login');
  end;
end;

procedure TfrmGuarded.HTMLRender1FormSubmit(Sender: TObject;
  const FormName: string; FormData: TStrings);
var
  Username, Password: string;
begin
  if FormName <> 'loginForm' then Exit;

  Username := FormData.Values['username'];
  Password := FormData.Values['password'];

  // Simple validation (in production, call your auth API)
  if (Username = 'admin') and (Password = 'password') then
  begin
    FIsLoggedIn := True;
    FUserName := Username;
    HTMLPages1.NavigateTo('dashboard');
  end
  else
  begin
    HTMLRender1.SetElementVisible('loginError', True);
  end;
end;

procedure TfrmGuarded.Logout(Action: String);
begin
  FIsLoggedIn := False;
  FUserName := '';
  HTMLPages1.NavigateTo('login');
end;

end.
```

---

13. Exercise: Wizard / Step-by-Step Form

Build a wizard with 4 pages (steps), Next/Back navigation, and validation before advancing.

Requirements

- Step 1: Personal Info (name, email) -- both required
- Step 2: Address (street, city, zip) -- all required
- Step 3: Preferences (language dropdown, newsletter checkbox)
- Step 4: Confirmation (summary of all entered data, submit button)
- "Next" button validates current step before advancing

- "Back" button always works (no validation needed)
- Step indicator showing current position (e.g., "Step 2 of 4")

Solution

```
unit WizardForm;

interface

uses
  System.SysUtils, System.Classes,
  FMX.Types, FMX.Controls, FMX.Forms,
  Tina4HTMLRender, Tina4HTMLPages;

type
  TfrmWizard = class(TForm)
    HTMLRender1: TTina4HTMLRender;
    HTMLPages1: TTina4HTMLPages;
    procedure FormCreate(Sender: TObject);
    procedure HTMLPages1BeforeNavigate(Sender: TObject;
      const FromPage, ToPage: string; var Allow: Boolean);
    procedure HTMLRender1FormSubmit(Sender: TObject;
      const FormName: string; FormData: TStrings);
  private
    FData: TStringList;
    procedure SetupPages;
    function GetStyles: string;
    function StepHeader(Current: Integer): string;
    function ValidateStep(const StepName: string): Boolean;
  published
    procedure GoNext(FromStep: String);
    procedure GoBack(FromStep: String);
  end;

var
  frmWizard: TfrmWizard;

implementation

{$R *.fmx}

procedure TfrmWizard.FormCreate(Sender: TObject);
begin
  FData := TStringList.Create;
  HTMLPages1.Renderer := HTMLRender1;
  HTMLRender1.RegisterObject('Wizard', Self);
  SetupPages;
end;

function TfrmWizard.GetStyles: string;
begin
  Result :=
    '<style>' +
    '  body { font-family: Arial, sans-serif; background: #f5f5f5; }' +
    '  .wizard { max-width: 500px; margin: 30px auto; background: white;' +
    '    border-radius: 8px; box-shadow: 0 2px 8px rgba(0,0,0,0.1); overflow: hidden; }' +
    '  .steps { display: inline-block; width: 100%; background: #f8f9fa;' +
    '    padding: 16px 20px; border-bottom: 1px solid #e0e0e0; }' +
    '  .step-dot { display: inline-block; width: 30px; height: 30px; border-radius: 50%;' +
    '    background: #ddd; color: #888; text-align: center; line-height: 30px;' +
    '    font-size: 14px; margin-right: 8px; }' +
    '  .step-dot.active { background: #2563eb; color: white; }' +
```

```
    '  .step-dot.done { background: #16a34a; color: white; }' +
    '  .step-label { font-size: 13px; color: #888; margin-left: 4px; margin-right: 20px; }' +
    '  .body { padding: 24px; }' +
    '  .field { margin-bottom: 16px; }' +
    '  .field label { display: block; margin-bottom: 4px; font-weight: bold; font-size: 14px; }'
    '  .field input, .field select { width: 100%; padding: 8px; border: 1px solid #ccc;' +
    '    border-radius: 4px; }' +
    '  .error-text { color: #dc2626; font-size: 12px; display: none; margin-top: 4px; }' +
    '  .buttons { padding: 16px 24px; border-top: 1px solid #e0e0e0;' +
    '    display: inline-block; width: 100%; }' +
    '  .btn { padding: 8px 20px; border: none; border-radius: 4px; font-size: 14px; }' +
    '  .btn-next { background: #2563eb; color: white; float: right; }' +
    '  .btn-back { background: #e2e8f0; color: #333; float: left; }' +
    '  .btn-submit { background: #16a34a; color: white; float: right; }' +
    '  .summary-row { padding: 8px 0; border-bottom: 1px solid #f0f0f0; }' +
    '  .summary-label { font-weight: bold; color: #555; display: inline-block; width: 120px; }'
    '</style>';
end;

function TfrmWizard.StepHeader(Current: Integer): string;
var
  Labels: array[1..4] of string;
  I: Integer;
  CSSClass: string;
begin
  Labels[1] := 'Personal';
  Labels[2] := 'Address';
  Labels[3] := 'Preferences';
  Labels[4] := 'Confirm';

  Result := '<div class="steps">';
  for I := 1 to 4 do
  begin
    if I < Current then
      CSSClass := 'step-dot done'
    else if I = Current then
      CSSClass := 'step-dot active'
    else
      CSSClass := 'step-dot';

    Result := Result +
      Format('<span class="%s">%d</span><span class="step-label">%s</span>',
        [CSSClass, I, Labels[I]]);
  end;
  Result := Result + '</div>';
end;

procedure TfrmWizard.SetupPages;
var
  Page: TTina4Page;
begin
  // Step 1: Personal Info
  Page := HTMLPages1.Pages.Add;
  Page.PageName := 'step1';
  Page.IsDefault := True;
  Page.HTMLContent.Text := GetStyles +
    '<div class="wizard">' +
    StepHeader(1) +
    '<div class="body">' +
```

```
  '  <h2>Personal Information</h2>' +
  '  <div class="field">' +
  '    <label>Full Name</label>' +
  '    <input type="text" name="name" id="name" placeholder="Your name">' +
  '    <div class="error-text" id="nameError">Name is required</div>' +
  '  </div>' +
  '  <div class="field">' +
  '    <label>Email</label>' +
  '    <input type="email" name="email" id="email" placeholder="you@example.com">' +
  '    <div class="error-text" id="emailError">Valid email is required</div>' +
  '  </div>' +
  '</div>' +
  '<div class="buttons">' +
  '  <button class="btn btn-next" onclick="Wizard:GoNext(''step1'')">Next</button>' +
  '</div>' +
  '</div>';

// Step 2: Address
Page := HTMLPages1.Pages.Add;
Page.PageName := 'step2';
Page.HTMLContent.Text := GetStyles +
  '<div class="wizard">' +
  StepHeader(2) +
  '<div class="body">' +
  '  <h2>Address</h2>' +
  '  <div class="field">' +
  '    <label>Street</label>' +
  '    <input type="text" name="street" id="street">' +
  '    <div class="error-text" id="streetError">Street is required</div>' +
  '  </div>' +
  '  <div class="field">' +
  '    <label>City</label>' +
  '    <input type="text" name="city" id="city">' +
  '    <div class="error-text" id="cityError">City is required</div>' +
  '  </div>' +
  '  <div class="field">' +
  '    <label>Zip Code</label>' +
  '    <input type="text" name="zip" id="zip">' +
  '    <div class="error-text" id="zipError">Zip code is required</div>' +
  '  </div>' +
  '</div>' +
  '<div class="buttons">' +
  '  <button class="btn btn-back" onclick="Wizard:GoBack(''step2'')">Back</button>' +
  '  <button class="btn btn-next" onclick="Wizard:GoNext(''step2'')">Next</button>' +
  '</div>' +
  '</div>';

// Step 3: Preferences
Page := HTMLPages1.Pages.Add;
Page.PageName := 'step3';
Page.HTMLContent.Text := GetStyles +
  '<div class="wizard">' +
  StepHeader(3) +
  '<div class="body">' +
  '  <h2>Preferences</h2>' +
  '  <div class="field">' +
  '    <label>Preferred Language</label>' +
  '    <select name="language" id="language">' +
  '      <option value="en">English</option>' +
```

```
    '      <option value="fr">French</option>' +
    '      <option value="de">German</option>' +
    '      <option value="es">Spanish</option>' +
    '    </select>' +
    '  </div>' +
    '  <div class="field">' +
    '    <input type="checkbox" name="newsletter" id="newsletter">' +
    '    <label style="display: inline; font-weight: normal;">Subscribe to newsletter</label>' +
    '  </div>' +
    '</div>' +
    '<div class="buttons">' +
    '  <button class="btn btn-back" onclick="Wizard:GoBack(''step3'')">Back</button>' +
    '  <button class="btn btn-next" onclick="Wizard:GoNext(''step3'')">Next</button>' +
    '</div>' +
    '</div>';

  // Step 4: Confirmation
  Page := HTMLPages1.Pages.Add;
  Page.PageName := 'step4';
  Page.HTMLContent.Text := GetStyles +
    '<div class="wizard">' +
    StepHeader(4) +
    '<div class="body">' +
    '  <h2>Confirm Your Details</h2>' +
    '  <div id="summaryContent">' +
    '    <div class="summary-row"><span class="summary-label">Name:</span> <span id="sumName">-<
    '    <div class="summary-row"><span class="summary-label">Email:</span> <span id="sumEmail">
    '    <div class="summary-row"><span class="summary-label">Street:</span> <span id="sumStreet
    '    <div class="summary-row"><span class="summary-label">City:</span> <span id="sumCity">-<
    '    <div class="summary-row"><span class="summary-label">Zip:</span> <span id="sumZip">-</s
    '    <div class="summary-row"><span class="summary-label">Language:</span> <span id="sumLang
    '    <div class="summary-row"><span class="summary-label">Newsletter:</span> <span id="sumNe
    '  </div>' +
    '</div>' +
    '<div class="buttons">' +
    '  <button class="btn btn-back" onclick="Wizard:GoBack(''step4'')">Back</button>' +
    '  <form name="wizardSubmit" style="display:inline; float:right;">' +
    '    <button type="submit" class="btn btn-submit">Submit</button>' +
    '  </form>' +
    '</div>' +
    '</div>';
end;

function TfrmWizard.ValidateStep(const StepName: string): Boolean;
begin
  Result := True;

  if StepName = 'step1' then
  begin
    var Name := HTMLRender1.GetElementValue('name');
    var Email := HTMLRender1.GetElementValue('email');

    if Name.Trim = '' then
    begin
      HTMLRender1.SetElementVisible('nameError', True);
      Result := False;
    end
    else
      HTMLRender1.SetElementVisible('nameError', False);
```

```
    if (Email.Trim = '') or (not Email.Contains('@')) then
    begin
      HTMLRender1.SetElementVisible('emailError', True);
      Result := False;
    end
    else
      HTMLRender1.SetElementVisible('emailError', False);

    if Result then
    begin
      FData.Values['name'] := Name;
      FData.Values['email'] := Email;
    end;
  end

  else if StepName = 'step2' then
  begin
    var Street := HTMLRender1.GetElementValue('street');
    var City := HTMLRender1.GetElementValue('city');
    var Zip := HTMLRender1.GetElementValue('zip');

    if Street.Trim = '' then
    begin
      HTMLRender1.SetElementVisible('streetError', True);
      Result := False;
    end
    else
      HTMLRender1.SetElementVisible('streetError', False);

    if City.Trim = '' then
    begin
      HTMLRender1.SetElementVisible('cityError', True);
      Result := False;
    end
    else
      HTMLRender1.SetElementVisible('cityError', False);

    if Zip.Trim = '' then
    begin
      HTMLRender1.SetElementVisible('zipError', True);
      Result := False;
    end
    else
      HTMLRender1.SetElementVisible('zipError', False);

    if Result then
    begin
      FData.Values['street'] := Street;
      FData.Values['city'] := City;
      FData.Values['zip'] := Zip;
    end;
  end

  else if StepName = 'step3' then
  begin
    FData.Values['language'] := HTMLRender1.GetElementValue('language');
    FData.Values['newsletter'] := HTMLRender1.GetElementValue('newsletter');
  end;
end;
```

```
procedure TfrmWizard.GoNext(FromStep: String);
begin
  if not ValidateStep(FromStep) then Exit;

  if FromStep = 'step1' then
    HTMLPages1.NavigateTo('step2')
  else if FromStep = 'step2' then
    HTMLPages1.NavigateTo('step3')
  else if FromStep = 'step3' then
  begin
    HTMLPages1.NavigateTo('step4');
    // Populate summary after navigation renders
    HTMLRender1.SetElementText('sumName', FData.Values['name']);
    HTMLRender1.SetElementText('sumEmail', FData.Values['email']);
    HTMLRender1.SetElementText('sumStreet', FData.Values['street']);
    HTMLRender1.SetElementText('sumCity', FData.Values['city']);
    HTMLRender1.SetElementText('sumZip', FData.Values['zip']);
    HTMLRender1.SetElementText('sumLang', FData.Values['language']);
    HTMLRender1.SetElementText('sumNews', FData.Values['newsletter']);
  end;
end;

procedure TfrmWizard.GoBack(FromStep: String);
begin
  if FromStep = 'step2' then
    HTMLPages1.NavigateTo('step1')
  else if FromStep = 'step3' then
    HTMLPages1.NavigateTo('step2')
  else if FromStep = 'step4' then
    HTMLPages1.NavigateTo('step3');
end;

procedure TfrmWizard.HTMLPages1BeforeNavigate(Sender: TObject;
  const FromPage, ToPage: string; var Allow: Boolean);
begin
  // Allow all navigation (validation is handled in GoNext)
  Allow := True;
end;

procedure TfrmWizard.HTMLRender1FormSubmit(Sender: TObject;
  const FormName: string; FormData: TStrings);
begin
  if FormName = 'wizardSubmit' then
  begin
    ShowMessage(Format(
      'Registration complete!' + sLineBreak +
      'Name: %s' + sLineBreak +
      'Email: %s' + sLineBreak +
      'City: %s',
      [FData.Values['name'], FData.Values['email'], FData.Values['city']]));
  end;
end;

end.
```

---

14. Common Gotchas

Page Name Case Sensitivity

Symptom: Navigation does not work. The renderer stays on the current page.

Fix: Page names are case-sensitive. `#Dashboard` does not match a page named `dashboard`:

```
// Page defined as:
Page.PageName := 'dashboard';

// This works:
<a href="#dashboard">Dashboard</a>

// This does NOT work:
<a href="#Dashboard">Dashboard</a>
```

Circular Navigation in OnBeforeNavigate

Symptom: Application freezes or stack overflow.

Fix: If your `OnBeforeNavigate` handler calls `NavigateTo`, it triggers another `OnBeforeNavigate` event. Always allow navigation to the redirect target:

```
procedure TForm1.BeforeNavigate(Sender: TObject;
  const FromPage, ToPage: string; var Allow: Boolean);
begin
  // ALWAYS allow navigation to the login page
  if ToPage = 'login' then
  begin
    Allow := True;
    Exit;
  end;

  if not FIsLoggedIn then
  begin
    Allow := False;
    HTMLPages1.NavigateTo('login');  // This triggers BeforeNavigate again
    // Without the 'login' check above, you get infinite recursion
  end;
end;
```

TwigContent vs HTMLContent Priority

Symptom: HTML changes to `HTMLContent` have no effect.

Fix: If `TwigContent` is not empty, it takes priority and `HTMLContent` is ignored. Clear `TwigContent` if you want to use raw HTML:

```
// If TwigContent has content, HTMLContent is ignored
Page.TwigContent.Clear;
Page.HTMLContent.Text := '<h1>This will now display</h1>';
```

Styles Not Persisting Between Pages

Symptom: CSS styles defined on one page do not apply on another page.

Fix: Each page's HTML is self-contained. Styles defined in one page's `<style>` block do not carry over when you navigate to another page. Include shared styles in every page, or use a

helper function:

```
function GetSharedStyles: string;
begin
  Result := '<style>/* shared styles */</style>';
end;

// Use in every page
Page.HTMLContent.Text := GetSharedStyles + '<div>Page content</div>';
```

Form Data Lost on Navigation

Symptom: User fills in a form, navigates away, comes back, and the form is empty.

Fix: Page content is re-rendered from `HTMLContent` on every navigation. Form values are not preserved. Save form data to variables before navigating away (as shown in the wizard example), or store values in a `TStringList` and pre-populate fields after navigation.

Summary

What	How
Setup	`HTMLPages1.Renderer := HTMLRender1`
Add page (design)	Double-click component, use collection editor
Add page (runtime)	`HTMLPages1.Pages.Add` -- set `PageName`, `HTMLContent`
Default page	`Page.IsDefault := True`
Navigate via link	`<a href="#pagename">`
Navigate via code	`HTMLPages1.NavigateTo('pagename')`
Read active page	`HTMLPages1.ActivePage`
Guard navigation	`OnBeforeNavigate` -- set `Allow := False` to block
Post-navigation	`OnAfterNavigate` -- load data, update UI
Twig in pages	Set `TwigContent` + `SetTwigVariable`
File templates	Set `TwigTemplatePath` for `{% include %}`/`{% extends %}`
Page priority	`TwigContent` (if set) overrides `HTMLContent`

Twig Templates

HTML Without the String Concatenation Hell

You have built REST calls. You have populated MemTables. You have rendered HTML in your Delphi forms. But every time you construct HTML, you end up with code like this:

```
HTML := '<div class="card">' +
  '<h2>' + Customer.Name + '</h2>' +
  '<p>' + Customer.Email + '</p>' +
  '</div>';
```

Six lines. Two bugs waiting to happen. One missing quote away from a broken layout. And when the designer changes the card to include a phone number, you are back in the Pascal editor, escaping quotes and hoping the HTML is still valid.

Twig templates fix this. Write HTML in HTML files. Drop in variables with `{{ name }}`. Add loops with `{% for %}`. Inherit layouts with `{% extends %}`. The template engine handles the rest. Your Pascal code passes data in; the template decides how to display it.

TTina4Twig is a Twig-compatible engine built directly into the Tina4 Delphi component library. It supports variables, control structures, filters, functions, template inheritance, macros, and integration with the HTML renderer. Every feature documented here works at design time and runtime.

---

1. TTina4Twig Standalone Usage

The simplest way to use Twig is standalone -- create an instance, set variables, render a template string or file.

Create and Render

```
uses
  Tina4Twig;

procedure TForm1.RenderGreeting;
var
  Twig: TTina4Twig;
  Variables: TStringDict;
begin
  Twig := TTina4Twig.Create('C:\MyApp\templates');
  Variables := TStringDict.Create;
  try
    Variables.Add('name', 'Andre');
    Variables.Add('role', 'Developer');

    Memo1.Lines.Text := Twig.Render('greeting.html', Variables);
  finally
    Variables.Free;
    Twig.Free;
  end;
end;
```

The constructor takes a template path. This is the root directory where Twig looks for template files referenced by `{% include %}` and `{% extends %}` tags. Every file reference is relative to this path.

Render from a String

You can also render inline template strings without loading from a file:

```
var
  Twig: TTina4Twig;
  Variables: TStringDict;
begin
  Twig := TTina4Twig.Create('');
  Variables := TStringDict.Create;
  try
    Variables.Add('name', 'World');

    Memo1.Lines.Text := Twig.Render(
      '<h1>Hello {{ name }}!</h1>', Variables);
    // Output: <h1>Hello World!</h1>
  finally
    Variables.Free;
    Twig.Free;
  end;
end;
```

Passing Complex Data

Variables can be strings, numbers, arrays, or nested objects:

```
Variables.Add('name', 'Andre');
Variables.Add('score', TValue.From<Integer>(42));
Variables.Add('items', TValue.From<TArray<String>>(['Apple', 'Banana', 'Cherry']));
```

---

2. Variables

Variables are the bridge between your Pascal code and your templates. Everything you pass via `SetVariable` or through a `TStringDict` is accessible inside double curly braces.

Simple Variables

```
Hello {{ name }}!
Your score is {{ score }}.
```

Nested Properties

Access object properties with dot notation:

```
{{ user.email }}
{{ order.customer.name }}
{{ settings.theme.primaryColor }}
```

Setting Variables Inside Templates

Use `{% set %}` to define variables directly in a template:

```
{% set greeting = 'Hello' %}
```

```
{% set items = ['Apple', 'Banana'] %}
{% set total = price * quantity %}
{% set fullName = firstName ~ ' ' ~ lastName %}

<p>{{ greeting }}, {{ fullName }}!</p>
<p>Total: {{ total }}</p>
```

The `~` operator concatenates strings. Variables set with `{% set %}` are scoped to the current template block.

---

3. Control Structures

if / elseif / else

Conditional rendering. Test any expression -- variable truthiness, comparisons, filters:

```
{% if users|length > 0 %}
  <ul>
    {% for user in users %}
      <li>{{ user.name }}</li>
    {% endfor %}
  </ul>
{% elseif guests|length > 0 %}
  <p>Guests only today.</p>
{% else %}
  <p>No users found.</p>
{% endif %}
```

Combine conditions with `and`, `or`, `not`:

```
{% if user.isAdmin and user.isActive %}
  <a href="#admin">Admin Panel</a>
{% endif %}

{% if not user.isVerified %}
  <div class="alert alert-warning">Please verify your email.</div>
{% endif %}
```

for Loops

Iterate over arrays:

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

Key-value pairs:

```
{% for key, value in settings %}
  <tr>
    <td>{{ key }}</td>
    <td>{{ value }}</td>
  </tr>
{% endfor %}
```

Ranges:

```
{% for i in 0..10 %}
```

```
  <option value="{{ i }}">{{ i }}</option>
{% endfor %}

{% for letter in 'a'..'f' %}
  <span>{{ letter }}</span>
{% endfor %}
```

The `loop` variable is available inside every for loop:

```
{% for item in items %}
  <tr class="{% if loop.first %}first{% endif %}{% if loop.last %}last{% endif %}">
    <td>{{ loop.index }}</td>
    <td>{{ item.name }}</td>
  </tr>
{% endfor %}
```

with Blocks

Scope variables to a block without polluting the outer context:

```
{% with { title: 'Dashboard', subtitle: 'Overview' } %}
  <div class="header">
    <h1>{{ title }}</h1>
    <p>{{ subtitle }}</p>
  </div>
{% endwith %}
```

---

4. Template Inheritance

This is where Twig saves you from copy-pasting the same header and footer into 30 pages.

Base Template (base.html)

```
<!DOCTYPE html>
<html>
<head>
  <title>{% block title %}My App{% endblock %}</title>
  <style>
    body { font-family: Arial, sans-serif; margin: 0; padding: 20px; }
    .header { background: #2c3e50; color: white; padding: 15px; }
    .content { padding: 20px; }
    .footer { border-top: 1px solid #ccc; padding: 10px; color: #666; }
  </style>
  {% block head %}{% endblock %}
</head>
<body>
  <div class="header">
    <h1>{% block header %}My Application{% endblock %}</h1>
  </div>
  <div class="content">
    {% block content %}{% endblock %}
  </div>
  <div class="footer">
    {% block footer %}&copy; 2026 My Company{% endblock %}
  </div>
</body>
</html>
```

Child Template (dashboard.html)

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

{% block title %}Dashboard - My App{% endblock %}

{% block header %}Dashboard{% endblock %}

{% block content %}
  <h2>Welcome, {{ userName }}</h2>
  <p>You have {{ messageCount }} new messages.</p>
{% endblock %}
```

The child template only defines the blocks it wants to override. Everything else comes from the base. Change the footer in `base.html` and every page that extends it picks up the change.

include

Pull in reusable fragments:

```
{% include 'header.html' %}

<div class="main">
  {% include 'sidebar.html' with { menu: menuItems } %}
  <div class="content">
    {{ content }}
  </div>
</div>

{% include 'footer.html' %}
```

The `with` clause passes variables to the included template. Without it, the included template inherits the parent's variables.

Setting the Template Path in Pascal

For `{% extends %}` and `{% include %}` to resolve correctly, set the template path:

```
Twig := TTina4Twig.Create('C:\MyApp\templates');
```

All template references are relative to this path. If `dashboard.html` says `{% extends 'base.html' %}`, Twig looks for `C:\MyApp\templates\base.html`.

---

5. Macros

Macros are reusable template fragments -- the Twig equivalent of functions. Define them once, call them everywhere.

Define a Macro

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

Call a Macro

```
{{ input('username', '', 'text') }}
{{ input('email', user.email, 'email') }}
{{ input('password', '', 'password') }}
```

A More Complex Macro

```
{% macro alert(message, type) %}
  <div class="alert alert-{{ type|default('info') }}" role="alert">
    {% if type == 'danger' %}
      <strong>Error!</strong>
    {% elseif type == 'warning' %}
      <strong>Warning!</strong>
    {% endif %}
    {{ message }}
  </div>
{% endmacro %}

{{ alert('Record saved successfully.', 'success') }}
{{ alert('Please check your input.', 'warning') }}
{{ alert('Connection failed.', 'danger') }}
```

---

6. Filters Reference

Filters transform output values. Chain them with the pipe | operator:

```
{{ name|upper|length }}
{{ description|striptags|trim|truncate(100) }}
```

String Filters

Filter	What it does	Example	
`upper`	Uppercase	`{{ 'hello'\	upper }} produces HELLO`
`lower`	Lowercase	`{{ 'HELLO'\	lower }} produces hello`
`capitalize`	First letter uppercase	`{{ 'hello world'\	capitalize }} produces Hello world`
`title`	Title case every word	`{{ 'hello world'\	title }} produces Hello World`
`trim`	Strip leading/trailing whitespace	`{{ ' hi '\	trim }} produces hi`
`nl2br`	Convert newlines to ` `	`{{ "line1\nline2"\	nl2br }}`
`striptags`	Remove HTML tags	`{{ 'bold'\	striptags }} produces bold`
`replace`	Replace substrings	`{{ 'hello'\	replace({'e': 'a'}) }} produces hallo`
`split`	Split into array	`{{ 'a,b,c'\	split(',') }} produces ['a','b','c']`
`slug`	URL-friendly slug	`{{ 'Hello World!'\	slug }} produces hello-world`
`spaceless`	Remove whitespace between HTML tags	`{{ ' hi '\	spaceless }}`
`u`	Unicode string wrapper	`{{ text\	u }}`

Practical example -- building a navigation menu:

```
{% for page in pages %}
  <a href="/{{ page.title|slug }}"
     class="nav-link {% if page.title == currentPage %}active{% endif %}">
    {{ page.title|title }}
  </a>
{% endfor %}
```

Number Filters

Filter	What it does	Example	
`abs`	Absolute value	`{{ -42|	abs }} produces 42`
`number_format`	Format with decimals, separators	`{{ 1234.5|	number_format(2, '.', ',') }} produces 1,234.50`
`format_number`	Locale-aware number format	`{{ 1234|	format_number }}`
`format_currency`	Format as currency	`{{ 1234.50|	format_currency('USD') }} produces $1,234.50`

Practical example -- product pricing:

```
{% for product in products %}
  <div class="product-card">
    <h3>{{ product.name }}</h3>
    <p class="price">{{ product.price|format_currency('USD') }}</p>
    {% if product.discount > 0 %}
      <p class="discount">Save {{ product.discount|abs }}%</p>
    {% endif %}
  </div>
{% endfor %}
```

Array Filters

Filter	What it does	Example	
`length`	Count elements	`{{ items\	length }}`
`first`	First element	`{{ items\	first }}`
`last`	Last element	`{{ items\	last }}`
`join`	Join into string	`{{ items\	join(', ') }}`
`keys`	Get object keys	`{{ settings\	keys }}`
`merge`	Merge two arrays	`{{ defaults\	merge(overrides) }}`
`sort`	Sort ascending	`{{ items\	sort }}`
`reverse`	Reverse order	`{{ items\	reverse }}`
`shuffle`	Random order	`{{ items\	shuffle }}`
`slice`	Extract subset	`{{ items\	slice(0, 5) }}` -- first 5 items
`batch`	Split into chunks	`{{ items\	batch(3) }}` -- groups of 3
`column`	Extract one property	`{{ users\	column('name') }}`
`filter`	Keep matching items	`{{ items\	filter }}`
`find`	Find first match	`{{ items\	find }}`
`map`	Transform each item	`{{ items\	map }}`
`reduce`	Reduce to single value	`{{ items\	reduce }}`
`min`	Smallest value	`{{ prices\	min }}`
`max`	Largest value	`{{ prices\	max }}`

Practical example -- rendering a product grid:

```
{% for row in products|batch(3) %}
  <div class="row">
    {% for product in row %}
      <div class="col">
        <h4>{{ product.name }}</h4>
        <p>{{ product.price|number_format(2) }}</p>
      </div>
    {% endfor %}
  </div>
{% endfor %}

<p>Showing {{ products|length }} products.
   Price range: {{ products|column('price')|min|format_currency('USD') }}
   to {{ products|column('price')|max|format_currency('USD') }}</p>

<p>Categories: {{ products|column('category')|sort|join(', ') }}</p>
```

Date Filters

Filter	What it does	Example	
`date`	Format a date	`{{ order.created|	date('Y-m-d') }}`
`date_modify`	Shift a date	`{{ order.created|	date_modify('+30 days') }}`
`format_date`	Format date (alias)	`{{ order.created|	format_date }}`
`format_datetime`	Format date and time	`{{ order.created|	format_datetime }}`
`format_time`	Format time only	`{{ order.created|	format_time }}`

Date format specifiers (PHP-style, auto-converted by Tina4):

Specifier	Meaning	Output
`Y`	4-digit year	`2026`
`y`	2-digit year	`26`
`m`	Month, zero-padded	`03`
`n`	Month, no padding	`3`
`d`	Day, zero-padded	`01`
`j`	Day, no padding	`1`
`H`	Hour 24h, zero-padded	`14`
`h`	Hour 12h, zero-padded	`02`
`i`	Minutes	`30`
`s`	Seconds	`05`
`A`	AM/PM	`PM`
`D`	Short day name	`Mon`
`l`	Full day name	`Monday`
`M`	Short month name	`Jan`
`F`	Full month name	`January`

Practical example -- order timeline:

```
<h3>Order #{{ order.id }}</h3>
<table>
  <tr>
    <td>Placed:</td>
    <td>{{ order.created|date('F j, Y') }} at {{ order.created|date('h:i A') }}</td>
  </tr>
  <tr>
    <td>Ships by:</td>
```

```
    <td>{{ order.created|date_modify('+3 days')|date('F j, Y') }}</td>
  </tr>
  <tr>
    <td>Delivery estimate:</td>
    <td>{{ order.created|date_modify('+7 days')|date('l, F j') }}</td>
  </tr>
</table>
```

Encoding Filters

Filter	What it does	Example	
`escape` / `e`	Escape HTML entities	`{{ userInput|	escape }}`
`raw`	Output without escaping	`{{ trustedHtml|	raw }}`
`url_encode`	URL-encode a string	`{{ query|	url_encode }}`
`json_encode`	Encode as JSON	`{{ data|	json_encode }}`
`json_decode`	Decode from JSON	`{{ jsonString|	json_decode }}`
`convert_encoding`	Convert character encoding	`{{ text|	convert_encoding('UTF-8') }}`
`data_uri`	Create a data URI	`{{ imageData|	data_uri }}`

Practical example -- building a search URL:

```
<a href="/search?q={{ searchTerm|url_encode }}&category={{ category|url_encode }}">
  Search for "{{ searchTerm|escape }}"
</a>

<script>
  var config = {{ appSettings|json_encode|raw }};
</script>
```

Other Filters

Filter	What it does	Example	
`default`	Fallback value if empty/null	`{{ name|	default('Guest') }}`
`format`	Printf-style formatting	`{{ 'Hello %s, you have %d items'|	format(name, count) }}`
`plural`	Pluralize a word	`{{ 'item'|	plural }} `produces` items`
`singular`	Singularize a word	`{{ 'items'|	singular }} `produces` item`

```
<p>You have {{ count }} {{ 'item'|plural }} in your cart.</p>
<p>Welcome, {{ userName|default('Guest') }}!</p>
<p>{{ 'Found %d %s in %s'|format(results|length, 'result'|plural, category) }}</p>
```

---

7. Functions

Functions work like filters but are called directly:

Function	What it does	Example
`range()`	Generate a sequence	`{% for i in range(1, 10) %}`
`dump()`	Debug output of a variable	`{{ dump(user) }}`
`date()`	Create or format a date	`{{ date('now', 'Y-m-d') }}`

```
{# Generate page numbers #}
{% for page in range(1, totalPages) %}
  <a href="?page={{ page }}"
     class="{% if page == currentPage %}active{% endif %}">
    {{ page }}
  </a>
{% endfor %}

{# Debug during development #}
{% if debug %}
  <pre>{{ dump(order) }}</pre>
{% endif %}

{# Show current date #}
<p>Report generated: {{ date('now', 'F j, Y \\a\\t h:i A') }}</p>
```

---

8. Operators

Category	Operators	Example
Comparison	`==`, `!=`, `<`, `>`, `<=`, `>=`	`{% if price > 100 %}`
Logical	`and`, `or`, `not`	`{% if active and verified %}`
String	`~` (concat), `in`, `starts with`, `ends with`, `matches`	`{{ first ~ ' ' ~ last }}`
Math	`+`, `-`, `*`, `/`, `%`, `**`	`{{ price * quantity }}`
Range	`..`	`{% for i in 1..10 %}`

String operators in action:

```
{# Concatenation #}
{% set fullName = firstName ~ ' ' ~ lastName %}

{# Membership test #}
{% if 'admin' in user.roles %}
```

```
  <a href="#admin">Admin Panel</a>
{% endif %}

{# String matching #}
{% if email ends with '@company.com' %}
  <span class="badge">Internal</span>
{% endif %}

{% if filename starts with 'report_' %}
  <span class="badge">Report</span>
{% endif %}

{% if phone matches '/^\\+27/' %}
  <span>South Africa</span>
{% endif %}
```

---

9. Integration with TTina4HTMLRender

The HTML renderer has built-in Twig support. You do not need to create a standalone TTina4Twig instance -- the renderer handles it internally.

Setting Variables and Rendering

```
// Set variables before assigning the template
Tina4HTMLRender1.SetTwigVariable('title', 'Dashboard');
Tina4HTMLRender1.SetTwigVariable('userName', 'Andre');
Tina4HTMLRender1.SetTwigVariable('messageCount', '5');

// Set the Twig template -- it renders to HTML automatically
Tina4HTMLRender1.Twig.Text :=
  '<div class="header">' +
  '  <h1>{{ title }}</h1>' +
  '  <p>Welcome back, {{ userName }}! You have {{ messageCount }} new messages.</p>' +
  '</div>';
```

File-Based Templates with the Renderer

```
// Set the template path for includes and extends
Tina4HTMLRender1.TwigTemplatePath := 'C:\MyApp\templates';

// Set variables
Tina4HTMLRender1.SetTwigVariable('userName', 'Andre');
Tina4HTMLRender1.SetTwigVariable('notifications', '3');

// Load from file -- Twig processes it, then the renderer displays the HTML
Tina4HTMLRender1.Twig.LoadFromFile('C:\MyApp\templates\dashboard.html');
```

Combining with REST Data

The real power emerges when you combine REST data with Twig templates:

```
procedure TForm1.LoadCustomerCard(CustomerID: Integer);
var
  StatusCode: Integer;
  Response: TJSONObject;
begin
  Response := Tina4REST1.Get(StatusCode, '/customers/' + CustomerID.ToString);
```

```
  try
    Tina4HTMLRender1.SetTwigVariable('customer',
      Response.GetValue<String>('name'));
    Tina4HTMLRender1.SetTwigVariable('email',
      Response.GetValue<String>('email'));
    Tina4HTMLRender1.SetTwigVariable('orders',
      Response.GetValue<String>('orderCount'));

    Tina4HTMLRender1.Twig.LoadFromFile(
      'C:\MyApp\templates\customer-card.html');
  finally
    Response.Free;
  end;
end;
```

---

10. Complete Example: Email Template System

Build a reusable email template system with a base layout, customer notification, and order confirmation.

File Structure

```
templates/
  email/
    base.html           -- shared email layout
    notification.html   -- customer notification
    order-confirm.html  -- order confirmation with product loop
```

base.html

```
<!DOCTYPE html>
<html>
<head>
  <style>
    body { font-family: Arial, sans-serif; background: #f4f4f4; margin: 0; padding: 0; }
    .container { max-width: 600px; margin: 20px auto; background: white; border-radius: 8px; ove
    .header { background: #2c3e50; color: white; padding: 20px; text-align: center; }
    .content { padding: 30px; }
    .footer { background: #ecf0f1; padding: 15px; text-align: center; font-size: 12px; color: #6
    .btn { display: inline-block; padding: 10px 20px; background: #1abc9c; color: white; text-de
  </style>
</head>
<body>
  <div class="container">
    <div class="header">
      <h1>{% block header %}{{ companyName|default('My Company') }}{% endblock %}</h1>
    </div>
    <div class="content">
      {% block content %}{% endblock %}
    </div>
    <div class="footer">
      {% block footer %}
        &copy; {{ date('now', 'Y') }} {{ companyName|default('My Company') }}. All rights reserv
      {% endblock %}
    </div>
  </div>
</body>
</html>
```

notification.html

```
{% extends 'email/base.html' %}

{% block header %}Notification{% endblock %}

{% block content %}
  <h2>Hello {{ customerName|default('Customer') }},</h2>
  <p>{{ message }}</p>

  {% if actionUrl %}
    <p style="text-align: center; margin: 30px 0;">
      <a href="{{ actionUrl }}" class="btn">{{ actionText|default('View Details') }}</a>
    </p>
  {% endif %}

  {% if notes|length > 0 %}
    <h3>Additional Notes:</h3>
    <ul>
      {% for note in notes %}
        <li>{{ note }}</li>
      {% endfor %}
    </ul>
  {% endif %}
{% endblock %}
```

order-confirm.html

```
{% extends 'email/base.html' %}

{% block header %}Order Confirmation{% endblock %}

{% block content %}
  <h2>Thank you, {{ customerName }}!</h2>
  <p>Your order <strong>#{{ orderId }}</strong> has been confirmed.</p>
  <p>Placed on: {{ orderDate|date('F j, Y') }}</p>

  <table style="width: 100%; border-collapse: collapse; margin: 20px 0;">
    <thead>
      <tr style="background: #ecf0f1;">
        <th style="padding: 10px; text-align: left;">Product</th>
        <th style="padding: 10px; text-align: right;">Qty</th>
        <th style="padding: 10px; text-align: right;">Price</th>
        <th style="padding: 10px; text-align: right;">Total</th>
      </tr>
    </thead>
    <tbody>
      {% for item in items %}
        <tr style="border-bottom: 1px solid #eee;">
          <td style="padding: 10px;">{{ item.name }}</td>
          <td style="padding: 10px; text-align: right;">{{ item.quantity }}</td>
          <td style="padding: 10px; text-align: right;">{{ item.price|format_currency('USD') }}<
          <td style="padding: 10px; text-align: right;">{{ item.total|format_currency('USD') }}<
        </tr>
      {% endfor %}
    </tbody>
    <tfoot>
      <tr>
        <td colspan="3" style="padding: 10px; text-align: right;"><strong>Subtotal:</strong></td
        <td style="padding: 10px; text-align: right;">{{ subtotal|format_currency('USD') }}</td>
      </tr>
      <tr>
        <td colspan="3" style="padding: 10px; text-align: right;">Tax ({{ taxRate }}%):</td>
        <td style="padding: 10px; text-align: right;">{{ tax|format_currency('USD') }}</td>
      </tr>
      {% if discount > 0 %}
        <tr style="color: #e74c3c;">
          <td colspan="3" style="padding: 10px; text-align: right;">Discount:</td>
          <td style="padding: 10px; text-align: right;">-{{ discount|format_currency('USD') }}</
        </tr>
      {% endif %}
      <tr style="font-size: 1.2em; font-weight: bold;">
        <td colspan="3" style="padding: 10px; text-align: right;">Total:</td>
        <td style="padding: 10px; text-align: right;">{{ grandTotal|format_currency('USD') }}</t
      </tr>
    </tfoot>
  </table>

  <p>Estimated delivery: {{ orderDate|date_modify('+5 days')|date('l, F j, Y') }}</p>
{% endblock %}
```

Pascal Code to Render

```
procedure TForm1.SendOrderConfirmation(OrderID: Integer);
var
  Twig: TTina4Twig;
  Variables: TStringDict;
  EmailBody: String;
begin
  Twig := TTina4Twig.Create('C:\MyApp\templates');
  Variables := TStringDict.Create;
  try
    Variables.Add('companyName', 'Acme Store');
    Variables.Add('customerName', 'Andre van Zuydam');
    Variables.Add('orderId', OrderID.ToString);
    Variables.Add('orderDate', FormatDateTime('yyyy-mm-dd', Now));
    Variables.Add('subtotal', '149.97');
    Variables.Add('taxRate', '15');
    Variables.Add('tax', '22.50');
    Variables.Add('discount', '10.00');
    Variables.Add('grandTotal', '162.47');

    // Items would be passed as a JSON array string or structured data
    Variables.Add('items', '[' +
      '{"name": "Widget Pro", "quantity": "2", "price": "49.99", "total": "99.98"},' +
      '{"name": "Gadget Mini", "quantity": "1", "price": "49.99", "total": "49.99"}' +
      ']');

    EmailBody := Twig.Render('email/order-confirm.html', Variables);
    Memo1.Lines.Text := EmailBody;
  finally
    Variables.Free;
    Twig.Free;
  end;
end;
```

---

11. Complete Example: Report Generator

Build a product catalog report with categories, pricing, inventory status, filters, and macros.

report-catalog.html

```
{% macro statusBadge(quantity) %}
  {% if quantity > 10 %}
    <span style="color: #27ae60; font-weight: bold;">In Stock ({{ quantity }})</span>
  {% elseif quantity > 0 %}
    <span style="color: #f39c12; font-weight: bold;">Low Stock ({{ quantity }})</span>
  {% else %}
    <span style="color: #e74c3c; font-weight: bold;">Out of Stock</span>
  {% endif %}
{% endmacro %}

{% macro priceCell(price, discount) %}
  {% if discount > 0 %}
    <span style="text-decoration: line-through; color: #999;">{{ price|format_currency('USD') }}
    <span style="color: #e74c3c; font-weight: bold;">
      {{ (price - discount)|format_currency('USD') }}
    </span>
  {% else %}
    {{ price|format_currency('USD') }}
  {% endif %}
{% endmacro %}

<!DOCTYPE html>
<html>
<head>
  <style>
    body { font-family: Arial, sans-serif; margin: 20px; }
    h1 { color: #2c3e50; }
    h2 { color: #34495e; border-bottom: 2px solid #1abc9c; padding-bottom: 5px; }
    table { width: 100%; border-collapse: collapse; margin-bottom: 30px; }
    th { background: #2c3e50; color: white; padding: 10px; text-align: left; }
    td { padding: 8px; border-bottom: 1px solid #eee; }
    tr:nth-child(even) { background: #f9f9f9; }
    .summary { background: #ecf0f1; padding: 15px; border-radius: 8px; margin: 20px 0; }
  </style>
</head>
<body>
  <h1>{{ reportTitle|default('Product Catalog') }}</h1>
  <p>Generated: {{ date('now', 'F j, Y \\a\\t h:i A') }}</p>
  <p>Total products: {{ products|length }}</p>

  <div class="summary">
    <strong>Price Range:</strong>
    {{ products|column('price')|min|format_currency('USD') }} -
    {{ products|column('price')|max|format_currency('USD') }}
  </div>

  {% for category in categories %}
    <h2>{{ category.name|title }}</h2>

    {% set categoryProducts = products|filter %}

    <table>
      <thead>
        <tr>
          <th>#</th>
          <th>Product</th>
          <th>SKU</th>
```

```
          <th>Price</th>
          <th>Stock</th>
        </tr>
      </thead>
      <tbody>
        {% for product in category.products %}
          <tr>
            <td>{{ loop.index }}</td>
            <td>
              <strong>{{ product.name }}</strong>
              {% if product.description %}
                <br><small style="color: #666;">{{ product.description|striptags|slice(0, 80) }}
              {% endif %}
            </td>
            <td>{{ product.sku|upper }}</td>
            <td>{{ priceCell(product.price, product.discount|default(0)) }}</td>
            <td>{{ statusBadge(product.stock) }}</td>
          </tr>
        {% endfor %}
      </tbody>
    </table>
  {% endfor %}

  <div class="summary">
    <h3>Report Summary</h3>
    <p>Categories: {{ categories|length }}</p>
    <p>Total Products: {{ products|length }}</p>
    <p>Products in stock: {{ products|column('stock')|filter|length }}</p>
  </div>
</body>
</html>
```

Pascal Code

```
procedure TForm1.GenerateCatalogReport;
var
  Twig: TTina4Twig;
  Variables: TStringDict;
begin
  Twig := TTina4Twig.Create('C:\MyApp\templates');
  Variables := TStringDict.Create;
  try
    Variables.Add('reportTitle', 'Q1 2026 Product Catalog');

    // In a real app, this data comes from GetJSONFromDB or a REST call
    Variables.Add('categories', '[' +
      '{"name": "electronics", "products": [' +
        '{"name": "USB-C Hub", "sku": "elec-001", "price": "29.99", "stock": "45", "description"
        '{"name": "Wireless Mouse", "sku": "elec-002", "price": "19.99", "stock": "3", "discount
        '{"name": "Mechanical Keyboard", "sku": "elec-003", "price": "89.99", "stock": "0"}' +
      ']},' +
      '{"name": "office supplies", "products": [' +
        '{"name": "Notebook Set", "sku": "off-001", "price": "12.99", "stock": "120"},' +
        '{"name": "Pen Pack", "sku": "off-002", "price": "8.99", "stock": "200"}' +
      ']}' +
      ']');

    Variables.Add('products', '[' +
      '{"name": "USB-C Hub", "price": "29.99", "stock": "45"},' +
      '{"name": "Wireless Mouse", "price": "19.99", "stock": "3"},' +
      '{"name": "Mechanical Keyboard", "price": "89.99", "stock": "0"},' +
      '{"name": "Notebook Set", "price": "12.99", "stock": "120"},' +
      '{"name": "Pen Pack", "price": "8.99", "stock": "200"}' +
      ']');

    WebBrowser1.Navigate('about:blank');
    // Render and display the report
    var HTML := Twig.Render('report-catalog.html', Variables);
    Memo1.Lines.Text := HTML;
  finally
    Variables.Free;
    Twig.Free;
  end;
end;
```

Exercise: Invoice Template

Build an invoice template with the following requirements:

- Company header with logo placeholder, company name, and address
- Customer billing and shipping address blocks
- Line items loop with description, quantity, unit price, and line total
- Subtotal, tax (configurable rate), and grand total calculations
- Conditional discount display (only shows discount row if discount > 0)
- Payment terms section with due date (30 days from invoice date)

- Use macros for the address block (reuse for billing and shipping)
- Use template inheritance -- extend a base invoice layout

Solution

invoice-base.html:

```
<!DOCTYPE html>
<html>
<head>
  <style>
    body { font-family: Arial, sans-serif; margin: 0; padding: 30px; color: #333; }
    .invoice-header { display: flex; justify-content: space-between; margin-bottom: 30px; }
    .company-info h1 { margin: 0; color: #2c3e50; }
    .invoice-meta { text-align: right; }
    .addresses { display: flex; gap: 40px; margin-bottom: 30px; }
    .address-block { flex: 1; }
    .address-block h3 { color: #2c3e50; margin-bottom: 5px; }
    table { width: 100%; border-collapse: collapse; }
    th { background: #2c3e50; color: white; padding: 10px; text-align: left; }
    td { padding: 8px 10px; border-bottom: 1px solid #eee; }
    .totals td { border: none; }
    .grand-total { font-size: 1.3em; font-weight: bold; color: #2c3e50; }
    .terms { margin-top: 30px; padding: 15px; background: #f9f9f9; border-radius: 4px; }
  </style>
  {% block head %}{% endblock %}
</head>
<body>
  {% block invoice %}{% endblock %}
</body>
</html>
```

invoice.html:

```
{% extends 'invoice-base.html' %}

{% macro addressBlock(label, addr) %}
  <div class="address-block">
    <h3>{{ label }}</h3>
    <p>
      <strong>{{ addr.name }}</strong><br>
      {{ addr.street }}<br>
      {{ addr.city }}, {{ addr.state }} {{ addr.zip }}<br>
      {% if addr.country %}{{ addr.country }}<br>{% endif %}
      {% if addr.phone %}Tel: {{ addr.phone }}{% endif %}
    </p>
  </div>
{% endmacro %}

{% block invoice %}
  <div class="invoice-header">
    <div class="company-info">
      <h1>{{ company.name }}</h1>
      <p>{{ company.address }}<br>
         {{ company.city }}, {{ company.state }} {{ company.zip }}<br>
         {{ company.email }}</p>
    </div>
    <div class="invoice-meta">
      <h2>INVOICE</h2>
```

```
    <p>
      <strong>Invoice #:</strong> {{ invoiceNumber }}<br>
      <strong>Date:</strong> {{ invoiceDate|date('F j, Y') }}<br>
      <strong>Due Date:</strong> {{ invoiceDate|date_modify('+30 days')|date('F j, Y') }}
    </p>
  </div>
</div>

<div class="addresses">
  {{ addressBlock('Bill To', billing) }}
  {{ addressBlock('Ship To', shipping) }}
</div>

<table>
  <thead>
    <tr>
      <th style="width: 50%;">Description</th>
      <th style="text-align: right;">Qty</th>
      <th style="text-align: right;">Unit Price</th>
      <th style="text-align: right;">Total</th>
    </tr>
  </thead>
  <tbody>
    {% for item in lineItems %}
      <tr>
        <td>{{ item.description }}</td>
        <td style="text-align: right;">{{ item.quantity }}</td>
        <td style="text-align: right;">{{ item.unitPrice|format_currency('USD') }}</td>
        <td style="text-align: right;">{{ item.lineTotal|format_currency('USD') }}</td>
      </tr>
    {% endfor %}
  </tbody>
  <tfoot>
    <tr class="totals">
      <td colspan="3" style="text-align: right; padding-top: 15px;"><strong>Subtotal:</strong>
      <td style="text-align: right; padding-top: 15px;">{{ subtotal|format_currency('USD') }}<
    </tr>
    {% if discount > 0 %}
      <tr class="totals" style="color: #27ae60;">
        <td colspan="3" style="text-align: right;">Discount ({{ discountPercent }}%):</td>
        <td style="text-align: right;">-{{ discount|format_currency('USD') }}</td>
      </tr>
    {% endif %}
    <tr class="totals">
      <td colspan="3" style="text-align: right;">Tax ({{ taxRate }}%):</td>
      <td style="text-align: right;">{{ tax|format_currency('USD') }}</td>
    </tr>
    <tr class="totals grand-total">
      <td colspan="3" style="text-align: right;">Total Due:</td>
      <td style="text-align: right;">{{ grandTotal|format_currency('USD') }}</td>
    </tr>
  </tfoot>
</table>

<div class="terms">
  <h3>Payment Terms</h3>
  <p>Payment is due within 30 days of the invoice date.
     Please reference invoice #{{ invoiceNumber }} with your payment.</p>
  <p>Bank: {{ company.bank|default('First National Bank') }}<br>
```

```
        Account: {{ company.account|default('Contact us for details') }}</p>
  </div>
{% endblock %}
```

Pascal code to render the invoice:

```
procedure TForm1.GenerateInvoice;
var
  Twig: TTina4Twig;
  Variables: TStringDict;
begin
  Twig := TTina4Twig.Create('C:\MyApp\templates');
  Variables := TStringDict.Create;
  try
    Variables.Add('invoiceNumber', 'INV-2026-0042');
    Variables.Add('invoiceDate', FormatDateTime('yyyy-mm-dd', Now));

    Variables.Add('company', '{"name": "Acme Corp", "address": "123 Main St", ' +
      '"city": "Cape Town", "state": "WC", "zip": "8001", ' +
      '"email": "billing@acme.co.za"}');

    Variables.Add('billing', '{"name": "John Smith", "street": "456 Oak Ave", ' +
      '"city": "Johannesburg", "state": "GP", "zip": "2001", "phone": "+27 11 555 0123"}');

    Variables.Add('shipping', '{"name": "John Smith", "street": "789 Pine Rd", ' +
      '"city": "Durban", "state": "KZN", "zip": "4001"}');

    Variables.Add('lineItems', '[' +
      '{"description": "Widget Pro - Annual License", "quantity": "5", "unitPrice": "99.00", "li
      '{"description": "Setup & Configuration", "quantity": "1", "unitPrice": "250.00", "lineTot
      '{"description": "Training (per hour)", "quantity": "4", "unitPrice": "75.00", "lineTotal"
      ']');

    Variables.Add('subtotal', '1045.00');
    Variables.Add('discountPercent', '10');
    Variables.Add('discount', '104.50');
    Variables.Add('taxRate', '15');
    Variables.Add('tax', '141.08');
    Variables.Add('grandTotal', '1081.58');

    Memo1.Lines.Text := Twig.Render('invoice.html', Variables);
  finally
    Variables.Free;
    Twig.Free;
  end;
end;
```

Common Gotchas

Template path resolution. Every `{% extends %}` and `{% include %}` path is relative to the path you pass to `TTina4Twig.Create()`. If you pass an empty string, file-based template references will fail silently. Always set a real path:

```
// Wrong -- includes and extends will not find files
Twig := TTina4Twig.Create('');

// Right
```

```
Twig := TTina4Twig.Create('C:\MyApp\templates');
```

Variable scope in for loops. Variables set inside a `{% for %}` block do not persist outside the loop. This trips people up when trying to accumulate a total:

```
{# This does NOT work as expected #}
{% set total = 0 %}
{% for item in items %}
  {% set total = total + item.price %}
{% endfor %}
<p>Total: {{ total }}</p>  {# Still 0! #}
```

Calculate totals in your Pascal code and pass them as variables instead.

HTML escaping by default. Twig escapes HTML entities by default. If you pass pre-formatted HTML as a variable, it will show raw tags. Use the `raw` filter for trusted content:

```
{# Variable contains '<strong>Bold</strong>' #}
{{ content }}          {# Shows: &lt;strong&gt;Bold&lt;/strong&gt; #}
{{ content|raw }}      {# Shows: Bold (rendered as HTML) #}
```

Only use `raw` on content you control. Never use it on user input.

TStringDict memory management. Always free the `TStringDict` and `TTina4Twig` in a `try/finally` block. Forgetting this is the most common memory leak with Twig templates:

```
Twig := TTina4Twig.Create('C:\templates');
Variables := TStringDict.Create;
try
  // ... render ...
finally
  Variables.Free;  // Always free both
  Twig.Free;
end;
```

Whitespace in templates. Twig preserves whitespace exactly as written in the template. If you see extra blank lines in your output, your template has extra blank lines. Use `{%-` and `-%}` to trim whitespace around tags:

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

WebSockets

Real-Time Without Polling

Your Delphi application displays a dashboard with live metrics. The user stares at the numbers. Nothing moves. They click refresh. The numbers update. They stare again. They click refresh again.

This is HTTP. You ask, the server answers, the connection closes. If you want fresh data, you ask again. Polling every 5 seconds works, but it is wasteful -- 90% of those requests return the same data. It hammers the server. It wastes bandwidth. And the user still sees data that is up to 5 seconds stale.

WebSocket is the fix. One connection opens. It stays open. The server pushes data the instant it changes. The client pushes messages back. No request/response cycle. No polling interval. No stale data.

TTina4WebSocketClient brings WebSocket to your Delphi FMX applications with RFC 6455 compliance, automatic reconnection, ping/pong keepalive, and event-driven message handling. Drop it on a form, set a URL, connect, and start receiving real-time data.

---

1. TTina4WebSocketClient Overview

The WebSocket client handles the full RFC 6455 protocol:

- **Full-duplex communication** -- send and receive simultaneously over a single TCP connection
- **Auto-reconnect** -- configurable reconnection with backoff when the connection drops
- **Ping/pong keepalive** -- automatic heartbeat to detect dead connections
- **Text and binary messages** -- handle both message types
- **Event-driven** -- OnConnected, OnMessage, OnDisconnected, OnError callbacks

Component Setup

Drop a `TTina4WebSocketClient` on your form from the Tina4 palette, or create it at runtime:

```
uses
  Tina4WebSocketClient;

var
  WSClient: TTina4WebSocketClient;
begin
  WSClient := TTina4WebSocketClient.Create(Self);
  WSClient.URL := 'wss://api.example.com/ws';
end;
```

---

2. Basic Connection

Design-Time Configuration

Set properties in the Object Inspector:

Property	Value
`URL`	`wss://api.example.com/ws`
`AutoReconnect`	`True`
`ReconnectInterval`	`5000` (ms)
`PingInterval`	`30000` (ms)

Runtime Connection

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  Tina4WebSocket1.URL := 'wss://api.example.com/ws';
  Tina4WebSocket1.AutoReconnect := True;
  Tina4WebSocket1.ReconnectInterval := 5000;

  Tina4WebSocket1.OnConnected := WebSocketConnected;
  Tina4WebSocket1.OnMessage := WebSocketMessage;
  Tina4WebSocket1.OnDisconnected := WebSocketDisconnected;
  Tina4WebSocket1.OnError := WebSocketError;

  Tina4WebSocket1.Connect;
end;
```

Event Handlers

```
procedure TForm1.WebSocketConnected(Sender: TObject);
begin
  TThread.Synchronize(nil, procedure
  begin
    LabelStatus.Text := 'Connected';
    LabelStatus.TextSettings.FontColor := TAlphaColorRec.Green;
  end);
end;

procedure TForm1.WebSocketMessage(Sender: TObject; const AMessage: string);
begin
  TThread.Synchronize(nil, procedure
  begin
    MemoMessages.Lines.Add(FormatDateTime('hh:nn:ss', Now) + ' ' + AMessage);
  end);
end;

procedure TForm1.WebSocketDisconnected(Sender: TObject; const ACode: Integer;
  const AReason: string);
begin
  TThread.Synchronize(nil, procedure
  begin
    LabelStatus.Text := 'Disconnected: ' + AReason;
    LabelStatus.TextSettings.FontColor := TAlphaColorRec.Red;
  end);
end;

procedure TForm1.WebSocketError(Sender: TObject; const AError: string);
begin
  TThread.Synchronize(nil, procedure
  begin
    MemoMessages.Lines.Add('ERROR: ' + AError);
  end);
end;
```

---

3. Sending Messages

Send Text

```
procedure TForm1.ButtonSendClick(Sender: TObject);
begin
  Tina4WebSocket1.Send(EditMessage.Text);
  EditMessage.Text := '';
end;
```

Send JSON

```
procedure TForm1.SendJSON;
var
  Obj: TJSONObject;
begin
  Obj := TJSONObject.Create;
  try
    Obj.AddPair('type', 'subscribe');
    Obj.AddPair('channel', 'notifications');
    Obj.AddPair('userId', TJSONNumber.Create(1001));

    Tina4WebSocket1.Send(Obj.ToString);
  finally
    Obj.Free;
  end;
end;
```

Check Connection Before Sending

```
procedure TForm1.SafeSend(const AMessage: string);
begin
  if Tina4WebSocket1.IsConnected then
    Tina4WebSocket1.Send(AMessage)
  else
    ShowMessage('Not connected to server');
end;
```

---

4. Auto-Reconnect Behavior

When the connection drops, TTina4WebSocketClient automatically attempts to reconnect if `AutoReconnect` is `True`.

Configuration

```
Tina4WebSocket1.AutoReconnect := True;
Tina4WebSocket1.ReconnectInterval := 5000;  // 5 seconds between attempts
```

Reconnect Flow

```
Connected -> Connection Lost -> Wait 5s -> Reconnect Attempt 1
  -> Fail -> Wait 5s -> Reconnect Attempt 2
  -> Success -> Connected
```

Handling Reconnection in Code

```
procedure TForm1.WebSocketConnected(Sender: TObject);
begin
  TThread.Synchronize(nil, procedure
  begin
    LabelStatus.Text := 'Connected';

    // Re-subscribe to channels after reconnection
    var Sub := TJSONObject.Create;
    try
      Sub.AddPair('type', 'subscribe');
      Sub.AddPair('channel', 'updates');
      Tina4WebSocket1.Send(Sub.ToString);
    finally
      Sub.Free;
    end;
  end);
end;
```

The OnConnected event fires every time the connection opens -- including after a reconnect. Use it to re-subscribe to channels or re-authenticate.

---

5. Ping/Pong Keepalive

WebSocket connections can silently die -- a router drops the connection, a firewall times it out, the server crashes. Without keepalive, your client thinks it is still connected while the connection is actually dead.

TTina4WebSocketClient sends periodic ping frames. If the server does not respond with a pong within a timeout, the connection is considered dead and reconnection begins.

```
Tina4WebSocket1.PingInterval := 30000;  // Send ping every 30 seconds
```

You do not need to handle pings manually. The component manages the ping/pong protocol internally.

---

6. Binary vs Text Messages

WebSocket supports two frame types: text and binary. TTina4WebSocketClient handles both.

Text Messages

Most WebSocket APIs use text frames with JSON payloads:

```
procedure TForm1.WebSocketMessage(Sender: TObject; const AMessage: string);
var
  JSON: TJSONObject;
begin
  JSON := StrToJSONObject(AMessage);
  if Assigned(JSON) then
  try
    var MsgType := JSON.GetValue<String>('type', '');
```

```
    TThread.Synchronize(nil, procedure
    begin
      if MsgType = 'price_update' then
        UpdatePriceDisplay(JSON)
      else if MsgType = 'notification' then
        ShowNotification(JSON)
      else if MsgType = 'error' then
        HandleServerError(JSON);
    end);
  finally
    JSON.Free;
  end;
end;
```

Binary Messages

For binary data (images, files, protocol buffers), use the binary message event:

```
procedure TForm1.WebSocketBinaryMessage(Sender: TObject; const AData: TBytes);
begin
  TThread.Synchronize(nil, procedure
  begin
    // Save binary data to file
    var Stream := TFileStream.Create('received_data.bin', fmCreate);
    try
      Stream.WriteBuffer(AData[0], Length(AData));
    finally
      Stream.Free;
    end;
  end);
end;
```

---

7. Disconnecting

Graceful Close

```
procedure TForm1.ButtonDisconnectClick(Sender: TObject);
begin
  Tina4WebSocket1.AutoReconnect := False;  // Prevent reconnection
  Tina4WebSocket1.Disconnect;
end;
```

Cleanup on Form Destroy

```
procedure TForm1.FormDestroy(Sender: TObject);
begin
  Tina4WebSocket1.AutoReconnect := False;
  if Tina4WebSocket1.IsConnected then
    Tina4WebSocket1.Disconnect;
end;
```

---

8. Complete Example: Real-Time Chat Client

Build a chat application that connects to a WebSocket server, sends and receives messages, and displays them in an HTML renderer with online status.

Form Layout

Place these components on your form:

- `TTina4WebSocketClient` (Tina4WebSocket1)
- `TTina4HTMLRender` (HTMLRender1) -- displays chat messages
- `TEdit` (EditMessage) -- message input
- `TButton` (ButtonSend) -- send button
- `TEdit` (EditUsername) -- username input
- `TButton` (ButtonConnect) -- connect/disconnect
- `TLabel` (LabelStatus) -- connection status

Implementation

```
unit ChatForm;

interface

uses
  System.SysUtils, System.Types, System.Classes, System.JSON,
  FMX.Types, FMX.Controls, FMX.Forms, FMX.StdCtrls, FMX.Edit,
  FMX.Memo, FMX.Layouts,
  Tina4WebSocketClient, Tina4HTMLRender, Tina4Core;

type
  TFormChat = class(TForm)
    Tina4WebSocket1: TTina4WebSocketClient;
    HTMLRender1: TTina4HTMLRender;
    EditMessage: TEdit;
    ButtonSend: TButton;
    EditUsername: TEdit;
    ButtonConnect: TButton;
    LabelStatus: TLabel;
    procedure FormCreate(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
    procedure ButtonConnectClick(Sender: TObject);
    procedure ButtonSendClick(Sender: TObject);
    procedure EditMessageKeyDown(Sender: TObject; var Key: Word;
      var KeyChar: Char; Shift: TShiftState);
  private
    FChatHistory: TStringList;
    FOnlineUsers: TStringList;
    FIsConnected: Boolean;
    procedure OnWSOpen(Sender: TObject);
    procedure OnWSMessage(Sender: TObject; const AMessage: string);
    procedure OnWSClose(Sender: TObject; const ACode: Integer;
      const AReason: string);
    procedure OnWSError(Sender: TObject; const AError: string);
    procedure AddChatMessage(const AUser, AMessage, ATime: string;
      AIsOwn: Boolean);
    procedure RefreshChatDisplay;
    procedure UpdateOnlineStatus;
  end;

var
  FormChat: TFormChat;

implementation

{$R *.fmx}

procedure TFormChat.FormCreate(Sender: TObject);
begin
  FChatHistory := TStringList.Create;
  FOnlineUsers := TStringList.Create;
  FIsConnected := False;

  Tina4WebSocket1.AutoReconnect := True;
  Tina4WebSocket1.ReconnectInterval := 3000;
  Tina4WebSocket1.PingInterval := 25000;

  Tina4WebSocket1.OnConnected := OnWSOpen;
```

```
  Tina4WebSocket1.OnMessage := OnWSMessage;
  Tina4WebSocket1.OnDisconnected := OnWSClose;
  Tina4WebSocket1.OnError := OnWSError;

  ButtonSend.Enabled := False;
  LabelStatus.Text := 'Disconnected';

  // Show empty chat
  RefreshChatDisplay;
end;

procedure TFormChat.FormDestroy(Sender: TObject);
begin
  Tina4WebSocket1.AutoReconnect := False;
  if Tina4WebSocket1.IsConnected then
    Tina4WebSocket1.Disconnect;
  FChatHistory.Free;
  FOnlineUsers.Free;
end;

procedure TFormChat.ButtonConnectClick(Sender: TObject);
begin
  if FIsConnected then
  begin
    Tina4WebSocket1.AutoReconnect := False;
    Tina4WebSocket1.Disconnect;
  end
  else
  begin
    if EditUsername.Text.Trim.IsEmpty then
    begin
      ShowMessage('Please enter a username');
      Exit;
    end;

    Tina4WebSocket1.URL := 'wss://chat.example.com/ws';
    Tina4WebSocket1.AutoReconnect := True;
    Tina4WebSocket1.Connect;
    LabelStatus.Text := 'Connecting...';
  end;
end;

procedure TFormChat.OnWSOpen(Sender: TObject);
begin
  TThread.Synchronize(nil, procedure
  begin
    FIsConnected := True;
    LabelStatus.Text := 'Connected';
    ButtonConnect.Text := 'Disconnect';
    ButtonSend.Enabled := True;
    EditUsername.Enabled := False;

    // Send join message
    var JoinMsg := TJSONObject.Create;
    try
      JoinMsg.AddPair('type', 'join');
      JoinMsg.AddPair('username', EditUsername.Text);
      Tina4WebSocket1.Send(JoinMsg.ToString);
    finally
```

```
      JoinMsg.Free;
    end;
  end);
end;

procedure TFormChat.OnWSMessage(Sender: TObject; const AMessage: string);
begin
  TThread.Synchronize(nil, procedure
  var
    JSON: TJSONObject;
  begin
    JSON := StrToJSONObject(AMessage);
    if not Assigned(JSON) then Exit;
    try
      var MsgType := JSON.GetValue<String>('type', '');

      if MsgType = 'chat' then
      begin
        var User := JSON.GetValue<String>('username', 'Unknown');
        var Text := JSON.GetValue<String>('message', '');
        var Time := JSON.GetValue<String>('time', FormatDateTime('hh:nn', Now));
        var IsOwn := (User = EditUsername.Text);

        AddChatMessage(User, Text, Time, IsOwn);
      end
      else if MsgType = 'user_list' then
      begin
        FOnlineUsers.Clear;
        var Users := JSON.GetValue<TJSONArray>('users');
        if Assigned(Users) then
          for var I := 0 to Users.Count - 1 do
            FOnlineUsers.Add(Users.Items[I].Value);
        UpdateOnlineStatus;
      end
      else if MsgType = 'user_joined' then
      begin
        var User := JSON.GetValue<String>('username', '');
        if not FOnlineUsers.Contains(User) then
          FOnlineUsers.Add(User);
        AddChatMessage('System', User + ' joined the chat', '', False);
        UpdateOnlineStatus;
      end
      else if MsgType = 'user_left' then
      begin
        var User := JSON.GetValue<String>('username', '');
        var Idx := FOnlineUsers.IndexOf(User);
        if Idx >= 0 then
          FOnlineUsers.Delete(Idx);
        AddChatMessage('System', User + ' left the chat', '', False);
        UpdateOnlineStatus;
      end;
    finally
      JSON.Free;
    end;
  end);
end;

procedure TFormChat.OnWSClose(Sender: TObject; const ACode: Integer;
  const AReason: string);
```

```
begin
  TThread.Synchronize(nil, procedure
  begin
    FIsConnected := False;
    LabelStatus.Text := 'Disconnected';
    ButtonConnect.Text := 'Connect';
    ButtonSend.Enabled := False;
    EditUsername.Enabled := True;
    FOnlineUsers.Clear;
    UpdateOnlineStatus;
  end);
end;

procedure TFormChat.OnWSError(Sender: TObject; const AError: string);
begin
  TThread.Synchronize(nil, procedure
  begin
    AddChatMessage('System', 'Error: ' + AError, '', False);
  end);
end;

procedure TFormChat.ButtonSendClick(Sender: TObject);
begin
  if EditMessage.Text.Trim.IsEmpty then Exit;
  if not Tina4WebSocket1.IsConnected then Exit;

  var Msg := TJSONObject.Create;
  try
    Msg.AddPair('type', 'chat');
    Msg.AddPair('message', EditMessage.Text.Trim);
    Tina4WebSocket1.Send(Msg.ToString);
  finally
    Msg.Free;
  end;

  EditMessage.Text := '';
  EditMessage.SetFocus;
end;

procedure TFormChat.EditMessageKeyDown(Sender: TObject; var Key: Word;
  var KeyChar: Char; Shift: TShiftState);
begin
  if Key = vkReturn then
    ButtonSendClick(Sender);
end;

procedure TFormChat.AddChatMessage(const AUser, AMessage, ATime: string;
  AIsOwn: Boolean);
var
  Alignment, BgColor, TextColor: string;
begin
  if AIsOwn then
  begin
    Alignment := 'right';
    BgColor := '#1abc9c';
    TextColor := 'white';
  end
  else if AUser = 'System' then
  begin
```

```
    Alignment := 'center';
    BgColor := '#f0f0f0';
    TextColor := '#666';
  end
  else
  begin
    Alignment := 'left';
    BgColor := '#ecf0f1';
    TextColor := '#333';
  end;

  FChatHistory.Add(Format(
    '<div style="text-align: %s; margin: 5px 0;">' +
    '  <div style="display: inline-block; background: %s; color: %s; ' +
    '    padding: 8px 12px; border-radius: 12px; max-width: 70%%;">' +
    '    <small><strong>%s</strong> %s</small><br>%s' +
    '  </div>' +
    '</div>',
    [Alignment, BgColor, TextColor, AUser, ATime, AMessage]));

  RefreshChatDisplay;
end;

procedure TFormChat.RefreshChatDisplay;
begin
  HTMLRender1.HTML.Text :=
    '<div style="font-family: Arial, sans-serif; padding: 10px;">' +
    '<h3 style="color: #2c3e50; border-bottom: 1px solid #eee; padding-bottom: 5px;">Chat</h3>'
    FChatHistory.Text +
    '</div>';
end;

procedure TFormChat.UpdateOnlineStatus;
begin
  // Update status label with online user count
  if FOnlineUsers.Count > 0 then
    LabelStatus.Text := Format('Connected (%d online)', [FOnlineUsers.Count])
  else if FIsConnected then
    LabelStatus.Text := 'Connected'
  else
    LabelStatus.Text := 'Disconnected';
end;

end.
```

---

9. Complete Example: Live Data Feed

Build a real-time price feed that connects to a WebSocket server, receives price updates, and displays them with color-coded changes.

Implementation

```
unit PriceFeedForm;

interface

uses
  System.SysUtils, System.Types, System.Classes, System.JSON,
  System.Generics.Collections, FMX.Types, FMX.Controls, FMX.Forms,
  FMX.StdCtrls, FMX.Layouts,
  Tina4WebSocketClient, Tina4HTMLRender, Tina4Core;

type
  TPriceInfo = record
    Symbol: string;
    Price: Double;
    PrevPrice: Double;
    Change: Double;
    ChangePercent: Double;
    LastUpdate: TDateTime;
  end;

  TFormPriceFeed = class(TForm)
    Tina4WebSocket1: TTina4WebSocketClient;
    HTMLRender1: TTina4HTMLRender;
    ButtonConnect: TButton;
    LabelStatus: TLabel;
    procedure FormCreate(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
    procedure ButtonConnectClick(Sender: TObject);
  private
    FPrices: TDictionary<string, TPriceInfo>;
    procedure OnWSOpen(Sender: TObject);
    procedure OnWSMessage(Sender: TObject; const AMessage: string);
    procedure OnWSClose(Sender: TObject; const ACode: Integer;
      const AReason: string);
    procedure OnWSError(Sender: TObject; const AError: string);
    procedure UpdatePriceDisplay;
    function GetChangeColor(AChange: Double): string;
    function GetChangeArrow(AChange: Double): string;
  end;

var
  FormPriceFeed: TFormPriceFeed;

implementation

{$R *.fmx}

procedure TFormPriceFeed.FormCreate(Sender: TObject);
begin
  FPrices := TDictionary<string, TPriceInfo>.Create;

  Tina4WebSocket1.URL := 'wss://feeds.example.com/prices';
  Tina4WebSocket1.AutoReconnect := True;
  Tina4WebSocket1.ReconnectInterval := 5000;
  Tina4WebSocket1.PingInterval := 20000;

  Tina4WebSocket1.OnConnected := OnWSOpen;
```

```
  Tina4WebSocket1.OnMessage := OnWSMessage;
  Tina4WebSocket1.OnDisconnected := OnWSClose;
  Tina4WebSocket1.OnError := OnWSError;

  LabelStatus.Text := 'Disconnected';
  UpdatePriceDisplay;
end;

procedure TFormPriceFeed.FormDestroy(Sender: TObject);
begin
  Tina4WebSocket1.AutoReconnect := False;
  if Tina4WebSocket1.IsConnected then
    Tina4WebSocket1.Disconnect;
  FPrices.Free;
end;

procedure TFormPriceFeed.ButtonConnectClick(Sender: TObject);
begin
  if Tina4WebSocket1.IsConnected then
  begin
    Tina4WebSocket1.AutoReconnect := False;
    Tina4WebSocket1.Disconnect;
    ButtonConnect.Text := 'Connect';
  end
  else
  begin
    Tina4WebSocket1.AutoReconnect := True;
    Tina4WebSocket1.Connect;
    LabelStatus.Text := 'Connecting...';
    ButtonConnect.Text := 'Disconnect';
  end;
end;

procedure TFormPriceFeed.OnWSOpen(Sender: TObject);
begin
  TThread.Synchronize(nil, procedure
  begin
    LabelStatus.Text := 'Connected - Receiving prices';

    // Subscribe to price updates
    var Sub := TJSONObject.Create;
    try
      Sub.AddPair('type', 'subscribe');
      var Symbols := TJSONArray.Create;
      Symbols.Add('BTC-USD');
      Symbols.Add('ETH-USD');
      Symbols.Add('AAPL');
      Symbols.Add('GOOGL');
      Symbols.Add('MSFT');
      Sub.AddPair('symbols', Symbols);
      Tina4WebSocket1.Send(Sub.ToString);
    finally
      Sub.Free;
    end;
  end);
end;

procedure TFormPriceFeed.OnWSMessage(Sender: TObject; const AMessage: string);
begin
```

```
  TThread.Synchronize(nil, procedure
  var
    JSON: TJSONObject;
    Info: TPriceInfo;
  begin
    JSON := StrToJSONObject(AMessage);
    if not Assigned(JSON) then Exit;
    try
      var MsgType := JSON.GetValue<String>('type', '');

      if MsgType = 'price' then
      begin
        var Symbol := JSON.GetValue<String>('symbol', '');
        var NewPrice := JSON.GetValue<Double>('price', 0);

        if FPrices.TryGetValue(Symbol, Info) then
        begin
          Info.PrevPrice := Info.Price;
          Info.Price := NewPrice;
          Info.Change := NewPrice - Info.PrevPrice;
          if Info.PrevPrice > 0 then
            Info.ChangePercent := (Info.Change / Info.PrevPrice) * 100
          else
            Info.ChangePercent := 0;
        end
        else
        begin
          Info.Symbol := Symbol;
          Info.Price := NewPrice;
          Info.PrevPrice := NewPrice;
          Info.Change := 0;
          Info.ChangePercent := 0;
        end;

        Info.LastUpdate := Now;
        FPrices.AddOrSetValue(Symbol, Info);
        UpdatePriceDisplay;
      end;
    finally
      JSON.Free;
    end;
  end);
end;

procedure TFormPriceFeed.OnWSClose(Sender: TObject; const ACode: Integer;
  const AReason: string);
begin
  TThread.Synchronize(nil, procedure
  begin
    LabelStatus.Text := 'Disconnected - Reconnecting...';
    ButtonConnect.Text := 'Connect';
  end);
end;

procedure TFormPriceFeed.OnWSError(Sender: TObject; const AError: string);
begin
  TThread.Synchronize(nil, procedure
  begin
    LabelStatus.Text := 'Error: ' + AError;
```

```
  end);
end;

function TFormPriceFeed.GetChangeColor(AChange: Double): string;
begin
  if AChange > 0 then
    Result := '#27ae60'
  else if AChange < 0 then
    Result := '#e74c3c'
  else
    Result := '#666';
end;

function TFormPriceFeed.GetChangeArrow(AChange: Double): string;
begin
  if AChange > 0 then
    Result := '&#9650;'  // up triangle
  else if AChange < 0 then
    Result := '&#9660;'  // down triangle
  else
    Result := '&#9644;'; // dash
end;

procedure TFormPriceFeed.UpdatePriceDisplay;
var
  HTML: TStringBuilder;
  Pair: TPair<string, TPriceInfo>;
begin
  HTML := TStringBuilder.Create;
  try
    HTML.AppendLine('<div style="font-family: Arial, sans-serif; padding: 15px;">');
    HTML.AppendLine('<h2 style="color: #2c3e50;">Live Price Feed</h2>');
    HTML.AppendLine('<table style="width: 100%; border-collapse: collapse;">');
    HTML.AppendLine('<thead>');
    HTML.AppendLine('<tr style="background: #2c3e50; color: white;">');
    HTML.AppendLine('  <th style="padding: 10px; text-align: left;">Symbol</th>');
    HTML.AppendLine('  <th style="padding: 10px; text-align: right;">Price</th>');
    HTML.AppendLine('  <th style="padding: 10px; text-align: right;">Change</th>');
    HTML.AppendLine('  <th style="padding: 10px; text-align: right;">%</th>');
    HTML.AppendLine('  <th style="padding: 10px; text-align: right;">Updated</th>');
    HTML.AppendLine('</tr>');
    HTML.AppendLine('</thead>');
    HTML.AppendLine('<tbody>');

    for Pair in FPrices do
    begin
      var Color := GetChangeColor(Pair.Value.Change);
      var Arrow := GetChangeArrow(Pair.Value.Change);

      HTML.AppendFormat(
        '<tr style="border-bottom: 1px solid #eee;">' +
        '  <td style="padding: 10px; font-weight: bold;">%s</td>' +
        '  <td style="padding: 10px; text-align: right; font-size: 1.1em;">$%.2f</td>' +
        '  <td style="padding: 10px; text-align: right; color: %s;">%s %.2f</td>' +
        '  <td style="padding: 10px; text-align: right; color: %s;">%.2f%%</td>' +
        '  <td style="padding: 10px; text-align: right; color: #999; font-size: 0.85em;">%s</td>
        '</tr>',
        [Pair.Value.Symbol, Pair.Value.Price, Color, Arrow,
         Abs(Pair.Value.Change), Color, Pair.Value.ChangePercent,
```

```
          FormatDateTime('hh:nn:ss', Pair.Value.LastUpdate)]);
      end;

      HTML.AppendLine('</tbody>');
      HTML.AppendLine('</table>');

      if FPrices.Count = 0 then
        HTML.AppendLine('<p style="color: #999; text-align: center; padding: 40px;">Waiting for pr

      HTML.AppendLine('</div>');

      HTMLRender1.HTML.Text := HTML.ToString;
    finally
      HTML.Free;
    end;
  end;

  end.
```

---

10. Exercise: Notification System

Build a notification system with these requirements:

- Connect to a WebSocket server at `wss://api.example.com/notifications`
- Receive notifications as JSON with `id`, `title`, `message`, `type` (info, warning, error), and `timestamp` fields
- Display notifications as toast-style messages in an HTML renderer
- Show a badge counter of unread notifications
- Clicking a notification marks it as read (send `{"type": "mark_read", "id": "..."}` back to the server)
- Notifications fade out after 10 seconds unless marked as read manually

Solution

```
unit NotificationForm;

interface

uses
  System.SysUtils, System.Types, System.Classes, System.JSON,
  System.Generics.Collections,
  FMX.Types, FMX.Controls, FMX.Forms, FMX.StdCtrls, FMX.Layouts,
  FMX.Objects,
  Tina4WebSocketClient, Tina4HTMLRender, Tina4Core;

type
  TNotification = record
    ID: string;
    Title: string;
    Message: string;
    NotifType: string;  // info, warning, error
    Timestamp: TDateTime;
    IsRead: Boolean;
  end;

  TFormNotifications = class(TForm)
    Tina4WebSocket1: TTina4WebSocketClient;
    HTMLRender1: TTina4HTMLRender;
    LabelBadge: TLabel;
    LabelStatus: TLabel;
    TimerFade: TTimer;
    procedure FormCreate(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
    procedure TimerFadeTimer(Sender: TObject);
  private
    FNotifications: TList<TNotification>;
    procedure OnWSOpen(Sender: TObject);
    procedure OnWSMessage(Sender: TObject; const AMessage: string);
    procedure OnWSClose(Sender: TObject; const ACode: Integer;
      const AReason: string);
    procedure RefreshDisplay;
    procedure UpdateBadge;
    procedure MarkAsRead(const AID: string);
    function GetTypeColor(const AType: string): string;
    function GetTypeIcon(const AType: string): string;
    function UnreadCount: Integer;
  end;

var
  FormNotifications: TFormNotifications;

implementation

{$R *.fmx}

procedure TFormNotifications.FormCreate(Sender: TObject);
begin
  FNotifications := TList<TNotification>.Create;

  Tina4WebSocket1.URL := 'wss://api.example.com/notifications';
  Tina4WebSocket1.AutoReconnect := True;
```

```
  Tina4WebSocket1.ReconnectInterval := 5000;
  Tina4WebSocket1.PingInterval := 30000;
  Tina4WebSocket1.OnConnected := OnWSOpen;
  Tina4WebSocket1.OnMessage := OnWSMessage;
  Tina4WebSocket1.OnDisconnected := OnWSClose;

  // Timer checks for notifications older than 10 seconds
  TimerFade.Interval := 2000;
  TimerFade.Enabled := True;

  // Register click handler for mark-as-read
  HTMLRender1.RegisterObject('Notif', Self);

  Tina4WebSocket1.Connect;
  LabelStatus.Text := 'Connecting...';
  UpdateBadge;
  RefreshDisplay;
end;

procedure TFormNotifications.FormDestroy(Sender: TObject);
begin
  Tina4WebSocket1.AutoReconnect := False;
  if Tina4WebSocket1.IsConnected then
    Tina4WebSocket1.Disconnect;
  FNotifications.Free;
end;

procedure TFormNotifications.OnWSOpen(Sender: TObject);
begin
  TThread.Synchronize(nil, procedure
  begin
    LabelStatus.Text := 'Connected';
  end);
end;

procedure TFormNotifications.OnWSMessage(Sender: TObject; const AMessage: string);
begin
  TThread.Synchronize(nil, procedure
  var
    JSON: TJSONObject;
    Notif: TNotification;
  begin
    JSON := StrToJSONObject(AMessage);
    if not Assigned(JSON) then Exit;
    try
      var MsgType := JSON.GetValue<String>('type', '');

      if (MsgType = 'notification') or (MsgType = 'info') or
         (MsgType = 'warning') or (MsgType = 'error') then
      begin
        Notif.ID := JSON.GetValue<String>('id', GetGUID);
        Notif.Title := JSON.GetValue<String>('title', 'Notification');
        Notif.Message := JSON.GetValue<String>('message', '');
        Notif.NotifType := JSON.GetValue<String>('type', 'info');
        Notif.Timestamp := Now;
        Notif.IsRead := False;

        FNotifications.Insert(0, Notif);  // Newest first
        UpdateBadge;
```

```
        RefreshDisplay;
      end;
    finally
      JSON.Free;
    end;
  end);
end;

procedure TFormNotifications.OnWSClose(Sender: TObject; const ACode: Integer;
  const AReason: string);
begin
  TThread.Synchronize(nil, procedure
  begin
    LabelStatus.Text := 'Reconnecting...';
  end);
end;

procedure TFormNotifications.TimerFadeTimer(Sender: TObject);
var
  Changed: Boolean;
  I: Integer;
begin
  Changed := False;

  // Remove unread notifications older than 10 seconds
  for I := FNotifications.Count - 1 downto 0 do
  begin
    var Notif := FNotifications[I];
    if (not Notif.IsRead) and (SecondsBetween(Now, Notif.Timestamp) > 10) then
    begin
      Notif.IsRead := True;
      FNotifications[I] := Notif;
      Changed := True;
    end;
  end;

  if Changed then
  begin
    UpdateBadge;
    RefreshDisplay;
  end;
end;

procedure TFormNotifications.MarkAsRead(const AID: string);
var
  I: Integer;
begin
  for I := 0 to FNotifications.Count - 1 do
  begin
    var Notif := FNotifications[I];
    if Notif.ID = AID then
    begin
      Notif.IsRead := True;
      FNotifications[I] := Notif;

      // Notify the server
      if Tina4WebSocket1.IsConnected then
      begin
        var Msg := TJSONObject.Create;
```

```
        try
          Msg.AddPair('type', 'mark_read');
          Msg.AddPair('id', AID);
          Tina4WebSocket1.Send(Msg.ToString);
        finally
          Msg.Free;
        end;
      end;

      UpdateBadge;
      RefreshDisplay;
      Break;
    end;
  end;
end;

function TFormNotifications.UnreadCount: Integer;
var
  I: Integer;
begin
  Result := 0;
  for I := 0 to FNotifications.Count - 1 do
    if not FNotifications[I].IsRead then
      Inc(Result);
end;

procedure TFormNotifications.UpdateBadge;
var
  Count: Integer;
begin
  Count := UnreadCount;
  if Count > 0 then
  begin
    LabelBadge.Text := Count.ToString;
    LabelBadge.Visible := True;
  end
  else
    LabelBadge.Visible := False;
end;

function TFormNotifications.GetTypeColor(const AType: string): string;
begin
  if AType = 'error' then
    Result := '#e74c3c'
  else if AType = 'warning' then
    Result := '#f39c12'
  else
    Result := '#3498db';
end;

function TFormNotifications.GetTypeIcon(const AType: string): string;
begin
  if AType = 'error' then
    Result := '&#10060;'
  else if AType = 'warning' then
    Result := '&#9888;'
  else
    Result := '&#8505;';
end;
```

```
procedure TFormNotifications.RefreshDisplay;
var
  HTML: TStringBuilder;
  Notif: TNotification;
begin
  HTML := TStringBuilder.Create;
  try
    HTML.AppendLine('<div style="font-family: Arial, sans-serif; padding: 10px;">');
    HTML.AppendLine('<h3 style="color: #2c3e50;">Notifications</h3>');

    if FNotifications.Count = 0 then
    begin
      HTML.AppendLine('<p style="color: #999; text-align: center; padding: 30px;">No notificatio
    end
    else
    begin
      for Notif in FNotifications do
      begin
        var Color := GetTypeColor(Notif.NotifType);
        var Icon := GetTypeIcon(Notif.NotifType);
        var Opacity: string;
        if Notif.IsRead then
          Opacity := '0.5'
        else
          Opacity := '1.0';

        HTML.AppendFormat(
          '<div style="border-left: 4px solid %s; padding: 10px 15px; margin: 8px 0; ' +
          '  background: white; border-radius: 0 4px 4px 0; opacity: %s; ' +
          '  box-shadow: 0 1px 3px rgba(0,0,0,0.1); cursor: pointer;" ' +
          '  onclick="Notif:MarkAsRead(''%s'')">' +
          '  <div style="display: flex; justify-content: space-between;">' +
          '    <strong>%s %s</strong>' +
          '    <small style="color: #999;">%s</small>' +
          '  </div>' +
          '  <p style="margin: 5px 0 0; color: #555;">%s</p>' +
          '  %s' +
          '</div>',
          [Color, Opacity, Notif.ID, Icon, Notif.Title,
           FormatDateTime('hh:nn:ss', Notif.Timestamp),
           Notif.Message,
           IfThen(not Notif.IsRead,
             '<small style="color: ' + Color + ';">Click to mark as read</small>', '')]);
      end;
    end;

    HTML.AppendLine('</div>');
    HTMLRender1.HTML.Text := HTML.ToString;
  finally
    HTML.Free;
  end;
end;

end.
```

Common Gotchas

Connection lifecycle. The WebSocket connection runs on a background thread. Every event handler fires on that background thread. If you touch any UI control without `TThread.Synchronize`, your application will crash with access violations, or worse, silently corrupt UI state. Always wrap UI updates:

```
// Wrong -- will crash
procedure TForm1.OnWSMessage(Sender: TObject; const AMessage: string);
begin
  Label1.Text := AMessage;  // Access violation!
end;

// Right
procedure TForm1.OnWSMessage(Sender: TObject; const AMessage: string);
begin
  TThread.Synchronize(nil, procedure
  begin
    Label1.Text := AMessage;
  end);
end;
```

Thread safety with shared data. If your message handler writes to a `TList` or `TDictionary` that the UI thread also reads, you need synchronization. The simplest approach is to do everything inside `TThread.Synchronize`. For high-frequency messages, consider a thread-safe queue.

Reconnect re-subscription. When auto-reconnect opens a new connection, the server does not remember your subscriptions from the previous connection. Always re-subscribe in the `OnConnected` handler:

```
procedure TForm1.OnWSOpen(Sender: TObject);
begin
  TThread.Synchronize(nil, procedure
  begin
    // This runs on every connect, including reconnects
    Tina4WebSocket1.Send('{"type": "subscribe", "channel": "updates"}');
  end);
end;
```

Memory management with TJSONObject. Every `TJSONObject` or `TJSONArray` you create from parsing must be freed. In the message handler, parse once, extract what you need, free immediately:

```
procedure TForm1.OnWSMessage(Sender: TObject; const AMessage: string);
var
  JSON: TJSONObject;
begin
  JSON := StrToJSONObject(AMessage);
  if not Assigned(JSON) then Exit;
  try
    // Extract values here
    var Name := JSON.GetValue<String>('name', '');
    // Use the values...
  finally
    JSON.Free;  // Always free
  end;
```

```
end;
```

Disconnect before destroy. Always disconnect the WebSocket before the form is destroyed. If the background thread tries to fire an event after the form is freed, you get a use-after-free crash:

```
procedure TForm1.FormDestroy(Sender: TObject);
begin
  Tina4WebSocket1.AutoReconnect := False;  // Stop reconnection first
  if Tina4WebSocket1.IsConnected then
    Tina4WebSocket1.Disconnect;
end;
```

Secure connections (WSS). For `wss://` URLs, you need the same OpenSSL DLLs required for HTTPS REST calls. Without them, the connection will fail silently. See the Installation chapter for SSL setup details.

Socket Server

Raw TCP Without the Ceremony

Sometimes WebSocket is too much. You don't need HTTP upgrade handshakes, frame masking, or RFC 6455 compliance. You need a raw TCP socket that listens on a port and handles bytes. A hardware device sends telemetry. A legacy system speaks a custom protocol. A game server needs low-latency binary messaging.

TTina4SocketServer is the answer. Drop it on a form, set a host and port, flip `Active` to `True`, and handle incoming bytes in the `OnMessage` event. The component manages the listener thread, accepts connections, and dispatches messages -- you focus on your protocol.

---

1. TTina4SocketServer Overview

The socket server provides:

- **TCP and UDP support** -- choose the socket type for your use case
- **Async accept loop** -- runs in a background task, never blocks your UI
- **Per-connection processing** -- each client gets its own receive loop
- **Event-driven messages** -- `OnMessage` fires with the client socket and raw bytes
- **Simple lifecycle** -- `Active := True` to start, `Active := False` to stop

Component Setup

Drop a `TTina4SocketServer` from the Tina4Delphi palette onto your form, or create it at runtime:

```
uses
  Tina4SocketServer, System.Net.Socket;

var
  Server: TTina4SocketServer;
begin
  Server := TTina4SocketServer.Create(Self);
  Server.Host := '0.0.0.0';
  Server.Port := 9000;
  Server.SocketType := TSocketType.TCP;
  Server.OnMessage := HandleMessage;
  Server.Active := True;
end;
```

---

2. Properties

Set these in the Object Inspector or at runtime:

Property	Type	Default	Description

`Host`	`String`	`''`	Bind address. Use `0.0.0.0` for all interfaces, `127.0.0.1` for localhost only
`Port`	`Integer`	`0`	Port number to listen on
`SocketType`	`TSocketType`	`TCP`	`TSocketType.TCP` or `TSocketType.UDP`
`Active`	`Boolean`	`False`	Set `True` to start listening, `False` to stop
`OnMessage`	`TTina4SocketEvent`	`nil`	Event handler fired when data arrives from a client

Event Signature

```
type
  TTina4SocketEvent = procedure(const Client: TSocket; Content: TBytes) of object;
```

The `Client` parameter is the connected client's socket -- use it to send responses back. `Content` is the raw bytes received.

---

3. Basic TCP Server

Design-Time Setup

- Drop `TTina4SocketServer` on your form
- Set `Host` to `0.0.0.0`
- Set `Port` to `9000`
- Set `SocketType` to `TCP`
- Double-click `OnMessage` to create the event handler
- Set `Active` to `True`

Handling Messages

```
procedure TForm1.Tina4SocketServer1Message(const Client: TSocket; Content: TBytes);
var
  Text: string;
  Response: TBytes;
begin
  // Decode incoming bytes to string
  Text := TEncoding.UTF8.GetString(Content);
  Memo1.Lines.Add('Received: ' + Text);

  // Send a response back to the client
  Response := TEncoding.UTF8.GetBytes('ACK: ' + Text);
  Client.Send(Response);
end;
```

Starting and Stopping

```
procedure TForm1.btnStartClick(Sender: TObject);
begin
  Tina4SocketServer1.Host := '0.0.0.0';
  Tina4SocketServer1.Port := StrToInt(edtPort.Text);
  Tina4SocketServer1.Active := True;
  lblStatus.Text := 'Listening on port ' + edtPort.Text;
end;

procedure TForm1.btnStopClick(Sender: TObject);
begin
  Tina4SocketServer1.Active := False;
  lblStatus.Text := 'Stopped';
end;
```

---

4. Practical Examples

Echo Server

The simplest possible server -- sends back whatever it receives:

```
procedure TForm1.Tina4SocketServer1Message(const Client: TSocket; Content: TBytes);
begin
  Client.Send(Content);
end;
```

JSON Command Server

Parse incoming JSON commands and respond with JSON:

```
uses
  JSON;

procedure TForm1.Tina4SocketServer1Message(const Client: TSocket; Content: TBytes);
var
  Request, Response: TJSONObject;
  Command: string;
begin
  try
    Request := TJSONObject.ParseJSONValue(Content, 0, Length(Content)) as TJSONObject;
    try
```

```
      Command := Request.GetValue<string>('command');

      Response := TJSONObject.Create;
      try
        if Command = 'ping' then
        begin
          Response.AddPair('status', 'pong');
          Response.AddPair('timestamp', DateTimeToStr(Now));
        end
        else if Command = 'status' then
        begin
          Response.AddPair('status', 'ok');
          Response.AddPair('clients', TJSONNumber.Create(1));
          Response.AddPair('uptime', FormatDateTime('hh:nn:ss', Now - FStartTime));
        end
        else
          Response.AddPair('error', 'unknown command: ' + Command);

        Client.Send(TEncoding.UTF8.GetBytes(Response.ToJSON));
      finally
        Response.Free;
      end;
    finally
      Request.Free;
    end;
  except
    on E: Exception do
      Client.Send(TEncoding.UTF8.GetBytes('{"error":"' + E.Message + '"}'));
  end;
end;
```

Telemetry Receiver

Receive fixed-size binary packets from a hardware device:

```
type
  TTelemetryPacket = packed record
    DeviceID: UInt16;
    Temperature: Single;
    Humidity: Single;
    BatteryPct: Byte;
  end;

procedure TForm1.Tina4SocketServer1Message(const Client: TSocket; Content: TBytes);
var
  Packet: TTelemetryPacket;
begin
  if Length(Content) >= SizeOf(TTelemetryPacket) then
  begin
    Move(Content[0], Packet, SizeOf(TTelemetryPacket));

    TThread.Synchronize(nil,
      procedure
      begin
        Memo1.Lines.Add(Format('Device %d: %.1f C, %.1f%% humidity, %d%% battery',
          [Packet.DeviceID, Packet.Temperature, Packet.Humidity, Packet.BatteryPct]));
      end);

    // Acknowledge receipt
```

```
    Client.Send(TEncoding.UTF8.GetBytes('OK'));
  end;
end;
```

---

5. Thread Safety

The `OnMessage` event fires on a background thread, not the main UI thread. To update UI controls, use `TThread.Synchronize` or `TThread.Queue`:

```
procedure TForm1.Tina4SocketServer1Message(const Client: TSocket; Content: TBytes);
var
  Text: string;
begin
  Text := TEncoding.UTF8.GetString(Content);

  // Safe UI update from background thread
  TThread.Queue(nil,
    procedure
    begin
      Memo1.Lines.Add(Text);
    end);
end;
```

`Synchronize` blocks the background thread until the UI update completes -- use it when you need the result before sending a response. `Queue` is fire-and-forget -- use it for logging and display updates where you don't need to wait.

---

6. Lifecycle

The server lifecycle is controlled entirely by the `Active` property:

```
Active := True
  -> Creates TSocket with configured SocketType
  -> Calls Listen(Host, '', Port)
  -> Starts background TTask
  -> BeginAccept loop runs until stopped
  -> Each accepted client gets its own receive loop

Active := False
  -> Sets CanRun := False
  -> Cancels the background task
  -> Client connections close on next receive cycle
```

The component cleans up automatically. Setting `Active := False` stops the accept loop, and each client connection closes when its receive loop detects the shutdown flag.

---

7. When to Use Socket Server vs WebSocket Client

Use Case	Component
Connect TO a WebSocket service	`TTina4WebSocketClient`
Accept raw TCP/UDP connections	`TTina4SocketServer`
Hardware device telemetry	`TTina4SocketServer`
Chat/notification from a web backend	`TTina4WebSocketClient`
Custom binary protocol	`TTina4SocketServer`
Browser-compatible real-time	`TTina4WebSocketClient`

TTina4WebSocketClient is a **client** that connects outward to a WebSocket server. TTina4SocketServer is a **server** that listens for incoming TCP or UDP connections. They solve different problems and can be used together in the same application.

---

8. Gotchas

- **Thread safety** -- `OnMessage` fires on a background thread. Always use `TThread.Synchronize` or `TThread.Queue` for UI updates. Forgetting this causes access violations.
- **Blocking receive loop** -- Each client connection runs a `Sleep(1000)` receive loop. This means message latency can be up to 1 second. For sub-second requirements, consider the WebSocket client instead.
- **No TLS** -- The raw socket server does not support TLS/SSL. For encrypted connections, terminate TLS at a reverse proxy or use the WebSocket client which supports `wss://`.
- **Port conflicts** -- Ensure your chosen port is not already in use. On mobile platforms (Android/iOS), binding to low ports (below 1024) may require special permissions.
- **Firewall** -- When deploying, ensure the server port is open in the OS firewall and any network firewalls between client and server.

Core Utilities

The Swiss Army Knife You Already Have

Every Delphi project accumulates a `Utils.pas` file. String conversion functions. Date formatting helpers. JSON parsing wrappers. HTTP request boilerplate. File encoding routines. You write them once, copy them across projects, fix bugs in three places, forget to fix the fourth.

`Tina4Core.pas` is that utils file, already written and tested. Add it to your `uses` clause and you get string helpers, GUID generation, date utilities, Base64 encoding, JSON parsing, database-to-JSON conversion, JSON-to-MemTable population, HTTP requests, multipart file uploads, and shell command execution. No components to drop on a form. No configuration. Just functions.

```
uses
  Tina4Core;
```

That single line gives you everything in this chapter.

---

1. String Helpers

CamelCase

Converts `snake_case` to `camelCase`. This is the function that makes database field names feel natural in JSON:

```
CamelCase('first_name');      // 'firstName'
CamelCase('id');              // 'id'
CamelCase('user_email');      // 'userEmail'
CamelCase('created_at');      // 'createdAt'
CamelCase('order_line_item'); // 'orderLineItem'
```

`GetJSONFromDB` uses this internally. When your database table has `first_name` and `last_name` columns, the JSON output automatically uses `firstName` and `lastName`.

SnakeCase

The reverse -- converts `camelCase` back to `snake_case`. Used when mapping JSON keys back to database columns:

```
SnakeCase('firstName');      // 'first_name'
SnakeCase('userEmail');      // 'user_email'
SnakeCase('createdAt');      // 'created_at'
SnakeCase('orderLineItem');  // 'order_line_item'
```

When You Need Both

```
// API sends camelCase JSON, database uses snake_case columns
var JSONKey := 'orderTotal';
var DBColumn := SnakeCase(JSONKey);  // 'order_total'

// Query returns snake_case, API expects camelCase
var DBField := 'shipping_address';
var APIField := CamelCase(DBField);  // 'shippingAddress'
```

---

2. GUID Generation

GetGUID

Returns a new GUID string without braces:

```
var ID := GetGUID;
// e.g. 'A1B2C3D4-E5F6-7890-ABCD-EF1234567890'

// Use as a primary key
FDQuery1.SQL.Text := 'INSERT INTO documents (id, title) VALUES (:id, :title)';
FDQuery1.ParamByName('id').AsString := GetGUID;
FDQuery1.ParamByName('title').AsString := 'New Document';
FDQuery1.ExecSQL;
```

Each call produces a unique value. Use it for primary keys, session tokens, file names, or any situation that needs a globally unique identifier.

```
// Generate unique filenames
var FileName := GetGUID + '.pdf';

// Create a batch ID for grouped operations
var BatchID := GetGUID;
for var I := 0 to Items.Count - 1 do
begin
  FDQuery1.SQL.Text := 'INSERT INTO batch_items (batch_id, item_id) VALUES (:batch, :item)';
  FDQuery1.ParamByName('batch').AsString := BatchID;
  FDQuery1.ParamByName('item').AsString := Items[I];
  FDQuery1.ExecSQL;
end;
```

---

3. Date Utilities

IsDate

Checks whether a Variant value represents a valid date. Supports multiple formats:

```
IsDate('2024-01-15');                  // True - ISO date
IsDate('2024-01-15T10:30:00');         // True - ISO datetime
IsDate('2024-01-15T10:30:00.000Z');    // True - ISO with milliseconds
IsDate('01/15/2024');                  // True - US date format
IsDate('2024-01-15 14:30:00');         // True - datetime with space
IsDate('42');                          // False - number
IsDate('hello');                       // False - string
```

```
IsDate('');                              // False - empty
```

Use it to validate user input or incoming API data:

```
procedure TForm1.ValidateDateField(const AValue: string);
begin
  if not IsDate(AValue) then
    raise Exception.Create('Invalid date format: ' + AValue);
end;
```

GetJSONDate

Converts a `TDateTime` to an ISO 8601 string -- the standard format for JSON APIs:

```
GetJSONDate(Now);
// '2026-03-26T14:30:00.000Z'

GetJSONDate(EncodeDate(2026, 1, 1));
// '2026-01-01T00:00:00.000Z'
```

Use it when building JSON payloads:

```
var Order := TJSONObject.Create;
try
  Order.AddPair('id', GetGUID);
  Order.AddPair('createdAt', GetJSONDate(Now));
  Order.AddPair('total', TJSONNumber.Create(99.99));
  // Send to API...
finally
  Order.Free;
end;
```

JSONDateToDateTime

Converts an ISO 8601 date string back to a `TDateTime`:

```
var DT := JSONDateToDateTime('2026-03-26T14:30:00.000Z');
// DT is now a TDateTime you can use with FormatDateTime, DateUtils, etc.

ShowMessage(FormatDateTime('dd/mm/yyyy hh:nn', DT));
// '26/03/2026 14:30'
```

The round-trip works cleanly:

```
var Original := Now;
var JSON := GetJSONDate(Original);
var Restored := JSONDateToDateTime(JSON);
// Original and Restored represent the same point in time
```

---

4. Encoding

DecodeBase64

Decodes a Base64-encoded string back to plain UTF-8:

```
DecodeBase64('SGVsbG8gV29ybGQ=');
// 'Hello World'

// Decode an API token
```

```
var Token := DecodeBase64(EncodedToken);
```

FileToBase64

Reads an entire file and returns its content as a Base64 string. Works with any file type:

```
var B64 := FileToBase64('C:\photos\avatar.jpg');
// B64 now contains the Base64-encoded JPEG data

// Embed in JSON for API upload
var Payload := TJSONObject.Create;
try
  Payload.AddPair('filename', 'avatar.jpg');
  Payload.AddPair('data', FileToBase64('C:\photos\avatar.jpg'));
  Payload.AddPair('mimeType', 'image/jpeg');
  // POST to API...
finally
  Payload.Free;
end;
```

BitmapToBase64EncodedString

Encodes an FMX `TBitmap` to a Base64 string with optional resizing. The default resize is 256x256 pixels -- ideal for thumbnails and avatars:

```
// Default: resize to 256x256
var Encoded := BitmapToBase64EncodedString(Image1.Bitmap);

// No resize -- keep original dimensions
var Encoded := BitmapToBase64EncodedString(Image1.Bitmap, False);

// Custom resize
var Encoded := BitmapToBase64EncodedString(Image1.Bitmap, True, 128, 128);
var Encoded := BitmapToBase64EncodedString(Image1.Bitmap, True, 512, 512);
```

Practical use -- capture and upload a profile photo:

```
procedure TForm1.ButtonUploadClick(Sender: TObject);
var
  StatusCode: Integer;
  Encoded: string;
begin
  Encoded := BitmapToBase64EncodedString(ImageAvatar.Bitmap, True, 200, 200);

  var Body := TJSONObject.Create;
  try
    Body.AddPair('userId', '1001');
    Body.AddPair('avatar', Encoded);

    SendHttpRequest(StatusCode,
      'https://api.example.com', '/users/1001/avatar', '',
      Body.ToString, 'application/json', 'utf-8', '', '', nil, 'Tina4Delphi',
      TTina4RequestType.Patch);

    if StatusCode = 200 then
      ShowMessage('Avatar uploaded')
    else
      ShowMessage('Upload failed: ' + StatusCode.ToString);
  finally
    Body.Free;
```

```
  end;
end;
```

BitmapToSkiaWepPEncodedString

Encodes an FMX `TBitmap` to a WebP Base64 string using Skia. Requires the `SKIA` compiler define. WebP produces smaller files than JPEG at the same quality:

```
{$IFDEF SKIA}
var WebPData := BitmapToSkiaWepPEncodedString(Image1.Bitmap, 90);
// quality parameter: 0-100 (higher = better quality, larger size)
{$ENDIF}
```

---

5. JSON Parsing

StrToJSONObject

Parses a JSON string into a `TJSONObject`. Returns `nil` if parsing fails -- always check:

```
var Obj := StrToJSONObject('{"name": "Andre", "age": 30}');
try
  if Assigned(Obj) then
  begin
    var Name := Obj.GetValue<String>('name');   // 'Andre'
    var Age := Obj.GetValue<Integer>('age');     // 30
    ShowMessage(Name + ' is ' + Age.ToString);
  end
  else
    ShowMessage('Invalid JSON');
finally
  Obj.Free;
end;
```

StrToJSONArray

Parses a JSON array string:

```
var Arr := StrToJSONArray('[1, 2, 3, 4, 5]');
try
  if Assigned(Arr) then
    for var I := 0 to Arr.Count - 1 do
      ShowMessage(Arr.Items[I].Value);
finally
  Arr.Free;
end;
```

StrToJSONValue

Parses any JSON value -- object, array, string, number, boolean, or null. Use when you do not know the structure in advance:

```
var Val := StrToJSONValue(APIResponse);
if Val is TJSONObject then
  // Handle object
else if Val is TJSONArray then
  // Handle array
else if Val is TJSONString then
```

```
  // Handle string
```

BytesToJSONObject

Parses a `TBytes` buffer into a `TJSONObject`. This is the bridge between raw HTTP responses and structured JSON:

```
var
  StatusCode: Integer;
  Response: TBytes;
begin
  Response := SendHttpRequest(StatusCode, 'https://api.example.com', '/users');

  var JSON := BytesToJSONObject(Response);
  try
    if Assigned(JSON) then
    begin
      // Process the response
      var Records := JSON.GetValue<TJSONArray>('records');
      ShowMessage('Found ' + Records.Count.ToString + ' users');
    end;
  finally
    JSON.Free;
  end;
end;
```

GetJSONFieldName

Strips surrounding quotes from a JSON field name string:

```
GetJSONFieldName('"firstName"');  // 'firstName'
GetJSONFieldName('"id"');         // 'id'
GetJSONFieldName('name');         // 'name' (no change)
```

---

6. Database to JSON

GetJSONFromDB

Executes a SQL query and returns the results as a `TJSONObject`. Three automatic conversions happen:

- **Field names** are converted from `snake_case` to `camelCase`
- **DateTime fields** are formatted as ISO 8601
- **Blob fields** are encoded as Base64

```
// Simple query
var Result := GetJSONFromDB(FDConnection1, 'SELECT * FROM users');
// {"records": [{"id": "1", "firstName": "Andre", "email": "andre@test.com"}, ...]}

// Custom dataset key
var Result := GetJSONFromDB(FDConnection1, 'SELECT * FROM products', nil, 'products');
// {"products": [{"id": "1", "productName": "Widget"}, ...]}
```

With Parameters

```
var Params := TFDParams.Create;
try
  Params.Add('status', 'active');
  Params.Add('minAge', 18);

  var Result := GetJSONFromDB(FDConnection1,
    'SELECT * FROM users WHERE status = :status AND age >= :minAge',
    Params);
  try
    Memo1.Lines.Text := Result.Format(2);
  finally
    Result.Free;
  end;
finally
  Params.Free;
end;
```

Serving JSON from a Database

Combine `GetJSONFromDB` with an HTTP response to build an instant API:

```
procedure TForm1.HandleGetUsers;
var
  Result: TJSONObject;
begin
  Result := GetJSONFromDB(FDConnection1,
    'SELECT id, first_name, last_name, email, created_at ' +
    'FROM users WHERE active = 1 ORDER BY last_name');
  try
    // Result is ready to send as an API response:
    // {
    //   "records": [
    //     {"id": "1", "firstName": "Andre", "lastName": "Van Zuydam",
    //      "email": "andre@test.com", "createdAt": "2026-03-15T10:30:00.000Z"},
    //     ...
    //   ]
    // }
    Memo1.Lines.Text := Result.Format(2);
  finally
    Result.Free;
  end;
end;
```

GetJSONFromTable

Converts rows in a `TFDMemTable` or `TFDTable` to JSON:

```
// Basic conversion
var JSON := GetJSONFromTable(FDMemTable1);
// {"records": [{"id": "1", "name": "Item 1"}, ...]}

// Ignore specific fields (e.g., sensitive data)
var JSON := GetJSONFromTable(FDMemTable1, 'records', 'password,secret_key');

// Ignore blank values (smaller JSON)
var JSON := GetJSONFromTable(FDMemTable1, 'records', '', True);
```

---

7. JSON to MemTable

GetFieldDefsFromJSONObject

Creates field definitions on a `TFDMemTable` from a `TJSONObject` structure. Call this before populating the table if you need auto-created fields:

```
var JSONObj := StrToJSONObject(
  '{"firstName": "Andre", "age": 30, "address": {"city": "Cape Town"}}');
try
  GetFieldDefsFromJSONObject(JSONObj, FDMemTable1, True);
  // Creates fields:
  //   first_name (ftString)  -- camelCase converted to snake_case
  //   age (ftString)
  //   address (ftMemo)       -- nested objects become memo fields
finally
  JSONObj.Free;
end;
```

The third parameter controls snake_case conversion:

```
// With snake_case conversion (True) -- good for database-style field names
GetFieldDefsFromJSONObject(JSONObj, MemTable, True);
// firstName -> first_name

// Without conversion (False) -- keeps camelCase
GetFieldDefsFromJSONObject(JSONObj, MemTable, False);
// firstName -> firstName
```

PopulateMemTableFromJSON

Populates a `TFDMemTable` from a JSON string. Two sync modes control how existing data is handled:

Clear mode (default) -- empties the table, then appends all records:

```
PopulateMemTableFromJSON(FDMemTable1, 'records',
  '{"records": [{"id": "1", "name": "Alice"}, {"id": "2", "name": "Bob"}]}');
// FDMemTable1 now has exactly 2 records
```

Sync mode -- matches existing records by a key field, updates them, and inserts new ones:

```
// First load
PopulateMemTableFromJSON(FDMemTable1, 'records',
  '{"records": [{"id": "1", "name": "Alice"}, {"id": "2", "name": "Bob"}]}');

// Later: sync with updated data
PopulateMemTableFromJSON(FDMemTable1, 'records',
  '{"records": [{"id": "1", "name": "Alice Updated"}, {"id": "3", "name": "Charlie"}]}',
  'id', TTina4RestSyncMode.Sync);
// FDMemTable1 now has 3 records:
//   id=1: "Alice Updated" (updated)
//   id=2: "Bob" (unchanged)
//   id=3: "Charlie" (inserted)
```

PopulateTableFromJSON

Inserts or updates rows directly into a **database table** (not a MemTable) from JSON. Uses a primary key for upsert logic:

```
var Result := PopulateTableFromJSON(
  FDConnection1,
  'users',                    // table name
  '{"response": [{"name": "Alice"}, {"name": "Bob"}]}',
  'response',                 // JSON key containing the array
  'id');                      // primary key field for upsert

// Rows are inserted or updated directly in the 'users' table
```

This is the fastest way to sync remote API data into a local database.

---

8. HTTP Requests

SendHttpRequest

Low-level HTTP function that returns raw `TBytes`. Supports GET, POST, PATCH, PUT, and DELETE:

```
var
  StatusCode: Integer;
  Response: TBytes;
begin
  // Simple GET
  Response := SendHttpRequest(StatusCode,
    'https://api.example.com', '/users');

  if StatusCode = 200 then
  begin
    var JSON := BytesToJSONObject(Response);
    try
      // Process response...
    finally
      JSON.Free;
    end;
  end;
end;
```

POST with JSON Body

```
var
  StatusCode: Integer;
  Body: string;
begin
  Body := '{"name": "Andre", "email": "andre@test.com"}';

  SendHttpRequest(StatusCode,
    'https://api.example.com',   // base URL
    '/users',                     // endpoint
    '',                           // query params
    Body,                         // request body
    'application/json',           // content type
    'utf-8',                      // charset
    '', '',                       // username, password (for Basic Auth)
    nil,                          // custom headers
    'Tina4Delphi',               // user agent
    TTina4RequestType.Post);      // request type

  case StatusCode of
    201: ShowMessage('User created');
    400: ShowMessage('Bad request');
    401: ShowMessage('Unauthorized');
    500: ShowMessage('Server error');
  end;
end;
```

With Basic Auth

```
Response := SendHttpRequest(StatusCode,
  'https://api.example.com', '/secure/data',
  '', '',
  'application/json', 'utf-8',
  'myuser', 'mypassword');
```

PATCH and DELETE

```
// Update a user
SendHttpRequest(StatusCode,
  'https://api.example.com', '/users/1001', '',
  '{"name": "Andre Updated"}',
  'application/json', 'utf-8', '', '', nil, 'Tina4Delphi',
  TTina4RequestType.Patch);

// Delete a user
SendHttpRequest(StatusCode,
  'https://api.example.com', '/users/1001', '',
  '', 'application/json', 'utf-8', '', '', nil, 'Tina4Delphi',
  TTina4RequestType.Delete);
```

SendMultipartFormData

Sends a multipart/form-data POST for file uploads with optional form fields:

```
var
  StatusCode: Integer;
  Response: TBytes;
begin
  Response := SendMultipartFormData(
```

```
    StatusCode,
    'https://api.example.com',      // base URL
    'upload/avatar',                 // endpoint
    ['userId', '1001',              // form fields (key-value pairs)
     'caption', 'Profile photo'],
    ['avatar', 'C:\photos\me.jpg'], // files (field name, file path)
    '',                              // query params
    'myuser', 'mypassword');         // auth

  if StatusCode = 200 then
    ShowMessage('Upload successful');
end;
```

Multiple files:

```
Response := SendMultipartFormData(
  StatusCode,
  'https://api.example.com', 'upload/documents',
  ['projectId', '42'],
  ['doc1', 'C:\docs\spec.pdf',
   'doc2', 'C:\docs\design.pdf',
   'doc3', 'C:\docs\timeline.xlsx'],
  '', '', '');
```

---

9. Shell Commands

ExecuteShellCommand

Runs a shell command and captures stdout. Works on Windows, Linux, and macOS:

```
var
  Output: String;
  ExitCode: Integer;
begin
  // Windows
  ExitCode := ExecuteShellCommand('dir C:\temp', Output);
  ShowMessage(Output);

  // macOS / Linux
  ExitCode := ExecuteShellCommand('ls -la /tmp', Output);
  ShowMessage(Output);
end;
```

Check the exit code:

```
var
  Output: String;
  ExitCode: Integer;
begin
  ExitCode := ExecuteShellCommand('ping -n 4 google.com', Output);

  if ExitCode = 0 then
    ShowMessage('Ping successful:' + sLineBreak + Output)
  else
    ShowMessage('Ping failed with exit code: ' + ExitCode.ToString);
end;
```

Cross-Platform Commands

```
procedure TForm1.RunCommand(const ACommand: string);
var
  Output: String;
  ExitCode: Integer;
begin
  ExitCode := ExecuteShellCommand(ACommand, Output);
  MemoOutput.Lines.Text := Output;
  LabelExitCode.Text := 'Exit code: ' + ExitCode.ToString;
end;

// Usage:
{$IFDEF MSWINDOWS}
RunCommand('ipconfig /all');
{$ELSE}
RunCommand('ifconfig');
{$ENDIF}
```

---

10. Complete Example: File Upload Utility

Build a utility that selects an image, resizes it, encodes to Base64, uploads via multipart form data, and displays the server response.

```
unit UploadForm;

interface

uses
  System.SysUtils, System.Types, System.Classes, System.JSON,
  FMX.Types, FMX.Controls, FMX.Forms, FMX.StdCtrls, FMX.Objects,
  FMX.Dialogs, FMX.Layouts, FMX.Memo,
  Tina4Core;

type
  TFormUpload = class(TForm)
    ImagePreview: TImage;
    ButtonSelect: TButton;
    ButtonUpload: TButton;
    LabelStatus: TLabel;
    MemoResponse: TMemo;
    OpenDialog1: TOpenDialog;
    LabelFileInfo: TLabel;
    procedure ButtonSelectClick(Sender: TObject);
    procedure ButtonUploadClick(Sender: TObject);
  private
    FSelectedFile: string;
    procedure UpdateFileInfo;
  end;

var
  FormUpload: TFormUpload;

implementation

{$R *.fmx}
```

```
procedure TFormUpload.ButtonSelectClick(Sender: TObject);
begin
  OpenDialog1.Filter := 'Image files|*.jpg;*.jpeg;*.png;*.bmp|All files|*.*';

  if OpenDialog1.Execute then
  begin
    FSelectedFile := OpenDialog1.FileName;
    ImagePreview.Bitmap.LoadFromFile(FSelectedFile);
    UpdateFileInfo;
    ButtonUpload.Enabled := True;
    LabelStatus.Text := 'Image selected. Ready to upload.';
  end;
end;

procedure TFormUpload.UpdateFileInfo;
var
  FileSize: Int64;
  Info: TSearchRec;
begin
  if FindFirst(FSelectedFile, faAnyFile, Info) = 0 then
  begin
    FileSize := Info.Size;
    FindClose(Info);

    LabelFileInfo.Text := Format('File: %s | Size: %s | Dimensions: %dx%d',
      [ExtractFileName(FSelectedFile),
       FormatFloat('#,##0', FileSize) + ' bytes',
       Round(ImagePreview.Bitmap.Width),
       Round(ImagePreview.Bitmap.Height)]);
  end;
end;

procedure TFormUpload.ButtonUploadClick(Sender: TObject);
var
  StatusCode: Integer;
  Response: TBytes;
  Encoded: string;
begin
  if FSelectedFile.IsEmpty then
  begin
    ShowMessage('Please select an image first');
    Exit;
  end;

  LabelStatus.Text := 'Resizing image...';
  Application.ProcessMessages;

  // Resize and encode to Base64
  Encoded := BitmapToBase64EncodedString(ImagePreview.Bitmap, True, 512, 512);

  LabelStatus.Text := Format('Encoded to Base64 (%d chars). Uploading...',
    [Length(Encoded)]);
  Application.ProcessMessages;

  // Method 1: Upload as multipart form data (file upload)
  Response := SendMultipartFormData(
    StatusCode,
    'https://api.example.com',
    'upload/image',
```

```
    ['userId', '1001',
     'description', 'Uploaded from Delphi'],
    ['image', FSelectedFile],
    '', '', '');

  if StatusCode in [200, 201] then
  begin
    LabelStatus.Text := 'Upload successful!';

    var JSON := BytesToJSONObject(Response);
    try
      if Assigned(JSON) then
        MemoResponse.Lines.Text := JSON.Format(2)
      else
        MemoResponse.Lines.Text := TEncoding.UTF8.GetString(Response);
    finally
      JSON.Free;
    end;
  end
  else
  begin
    LabelStatus.Text := 'Upload failed: HTTP ' + StatusCode.ToString;
    MemoResponse.Lines.Text := TEncoding.UTF8.GetString(Response);
  end;
end;

end.
```

---

11. Complete Example: Database Sync Tool

Fetch JSON from a remote API, compare with a local database, and sync changes using `PopulateTableFromJSON`.

```
unit SyncForm;

interface

uses
  System.SysUtils, System.Types, System.Classes, System.JSON,
  FMX.Types, FMX.Controls, FMX.Forms, FMX.StdCtrls, FMX.Memo,
  FMX.Layouts, FMX.Dialogs,
  FireDAC.Comp.Client, FireDAC.Stan.Def, FireDAC.Stan.Async,
  FireDAC.Phys.SQLite, FireDAC.DApt,
  Tina4Core;

type
  TFormSync = class(TForm)
    FDConnection1: TFDConnection;
    FDMemTableLocal: TFDMemTable;
    FDMemTableRemote: TFDMemTable;
    ButtonSync: TButton;
    ButtonCompare: TButton;
    MemoLog: TMemo;
    LabelStatus: TLabel;
    procedure FormCreate(Sender: TObject);
    procedure ButtonCompareClick(Sender: TObject);
```

```
    procedure ButtonSyncClick(Sender: TObject);
  private
    procedure Log(const AMessage: string);
    procedure FetchRemoteData;
    procedure LoadLocalData;
    procedure CompareAndReport;
    procedure SyncToLocal;
  end;

var
  FormSync: TFormSync;

implementation

{$R *.fmx}

procedure TFormSync.FormCreate(Sender: TObject);
begin
  // Setup SQLite connection
  FDConnection1.Params.Clear;
  FDConnection1.Params.Add('DriverID=SQLite');
  FDConnection1.Params.Add('Database=C:\MyApp\data\local.db');
  FDConnection1.Connected := True;

  // Ensure the products table exists
  FDConnection1.ExecSQL(
    'CREATE TABLE IF NOT EXISTS products (' +
    '  id TEXT PRIMARY KEY,' +
    '  name TEXT,' +
    '  price REAL,' +
    '  stock INTEGER,' +
    '  updated_at TEXT' +
    ')');

  Log('Database connected. Ready to sync.');
end;

procedure TFormSync.Log(const AMessage: string);
begin
  MemoLog.Lines.Add(FormatDateTime('hh:nn:ss', Now) + '  ' + AMessage);
end;

procedure TFormSync.FetchRemoteData;
var
  StatusCode: Integer;
  Response: TBytes;
  JSON: TJSONObject;
begin
  Log('Fetching remote data...');
  LabelStatus.Text := 'Fetching from API...';
  Application.ProcessMessages;

  Response := SendHttpRequest(StatusCode,
    'https://api.example.com', '/products', 'limit=1000');

  if StatusCode <> 200 then
  begin
    Log('API error: HTTP ' + StatusCode.ToString);
    Exit;
```

```
  end;

  var JSONStr := TEncoding.UTF8.GetString(Response);

  // Populate remote MemTable
  PopulateMemTableFromJSON(FDMemTableRemote, 'records', JSONStr);
  Log('Fetched ' + FDMemTableRemote.RecordCount.ToString + ' remote products');
end;

procedure TFormSync.LoadLocalData;
var
  Result: TJSONObject;
begin
  Log('Loading local data...');

  Result := GetJSONFromDB(FDConnection1, 'SELECT * FROM products');
  try
    if Assigned(Result) then
    begin
      PopulateMemTableFromJSON(FDMemTableLocal, 'records', Result.ToString);
      Log('Loaded ' + FDMemTableLocal.RecordCount.ToString + ' local products');
    end;
  finally
    Result.Free;
  end;
end;

procedure TFormSync.CompareAndReport;
var
  LocalCount, RemoteCount: Integer;
  NewCount, UpdatedCount: Integer;
begin
  LocalCount := FDMemTableLocal.RecordCount;
  RemoteCount := FDMemTableRemote.RecordCount;
  NewCount := 0;
  UpdatedCount := 0;

  // Check each remote record against local
  FDMemTableRemote.First;
  while not FDMemTableRemote.Eof do
  begin
    var RemoteID := FDMemTableRemote.FieldByName('id').AsString;

    if FDMemTableLocal.Locate('id', RemoteID) then
    begin
      // Check if updated
      var RemoteName := FDMemTableRemote.FieldByName('name').AsString;
      var LocalName := FDMemTableLocal.FieldByName('name').AsString;
      if RemoteName <> LocalName then
        Inc(UpdatedCount);
    end
    else
      Inc(NewCount);

    FDMemTableRemote.Next;
  end;

  Log('---');
  Log('Comparison Results:');
```

```
  Log('  Local records:  ' + LocalCount.ToString);
  Log('  Remote records: ' + RemoteCount.ToString);
  Log('  New records:    ' + NewCount.ToString);
  Log('  Changed records: ' + UpdatedCount.ToString);
  Log('---');
end;

procedure TFormSync.ButtonCompareClick(Sender: TObject);
begin
  FetchRemoteData;
  LoadLocalData;
  CompareAndReport;
  LabelStatus.Text := 'Comparison complete';
end;

procedure TFormSync.ButtonSyncClick(Sender: TObject);
begin
  FetchRemoteData;
  SyncToLocal;
  LoadLocalData;  // Reload to verify
  LabelStatus.Text := 'Sync complete';
end;

procedure TFormSync.SyncToLocal;
var
  StatusCode: Integer;
  Response: TBytes;
begin
  Log('Syncing remote data to local database...');
  LabelStatus.Text := 'Syncing...';
  Application.ProcessMessages;

  // Fetch fresh data
  Response := SendHttpRequest(StatusCode,
    'https://api.example.com', '/products', 'limit=1000');

  if StatusCode <> 200 then
  begin
    Log('Sync failed: HTTP ' + StatusCode.ToString);
    Exit;
  end;

  var JSONStr := TEncoding.UTF8.GetString(Response);

  // Upsert into the database table
  var Result := PopulateTableFromJSON(
    FDConnection1,
    'products',     // table name
    JSONStr,        // JSON data
    'records',      // JSON array key
    'id');          // primary key for upsert

  Log('Sync complete. Processed records.');
end;

end.
```

---

12. Exercise: CLI Wrapper

Build a command-line wrapper that executes shell commands from a Delphi form, captures and displays output, and supports common operations.

Requirements

- A text input for custom commands
- Quick-action buttons for common operations: Ping, Traceroute, Directory listing, IP configuration
- Output display with timestamps
- Cross-platform command detection (Windows vs macOS/Linux)
- Exit code display

Solution

```
unit CLIForm;

interface

uses
  System.SysUtils, System.Types, System.Classes,
  FMX.Types, FMX.Controls, FMX.Forms, FMX.StdCtrls, FMX.Edit,
  FMX.Memo, FMX.Layouts,
  Tina4Core;

type
  TFormCLI = class(TForm)
    EditCommand: TEdit;
    ButtonRun: TButton;
    ButtonPing: TButton;
    ButtonTrace: TButton;
    ButtonDir: TButton;
    ButtonIPConfig: TButton;
    ButtonClear: TButton;
    MemoOutput: TMemo;
    LabelExitCode: TLabel;
    LabelPlatform: TLabel;
    procedure FormCreate(Sender: TObject);
    procedure ButtonRunClick(Sender: TObject);
    procedure ButtonPingClick(Sender: TObject);
    procedure ButtonTraceClick(Sender: TObject);
    procedure ButtonDirClick(Sender: TObject);
    procedure ButtonIPConfigClick(Sender: TObject);
    procedure ButtonClearClick(Sender: TObject);
    procedure EditCommandKeyDown(Sender: TObject; var Key: Word;
      var KeyChar: Char; Shift: TShiftState);
  private
    procedure RunCommand(const ACommand: string);
    function IsWindows: Boolean;
  end;

var
  FormCLI: TFormCLI;

implementation

{$R *.fmx}

function TFormCLI.IsWindows: Boolean;
begin
  {$IFDEF MSWINDOWS}
  Result := True;
  {$ELSE}
  Result := False;
  {$ENDIF}
end;

procedure TFormCLI.FormCreate(Sender: TObject);
begin
  if IsWindows then
    LabelPlatform.Text := 'Platform: Windows'
  else
```

```
    LabelPlatform.Text := 'Platform: macOS / Linux';
end;

procedure TFormCLI.RunCommand(const ACommand: string);
var
  Output: String;
  ExitCode: Integer;
begin
  MemoOutput.Lines.Add('');
  MemoOutput.Lines.Add('=== ' + FormatDateTime('yyyy-mm-dd hh:nn:ss', Now) + ' ===');
  MemoOutput.Lines.Add('$ ' + ACommand);
  MemoOutput.Lines.Add('');

  LabelExitCode.Text := 'Running...';
  Application.ProcessMessages;

  ExitCode := ExecuteShellCommand(ACommand, Output);

  MemoOutput.Lines.Add(Output);
  MemoOutput.Lines.Add('');

  if ExitCode = 0 then
    LabelExitCode.Text := 'Exit code: 0 (success)'
  else
    LabelExitCode.Text := 'Exit code: ' + ExitCode.ToString + ' (error)';

  // Scroll to bottom
  MemoOutput.GoToTextEnd;
end;

procedure TFormCLI.ButtonRunClick(Sender: TObject);
begin
  if EditCommand.Text.Trim.IsEmpty then
  begin
    ShowMessage('Enter a command to run');
    Exit;
  end;

  RunCommand(EditCommand.Text.Trim);
end;

procedure TFormCLI.EditCommandKeyDown(Sender: TObject; var Key: Word;
  var KeyChar: Char; Shift: TShiftState);
begin
  if Key = vkReturn then
    ButtonRunClick(Sender);
end;

procedure TFormCLI.ButtonPingClick(Sender: TObject);
begin
  if IsWindows then
    RunCommand('ping -n 4 google.com')
  else
    RunCommand('ping -c 4 google.com');
end;

procedure TFormCLI.ButtonTraceClick(Sender: TObject);
begin
  if IsWindows then
```

```
    RunCommand('tracert google.com')
  else
    RunCommand('traceroute google.com');
end;

procedure TFormCLI.ButtonDirClick(Sender: TObject);
begin
  if IsWindows then
    RunCommand('dir')
  else
    RunCommand('ls -la');
end;

procedure TFormCLI.ButtonIPConfigClick(Sender: TObject);
begin
  if IsWindows then
    RunCommand('ipconfig')
  else
    RunCommand('ifconfig');
end;

procedure TFormCLI.ButtonClearClick(Sender: TObject);
begin
  MemoOutput.Lines.Clear;
  LabelExitCode.Text := '';
end;

end.
```

Common Gotchas

TJSONObject memory leaks. Every `TJSONObject`, `TJSONArray`, or `TJSONValue` you create owns its children. When you call `Free` on the parent, all children are freed too. But if you create one and forget to free it, you leak memory. The pattern is always `try/finally`:

```
var Obj := StrToJSONObject(SomeString);
if Assigned(Obj) then
try
  // Use Obj...
finally
  Obj.Free;
end;
```

Do NOT free children of a `TJSONObject` separately -- the parent owns them:

```
// Wrong -- double free
var Obj := StrToJSONObject('{"items": [1,2,3]}');
var Arr := Obj.GetValue<TJSONArray>('items');
Arr.Free;   // CRASH! Obj owns Arr
Obj.Free;

// Right
var Obj := StrToJSONObject('{"items": [1,2,3]}');
try
  var Arr := Obj.GetValue<TJSONArray>('items');
  // Use Arr... don't free it
```

```
finally
  Obj.Free;  // Frees everything
end;
```

TBytes vs String conversion. HTTP responses come back as `TBytes`. To get a string:

```
var Text := TEncoding.UTF8.GetString(Response);
```

To go the other way:

```
var Bytes := TEncoding.UTF8.GetBytes(MyString);
```

Never assume ASCII. Always use `TEncoding.UTF8`.

Cross-platform shell commands. `ExecuteShellCommand` works differently on Windows and macOS/Linux. Windows uses `cmd.exe`, macOS/Linux uses `/bin/sh`. Always use conditional compilation for platform-specific commands:

```
{$IFDEF MSWINDOWS}
ExitCode := ExecuteShellCommand('dir /b C:\temp', Output);
{$ELSE}
ExitCode := ExecuteShellCommand('ls /tmp', Output);
{$ENDIF}
```

GetJSONFromDB field name conversion. The automatic `snake_case` to `camelCase` conversion is helpful, but it means your JSON keys will not match your database column names. If you need exact column names in your JSON, use `GetJSONFromTable` which does not convert names by default.

PopulateMemTableFromJSON field matching. When using Sync mode, the `IndexFieldNames` must be set on the MemTable before calling `PopulateMemTableFromJSON`. If the field names do not match (camelCase vs snake_case), sync will insert duplicates instead of updating:

```
// Set the index field BEFORE populating
FDMemTable1.IndexFieldNames := 'id';

// Now sync works correctly
PopulateMemTableFromJSON(FDMemTable1, 'records', JSONData,
  'id', TTina4RestSyncMode.Sync);
```

SendHttpRequest timeout. The default timeout is system-dependent. For slow APIs or large uploads, you may get a timeout before the request completes. Pass a timeout parameter when available, or use `TTina4RESTRequest` with async execution for long-running requests.

Building a CRUD Application

A Contact Manager From Scratch

No theory in this chapter. No isolated snippets. We are building a complete, working contact management application that uses every Tina4 Delphi component you have learned so far.

By the end, you will have:

- A SQLite database storing contacts
- A REST API backend (described, so you know what to build with Tina4 Python/PHP/Node.js/Ruby)
- A contact list displayed in a StringGrid
- A detail view rendered with HTML and Twig templates
- Create, update, and delete operations via HTML forms
- Search and filtering
- Loading states and error handling

Every line of code in this chapter is part of the final application. Read it top to bottom and you will have a working app.

---

1. What We Are Building

A contact management application with these features:

Feature	Components Used
List all contacts	TTina4RESTRequest, FDMemTable, StringGrid
View contact detail	TTina4HTMLRender, TTina4Twig
Create new contact	TTina4HTMLRender (form), OnFormSubmit, TTina4REST.Post
Edit existing contact	TTina4HTMLRender (form), OnFormSubmit, TTina4REST.Patch
Delete contact	TTina4REST.Delete with confirmation
Search/filter	MemTable filtering or REST query params
Status messages	TTina4HTMLRender for toast-style feedback

---

2. The REST API

Before building the Delphi client, here is the API it consumes. Build this with Tina4 Python, PHP, Node.js, or Ruby -- any backend that returns this JSON shape works.

Endpoints

Method	Endpoint	Description
GET	`/api/contacts`	List all contacts
GET	`/api/contacts?search=query`	Search contacts
GET	`/api/contacts/{id}`	Get single contact
POST	`/api/contacts`	Create contact
PATCH	`/api/contacts/{id}`	Update contact
DELETE	`/api/contacts/{id}`	Delete contact

Response Format

List response:

```
{
  "records": [
    {
      "id": "1",
      "firstName": "Andre",
      "lastName": "Van Zuydam",
      "email": "andre@example.com",
      "phone": "+27 21 555 0100",
      "company": "Tina4 Stack",
      "notes": "Framework creator",
      "createdAt": "2026-01-15T10:30:00.000Z"
    }
  ],
  "total": 42
}
```

Single record response:

```
{
  "id": "1",
  "firstName": "Andre",
  "lastName": "Van Zuydam",
  "email": "andre@example.com",
  "phone": "+27 21 555 0100",
  "company": "Tina4 Stack",
  "notes": "Framework creator",
  "createdAt": "2026-01-15T10:30:00.000Z"
}
```

Error response:

```
{
```

```
  "error": "Contact not found",
  "statusCode": 404
}
```

---

3. Project Setup

New FMX Project

Create a new **Multi-Device Application** in Delphi. Save the project as `ContactManager.dpr` with the main form unit as `MainForm.pas`.

Form Components

Drop these components on the form:

Data components (non-visual):

Component	Name	Purpose
TTina4REST	Tina4REST1	Base REST configuration
TTina4RESTRequest	RESTRequestList	Fetch contact list
TTina4RESTRequest	RESTRequestDetail	Fetch single contact
TFDMemTable	MemTableContacts	Contact list data
TDataSource	DataSourceContacts	Binds MemTable to grid

Visual components:

Component	Name	Purpose
TLayout	LayoutMain	Main container
TLayout	LayoutLeft	Left panel (list)
TLayout	LayoutRight	Right panel (detail/form)
TSplitter	Splitter1	Resizable divider
TStringGrid	GridContacts	Contact list
TEdit	EditSearch	Search input
TButton	ButtonSearch	Search button
TButton	ButtonNew	New contact button
TButton	ButtonRefresh	Refresh list button
TTina4HTMLRender	HTMLRenderDetail	Contact detail / form display
TLabel	LabelStatus	Status bar

Layout Structure

```
Form
  LayoutMain (Align=Client)
    LayoutLeft (Align=Left, Width=400)
      EditSearch (Align=Top)
      ButtonSearch (Align=Top)
      ButtonNew (Align=Top)
      ButtonRefresh (Align=Top)
      GridContacts (Align=Client)
    Splitter1 (Align=Left)
    LayoutRight (Align=Client)
      HTMLRenderDetail (Align=Client)
    LabelStatus (Align=Bottom)
```

---

4. REST Configuration

```
procedure TFormMain.FormCreate(Sender: TObject);
begin
  // Configure REST client
  Tina4REST1.BaseUrl := 'https://api.example.com/v1';
  // Tina4REST1.SetBearer('your-token-here');  // If auth required

  // Configure list request
  RESTRequestList.Tina4REST := Tina4REST1;
  RESTRequestList.EndPoint := '/api/contacts';
  RESTRequestList.RequestType := TTina4RequestType.Get;
  RESTRequestList.DataKey := 'records';
  RESTRequestList.MemTable := MemTableContacts;
  RESTRequestList.SyncMode := TTina4RestSyncMode.Clear;

  // Configure detail request
  RESTRequestDetail.Tina4REST := Tina4REST1;
  RESTRequestDetail.RequestType := TTina4RequestType.Get;
  RESTRequestDetail.DataKey := '';

  // Configure grid columns
  SetupGrid;

  // Configure HTML renderer
  HTMLRenderDetail.TwigTemplatePath := ExtractFilePath(ParamStr(0)) + 'templates';
  HTMLRenderDetail.OnFormSubmit := HandleFormSubmit;
  HTMLRenderDetail.OnElementClick := HandleElementClick;
  HTMLRenderDetail.RegisterObject('App', Self);

  // Load contacts
  RefreshContacts;
end;
```

---

5. Grid Setup

```
procedure TFormMain.SetupGrid;
begin
  GridContacts.ColumnCount := 4;

  GridContacts.Columns[0].Header := 'Name';
  GridContacts.Columns[0].Width := 150;

  GridContacts.Columns[1].Header := 'Email';
  GridContacts.Columns[1].Width := 150;

  GridContacts.Columns[2].Header := 'Phone';
  GridContacts.Columns[2].Width := 100;

  GridContacts.Columns[3].Header := 'Company';
  GridContacts.Columns[3].Width := 100;

  GridContacts.OnCellClick := GridCellClick;
end;
```

---

6. List Contacts

Fetching and Displaying

```
procedure TFormMain.RefreshContacts;
begin
  SetStatus('Loading contacts...');

  RESTRequestList.OnExecuteDone := procedure(Sender: TObject)
  begin
    TThread.Synchronize(nil, procedure
    begin
      PopulateGrid;
      SetStatus(Format('Loaded %d contacts', [MemTableContacts.RecordCount]));
    end);
  end;

  RESTRequestList.ExecuteRESTCallAsync;
end;

procedure TFormMain.PopulateGrid;
begin
  GridContacts.RowCount := MemTableContacts.RecordCount;

  MemTableContacts.First;
  var Row := 0;

  while not MemTableContacts.Eof do
  begin
    var FirstName := MemTableContacts.FieldByName('first_name').AsString;
    var LastName := MemTableContacts.FieldByName('last_name').AsString;

    GridContacts.Cells[0, Row] := FirstName + ' ' + LastName;
    GridContacts.Cells[1, Row] := MemTableContacts.FieldByName('email').AsString;
    GridContacts.Cells[2, Row] := MemTableContacts.FieldByName('phone').AsString;
    GridContacts.Cells[3, Row] := MemTableContacts.FieldByName('company').AsString;

    MemTableContacts.Next;
    Inc(Row);
  end;
end;

procedure TFormMain.ButtonRefreshClick(Sender: TObject);
begin
  RefreshContacts;
end;
```

---

7. View Contact Detail

When the user clicks a row in the grid, show the contact detail in the HTML renderer.

Grid Click Handler

```
procedure TFormMain.GridCellClick(const Column: TColumn; const Row: Integer);
begin
  // Navigate to the selected record in MemTable
  MemTableContacts.First;
  MemTableContacts.MoveBy(Row);

  var ContactID := MemTableContacts.FieldByName('id').AsString;
  ShowContactDetail(ContactID);
end;
```

Contact Detail Template (templates/contact-detail.html)

```
<div style="font-family: Arial, sans-serif; padding: 20px;">
  <div style="display: flex; justify-content: space-between; align-items: center; margin-bottom:
    <h2 style="color: #2c3e50; margin: 0;">{{ firstName }} {{ lastName }}</h2>
    <div>
      <button onclick="App:EditContact('{{ id }}')"
              style="background: #3498db; color: white; border: none; padding: 8px 16px; border-
        Edit
      </button>
      <button onclick="App:DeleteContact('{{ id }}')"
              style="background: #e74c3c; color: white; border: none; padding: 8px 16px; border-
        Delete
      </button>
    </div>
  </div>

  <div style="background: #f9f9f9; border-radius: 8px; padding: 20px;">
    <table style="width: 100%;">
      <tr>
        <td style="padding: 8px; color: #666; width: 120px;"><strong>Email</strong></td>
        <td style="padding: 8px;">
          {% if email %}
            <a href="mailto:{{ email }}">{{ email }}</a>
          {% else %}
            <span style="color: #ccc;">Not set</span>
          {% endif %}
        </td>
      </tr>
      <tr>
        <td style="padding: 8px; color: #666;"><strong>Phone</strong></td>
        <td style="padding: 8px;">{{ phone|default('Not set') }}</td>
      </tr>
      <tr>
        <td style="padding: 8px; color: #666;"><strong>Company</strong></td>
        <td style="padding: 8px;">{{ company|default('Not set') }}</td>
      </tr>
      <tr>
        <td style="padding: 8px; color: #666;"><strong>Notes</strong></td>
        <td style="padding: 8px;">{{ notes|default('No notes')|nl2br }}</td>
      </tr>
      <tr>
        <td style="padding: 8px; color: #666;"><strong>Created</strong></td>
        <td style="padding: 8px;">{{ createdAt|date('F j, Y') }}</td>
      </tr>
    </table>
  </div>
</div>
```

Show Detail

```
procedure TFormMain.ShowContactDetail(const AContactID: string);
var
  StatusCode: Integer;
  Response: TJSONObject;
begin
  SetStatus('Loading contact...');

  Response := Tina4REST1.Get(StatusCode, '/api/contacts/' + AContactID);
  try
    if StatusCode <> 200 then
    begin
      SetStatus('Error loading contact: HTTP ' + StatusCode.ToString);
      Exit;
    end;

    if not Assigned(Response) then
    begin
      SetStatus('Invalid response');
      Exit;
    end;

    // Set Twig variables from the JSON response
    HTMLRenderDetail.SetTwigVariable('id', Response.GetValue<String>('id', ''));
    HTMLRenderDetail.SetTwigVariable('firstName', Response.GetValue<String>('firstName', ''));
    HTMLRenderDetail.SetTwigVariable('lastName', Response.GetValue<String>('lastName', ''));
    HTMLRenderDetail.SetTwigVariable('email', Response.GetValue<String>('email', ''));
    HTMLRenderDetail.SetTwigVariable('phone', Response.GetValue<String>('phone', ''));
    HTMLRenderDetail.SetTwigVariable('company', Response.GetValue<String>('company', ''));
    HTMLRenderDetail.SetTwigVariable('notes', Response.GetValue<String>('notes', ''));
    HTMLRenderDetail.SetTwigVariable('createdAt', Response.GetValue<String>('createdAt', ''));

    // Render the template
    HTMLRenderDetail.Twig.LoadFromFile(
      ExtractFilePath(ParamStr(0)) + 'templates\contact-detail.html');

    SetStatus('Viewing: ' + Response.GetValue<String>('firstName', '') + ' ' +
      Response.GetValue<String>('lastName', ''));
  finally
    Response.Free;
  end;
end;
```

---

8. Create Contact

Contact Form Template (templates/contact-form.html)

```
<div style="font-family: Arial, sans-serif; padding: 20px;">
  <h2 style="color: #2c3e50;">
    {% if id %}Edit Contact{% else %}New Contact{% endif %}
  </h2>

  <form name="contactForm">
    {% if id %}
      <input type="hidden" name="id" value="{{ id }}">
    {% endif %}

    <div style="margin-bottom: 15px;">
      <label style="display: block; color: #666; margin-bottom: 4px;">First Name *</label>
      <input type="text" name="firstName" value="{{ firstName|default('') }}"
             class="form-control" placeholder="Enter first name"
             style="width: 100%; padding: 8px; border: 1px solid #ddd; border-radius: 4px;">
    </div>

    <div style="margin-bottom: 15px;">
      <label style="display: block; color: #666; margin-bottom: 4px;">Last Name *</label>
      <input type="text" name="lastName" value="{{ lastName|default('') }}"
             class="form-control" placeholder="Enter last name"
             style="width: 100%; padding: 8px; border: 1px solid #ddd; border-radius: 4px;">
    </div>

    <div style="margin-bottom: 15px;">
      <label style="display: block; color: #666; margin-bottom: 4px;">Email</label>
      <input type="email" name="email" value="{{ email|default('') }}"
             class="form-control" placeholder="name@example.com"
             style="width: 100%; padding: 8px; border: 1px solid #ddd; border-radius: 4px;">
    </div>

    <div style="margin-bottom: 15px;">
      <label style="display: block; color: #666; margin-bottom: 4px;">Phone</label>
      <input type="text" name="phone" value="{{ phone|default('') }}"
             class="form-control" placeholder="+27 21 555 0100"
             style="width: 100%; padding: 8px; border: 1px solid #ddd; border-radius: 4px;">
    </div>

    <div style="margin-bottom: 15px;">
      <label style="display: block; color: #666; margin-bottom: 4px;">Company</label>
      <input type="text" name="company" value="{{ company|default('') }}"
             class="form-control" placeholder="Company name"
             style="width: 100%; padding: 8px; border: 1px solid #ddd; border-radius: 4px;">
    </div>

    <div style="margin-bottom: 20px;">
      <label style="display: block; color: #666; margin-bottom: 4px;">Notes</label>
      <textarea name="notes" rows="4" class="form-control"
                placeholder="Additional notes..."
                style="width: 100%; padding: 8px; border: 1px solid #ddd; border-radius: 4px;">{
    </div>

    <div style="display: flex; gap: 10px;">
      <button type="submit" class="btn btn-primary"
              style="background: #1abc9c; color: white; border: none; padding: 10px 24px; border
```

```
        {% if id %}Save Changes{% else %}Create Contact{% endif %}
      </button>
      <button type="button" onclick="App:CancelForm()"
              style="background: #95a5a6; color: white; border: none; padding: 10px 24px; border
        Cancel
      </button>
    </div>
  </form>
</div>
```

Show Create Form

```
procedure TFormMain.ButtonNewClick(Sender: TObject);
begin
  ShowContactForm('', '', '', '', '', '', '');
end;

procedure TFormMain.ShowContactForm(const AID, AFirstName, ALastName,
  AEmail, APhone, ACompany, ANotes: string);
begin
  HTMLRenderDetail.SetTwigVariable('id', AID);
  HTMLRenderDetail.SetTwigVariable('firstName', AFirstName);
  HTMLRenderDetail.SetTwigVariable('lastName', ALastName);
  HTMLRenderDetail.SetTwigVariable('email', AEmail);
  HTMLRenderDetail.SetTwigVariable('phone', APhone);
  HTMLRenderDetail.SetTwigVariable('company', ACompany);
  HTMLRenderDetail.SetTwigVariable('notes', ANotes);

  HTMLRenderDetail.Twig.LoadFromFile(
    ExtractFilePath(ParamStr(0)) + 'templates\contact-form.html');

  if AID.IsEmpty then
    SetStatus('Creating new contact...')
  else
    SetStatus('Editing contact...');
end;
```

Handle Form Submission

```
procedure TFormMain.HandleFormSubmit(Sender: TObject;
  const FormName: string; FormData: TStrings);
var
  StatusCode: Integer;
  Response: TJSONObject;
  ContactID: string;
  Body: TJSONObject;
begin
  if FormName <> 'contactForm' then Exit;

  // Validate required fields
  var FirstName := FormData.Values['firstName'].Trim;
  var LastName := FormData.Values['lastName'].Trim;

  if FirstName.IsEmpty or LastName.IsEmpty then
  begin
    ShowMessage('First name and last name are required');
    Exit;
  end;

  ContactID := FormData.Values['id'];

  // Build the request body
  Body := TJSONObject.Create;
  try
    Body.AddPair('firstName', FirstName);
    Body.AddPair('lastName', LastName);
    Body.AddPair('email', FormData.Values['email'].Trim);
    Body.AddPair('phone', FormData.Values['phone'].Trim);
    Body.AddPair('company', FormData.Values['company'].Trim);
    Body.AddPair('notes', FormData.Values['notes'].Trim);

    if ContactID.IsEmpty then
    begin
      // CREATE
      SetStatus('Creating contact...');
      Response := Tina4REST1.Post(StatusCode, '/api/contacts', '', Body.ToString);
    end
    else
    begin
      // UPDATE
      SetStatus('Updating contact...');
      Response := Tina4REST1.Patch(StatusCode, '/api/contacts/' + ContactID, '', Body.ToString);
    end;
  finally
    Body.Free;
  end;

  try
    if StatusCode in [200, 201] then
    begin
      if ContactID.IsEmpty then
        SetStatus('Contact created successfully')
      else
        SetStatus('Contact updated successfully');

      // Refresh the list and show the detail
```

```
      RefreshContacts;

      if Assigned(Response) then
      begin
        var NewID := Response.GetValue<String>('id', ContactID);
        ShowContactDetail(NewID);
      end;
    end
    else
    begin
      var ErrorMsg := 'Unknown error';
      if Assigned(Response) then
        ErrorMsg := Response.GetValue<String>('error', ErrorMsg);
      SetStatus('Error: ' + ErrorMsg);
      ShowMessage('Failed to save contact: ' + ErrorMsg);
    end;
  finally
    Response.Free;
  end;
end;
```

---

9. Update Contact

Edit Button Handler

The Edit button in the detail template uses RTTI to call `EditContact`:

```
procedure TFormMain.EditContact(const AID: string);
var
  StatusCode: Integer;
  Response: TJSONObject;
begin
  // Fetch the current contact data
  Response := Tina4REST1.Get(StatusCode, '/api/contacts/' + AID);
  try
    if (StatusCode = 200) and Assigned(Response) then
    begin
      ShowContactForm(
        Response.GetValue<String>('id', ''),
        Response.GetValue<String>('firstName', ''),
        Response.GetValue<String>('lastName', ''),
        Response.GetValue<String>('email', ''),
        Response.GetValue<String>('phone', ''),
        Response.GetValue<String>('company', ''),
        Response.GetValue<String>('notes', ''));
    end
    else
      ShowMessage('Could not load contact for editing');
  finally
    Response.Free;
  end;
end;
```

The `HandleFormSubmit` method already handles both create and update -- it checks whether `id` is present in the form data.

---

10. Delete Contact

Delete with Confirmation

```
procedure TFormMain.DeleteContact(const AID: string);
begin
  // Confirm before deleting
  MessageDlg('Are you sure you want to delete this contact?',
    TMsgDlgType.mtConfirmation, [TMsgDlgBtn.mbYes, TMsgDlgBtn.mbNo], 0,
    procedure(const AResult: TModalResult)
    var
      StatusCode: Integer;
      Response: TJSONObject;
    begin
      if AResult <> mrYes then Exit;

      SetStatus('Deleting contact...');

      Response := Tina4REST1.Delete(StatusCode, '/api/contacts/' + AID);
      try
        if StatusCode in [200, 204] then
        begin
          SetStatus('Contact deleted');
          RefreshContacts;
          ShowEmptyState;
        end
        else
        begin
          var ErrorMsg := 'Delete failed';
          if Assigned(Response) then
            ErrorMsg := Response.GetValue<String>('error', ErrorMsg);
          SetStatus('Error: ' + ErrorMsg);
          ShowMessage(ErrorMsg);
        end;
      finally
        Response.Free;
      end;
    end);
end;

procedure TFormMain.ShowEmptyState;
begin
  HTMLRenderDetail.HTML.Text :=
    '<div style="font-family: Arial, sans-serif; padding: 40px; text-align: center; color: #999;
    '  <h2>No Contact Selected</h2>' +
    '  <p>Select a contact from the list or create a new one.</p>' +
    '</div>';
end;
```

---

11. Search and Filter

Option 1: Filter MemTable Locally

For small datasets, filter the existing MemTable without making another API call:

```
procedure TFormMain.ButtonSearchClick(Sender: TObject);
var
  SearchTerm: string;
begin
  SearchTerm := EditSearch.Text.Trim.ToLower;

  if SearchTerm.IsEmpty then
  begin
    MemTableContacts.Filtered := False;
    PopulateGrid;
    SetStatus(Format('Showing all %d contacts', [MemTableContacts.RecordCount]));
    Exit;
  end;

  MemTableContacts.OnFilterRecord := procedure(DataSet: TDataSet;
    var Accept: Boolean)
  begin
    var FirstName := DataSet.FieldByName('first_name').AsString.ToLower;
    var LastName := DataSet.FieldByName('last_name').AsString.ToLower;
    var Email := DataSet.FieldByName('email').AsString.ToLower;
    var Company := DataSet.FieldByName('company').AsString.ToLower;

    Accept := FirstName.Contains(SearchTerm) or
              LastName.Contains(SearchTerm) or
              Email.Contains(SearchTerm) or
              Company.Contains(SearchTerm);
  end;

  MemTableContacts.Filtered := True;
  PopulateGrid;
  SetStatus(Format('Found %d matching contacts', [MemTableContacts.RecordCount]));
end;
```

Option 2: Search via API

For large datasets, send the search term to the API:

```
procedure TFormMain.SearchViaAPI(const ASearchTerm: string);
begin
  if ASearchTerm.Trim.IsEmpty then
    RESTRequestList.EndPoint := '/api/contacts'
  else
    RESTRequestList.EndPoint := '/api/contacts?search=' + ASearchTerm.Trim;

  RESTRequestList.OnExecuteDone := procedure(Sender: TObject)
  begin
    TThread.Synchronize(nil, procedure
    begin
      PopulateGrid;
      SetStatus(Format('Found %d contacts for "%s"',
        [MemTableContacts.RecordCount, ASearchTerm]));
    end);
  end;

  RESTRequestList.ExecuteRESTCallAsync;
end;
```

Clear Search

```
procedure TFormMain.EditSearchKeyDown(Sender: TObject; var Key: Word;
  var KeyChar: Char; Shift: TShiftState);
begin
  if Key = vkReturn then
    ButtonSearchClick(Sender)
  else if Key = vkEscape then
  begin
    EditSearch.Text := '';
    MemTableContacts.Filtered := False;
    PopulateGrid;
  end;
end;
```

---

12. Polish: Status, Loading, and Error Handling

Status Bar

```
procedure TFormMain.SetStatus(const AMessage: string);
begin
  LabelStatus.Text := FormatDateTime('hh:nn:ss', Now) + '  ' + AMessage;
end;
```

Cancel Form Navigation

```
procedure TFormMain.CancelForm;
begin
  // If a contact was selected before, show its detail again
  if not MemTableContacts.IsEmpty then
  begin
    var ContactID := MemTableContacts.FieldByName('id').AsString;
    ShowContactDetail(ContactID);
  end
  else
    ShowEmptyState;
end;
```

Error Handling Wrapper

```
procedure TFormMain.SafeAPICall(AProc: TProc);
begin
  try
    AProc();
  except
    on E: Exception do
    begin
      SetStatus('Error: ' + E.Message);
      ShowMessage('An error occurred: ' + E.Message);
    end;
  end;
end;

// Usage:
procedure TFormMain.ButtonRefreshClick(Sender: TObject);
begin
  SafeAPICall(procedure
  begin
    RefreshContacts;
  end);
end;
```

---

13. Full Source Code

Here is the complete main form unit:

```
unit MainForm;

interface

uses
  System.SysUtils, System.Types, System.Classes, System.JSON,
  FMX.Types, FMX.Controls, FMX.Forms, FMX.StdCtrls, FMX.Edit,
  FMX.Layouts, FMX.Grid, FMX.Grid.Style, FMX.ScrollBox,
  FMX.Dialogs,
  FireDAC.Comp.Client, FireDAC.Stan.Intf,
  Data.DB,
  Tina4REST, Tina4RESTRequest, Tina4HTMLRender, Tina4Core;

type
  TFormMain = class(TForm)
    LayoutMain: TLayout;
    LayoutLeft: TLayout;
    LayoutRight: TLayout;
    Splitter1: TSplitter;
    GridContacts: TStringGrid;
    EditSearch: TEdit;
    ButtonSearch: TButton;
    ButtonNew: TButton;
    ButtonRefresh: TButton;
    HTMLRenderDetail: TTina4HTMLRender;
    LabelStatus: TLabel;
    Tina4REST1: TTina4REST;
    RESTRequestList: TTina4RESTRequest;
    RESTRequestDetail: TTina4RESTRequest;
```

```
    MemTableContacts: TFDMemTable;
    DataSourceContacts: TDataSource;

    procedure FormCreate(Sender: TObject);
    procedure ButtonRefreshClick(Sender: TObject);
    procedure ButtonNewClick(Sender: TObject);
    procedure ButtonSearchClick(Sender: TObject);
    procedure EditSearchKeyDown(Sender: TObject; var Key: Word;
      var KeyChar: Char; Shift: TShiftState);
  private
    procedure SetupGrid;
    procedure RefreshContacts;
    procedure PopulateGrid;
    procedure SetStatus(const AMessage: string);
    procedure ShowEmptyState;
    procedure ShowContactDetail(const AContactID: string);
    procedure ShowContactForm(const AID, AFirstName, ALastName,
      AEmail, APhone, ACompany, ANotes: string);
    procedure HandleFormSubmit(Sender: TObject;
      const FormName: string; FormData: TStrings);
    procedure HandleElementClick(Sender: TObject;
      const ObjectName, MethodName: string; Params: TStrings);
    procedure GridCellClick(const Column: TColumn; const Row: Integer);
  public
    procedure EditContact(const AID: string);
    procedure DeleteContact(const AID: string);
    procedure CancelForm;
  end;

var
  FormMain: TFormMain;

implementation

{$R *.fmx}

procedure TFormMain.FormCreate(Sender: TObject);
begin
  // REST configuration
  Tina4REST1.BaseUrl := 'https://api.example.com/v1';

  RESTRequestList.Tina4REST := Tina4REST1;
  RESTRequestList.EndPoint := '/api/contacts';
  RESTRequestList.RequestType := TTina4RequestType.Get;
  RESTRequestList.DataKey := 'records';
  RESTRequestList.MemTable := MemTableContacts;
  RESTRequestList.SyncMode := TTina4RestSyncMode.Clear;

  RESTRequestDetail.Tina4REST := Tina4REST1;
  RESTRequestDetail.RequestType := TTina4RequestType.Get;

  // Grid
  SetupGrid;

  // HTML Renderer
  HTMLRenderDetail.TwigTemplatePath := ExtractFilePath(ParamStr(0)) + 'templates';
  HTMLRenderDetail.OnFormSubmit := HandleFormSubmit;
  HTMLRenderDetail.RegisterObject('App', Self);
```

```
  // Initial state
  ShowEmptyState;
  RefreshContacts;
end;

procedure TFormMain.SetupGrid;
begin
  GridContacts.ColumnCount := 4;
  GridContacts.Columns[0].Header := 'Name';
  GridContacts.Columns[0].Width := 150;
  GridContacts.Columns[1].Header := 'Email';
  GridContacts.Columns[1].Width := 130;
  GridContacts.Columns[2].Header := 'Phone';
  GridContacts.Columns[2].Width := 100;
  GridContacts.Columns[3].Header := 'Company';
  GridContacts.Columns[3].Width := 100;
  GridContacts.OnCellClick := GridCellClick;
end;

procedure TFormMain.RefreshContacts;
begin
  SetStatus('Loading contacts...');

  RESTRequestList.OnExecuteDone := procedure(Sender: TObject)
  begin
    TThread.Synchronize(nil, procedure
    begin
      PopulateGrid;
      SetStatus(Format('Loaded %d contacts', [MemTableContacts.RecordCount]));
    end);
  end;

  RESTRequestList.ExecuteRESTCallAsync;
end;

procedure TFormMain.PopulateGrid;
begin
  GridContacts.RowCount := MemTableContacts.RecordCount;
  MemTableContacts.First;
  var Row := 0;

  while not MemTableContacts.Eof do
  begin
    GridContacts.Cells[0, Row] :=
      MemTableContacts.FieldByName('first_name').AsString + ' ' +
      MemTableContacts.FieldByName('last_name').AsString;
    GridContacts.Cells[1, Row] := MemTableContacts.FieldByName('email').AsString;
    GridContacts.Cells[2, Row] := MemTableContacts.FieldByName('phone').AsString;
    GridContacts.Cells[3, Row] := MemTableContacts.FieldByName('company').AsString;

    MemTableContacts.Next;
    Inc(Row);
  end;
end;

procedure TFormMain.GridCellClick(const Column: TColumn; const Row: Integer);
begin
  MemTableContacts.First;
  MemTableContacts.MoveBy(Row);
```

```
  var ContactID := MemTableContacts.FieldByName('id').AsString;
  ShowContactDetail(ContactID);
end;

procedure TFormMain.ShowContactDetail(const AContactID: string);
var
  StatusCode: Integer;
  Response: TJSONObject;
begin
  SetStatus('Loading contact...');
  Response := Tina4REST1.Get(StatusCode, '/api/contacts/' + AContactID);
  try
    if (StatusCode = 200) and Assigned(Response) then
    begin
      HTMLRenderDetail.SetTwigVariable('id', Response.GetValue<String>('id', ''));
      HTMLRenderDetail.SetTwigVariable('firstName', Response.GetValue<String>('firstName', ''));
      HTMLRenderDetail.SetTwigVariable('lastName', Response.GetValue<String>('lastName', ''));
      HTMLRenderDetail.SetTwigVariable('email', Response.GetValue<String>('email', ''));
      HTMLRenderDetail.SetTwigVariable('phone', Response.GetValue<String>('phone', ''));
      HTMLRenderDetail.SetTwigVariable('company', Response.GetValue<String>('company', ''));
      HTMLRenderDetail.SetTwigVariable('notes', Response.GetValue<String>('notes', ''));
      HTMLRenderDetail.SetTwigVariable('createdAt', Response.GetValue<String>('createdAt', ''));

      HTMLRenderDetail.Twig.LoadFromFile(
        ExtractFilePath(ParamStr(0)) + 'templates\contact-detail.html');

      SetStatus('Viewing: ' + Response.GetValue<String>('firstName', '') + ' ' +
        Response.GetValue<String>('lastName', ''));
    end
    else
      SetStatus('Error loading contact');
  finally
    Response.Free;
  end;
end;

procedure TFormMain.ShowContactForm(const AID, AFirstName, ALastName,
  AEmail, APhone, ACompany, ANotes: string);
begin
  HTMLRenderDetail.SetTwigVariable('id', AID);
  HTMLRenderDetail.SetTwigVariable('firstName', AFirstName);
  HTMLRenderDetail.SetTwigVariable('lastName', ALastName);
  HTMLRenderDetail.SetTwigVariable('email', AEmail);
  HTMLRenderDetail.SetTwigVariable('phone', APhone);
  HTMLRenderDetail.SetTwigVariable('company', ACompany);
  HTMLRenderDetail.SetTwigVariable('notes', ANotes);

  HTMLRenderDetail.Twig.LoadFromFile(
    ExtractFilePath(ParamStr(0)) + 'templates\contact-form.html');

  if AID.IsEmpty then
    SetStatus('Creating new contact...')
  else
    SetStatus('Editing contact...');
end;

procedure TFormMain.ButtonNewClick(Sender: TObject);
begin
  ShowContactForm('', '', '', '', '', '', '', '');
```

```
end;

procedure TFormMain.HandleFormSubmit(Sender: TObject;
  const FormName: string; FormData: TStrings);
var
  StatusCode: Integer;
  Response: TJSONObject;
  ContactID: string;
  Body: TJSONObject;
begin
  if FormName <> 'contactForm' then Exit;

  var FirstName := FormData.Values['firstName'].Trim;
  var LastName := FormData.Values['lastName'].Trim;

  if FirstName.IsEmpty or LastName.IsEmpty then
  begin
    ShowMessage('First name and last name are required');
    Exit;
  end;

  ContactID := FormData.Values['id'];
  Body := TJSONObject.Create;
  try
    Body.AddPair('firstName', FirstName);
    Body.AddPair('lastName', LastName);
    Body.AddPair('email', FormData.Values['email'].Trim);
    Body.AddPair('phone', FormData.Values['phone'].Trim);
    Body.AddPair('company', FormData.Values['company'].Trim);
    Body.AddPair('notes', FormData.Values['notes'].Trim);

    if ContactID.IsEmpty then
    begin
      SetStatus('Creating contact...');
      Response := Tina4REST1.Post(StatusCode, '/api/contacts', '', Body.ToString);
    end
    else
    begin
      SetStatus('Saving changes...');
      Response := Tina4REST1.Patch(StatusCode, '/api/contacts/' + ContactID, '', Body.ToString);
    end;
  finally
    Body.Free;
  end;

  try
    if StatusCode in [200, 201] then
    begin
      SetStatus(IfThen(ContactID.IsEmpty, 'Contact created', 'Contact updated'));
      RefreshContacts;
      if Assigned(Response) then
        ShowContactDetail(Response.GetValue<String>('id', ContactID));
    end
    else
    begin
      var ErrorMsg := 'Save failed';
      if Assigned(Response) then
        ErrorMsg := Response.GetValue<String>('error', ErrorMsg);
      SetStatus('Error: ' + ErrorMsg);
```

```
      ShowMessage(ErrorMsg);
    end;
  finally
    Response.Free;
  end;
end;

procedure TFormMain.HandleElementClick(Sender: TObject;
  const ObjectName, MethodName: string; Params: TStrings);
begin
  // RTTI-based onclick routing is handled automatically
  // This handler is available for custom processing if needed
end;

procedure TFormMain.EditContact(const AID: string);
var
  StatusCode: Integer;
  Response: TJSONObject;
begin
  Response := Tina4REST1.Get(StatusCode, '/api/contacts/' + AID);
  try
    if (StatusCode = 200) and Assigned(Response) then
      ShowContactForm(
        Response.GetValue<String>('id', ''),
        Response.GetValue<String>('firstName', ''),
        Response.GetValue<String>('lastName', ''),
        Response.GetValue<String>('email', ''),
        Response.GetValue<String>('phone', ''),
        Response.GetValue<String>('company', ''),
        Response.GetValue<String>('notes', ''))
    else
      ShowMessage('Could not load contact');
  finally
    Response.Free;
  end;
end;

procedure TFormMain.DeleteContact(const AID: string);
begin
  MessageDlg('Delete this contact?', TMsgDlgType.mtConfirmation,
    [TMsgDlgBtn.mbYes, TMsgDlgBtn.mbNo], 0,
    procedure(const AResult: TModalResult)
    var
      StatusCode: Integer;
      Response: TJSONObject;
    begin
      if AResult <> mrYes then Exit;

      SetStatus('Deleting...');
      Response := Tina4REST1.Delete(StatusCode, '/api/contacts/' + AID);
      try
        if StatusCode in [200, 204] then
        begin
          SetStatus('Contact deleted');
          RefreshContacts;
          ShowEmptyState;
        end
        else
          SetStatus('Delete failed');
```

```
      finally
        Response.Free;
      end;
    end);
end;

procedure TFormMain.CancelForm;
begin
  if not MemTableContacts.IsEmpty then
    ShowContactDetail(MemTableContacts.FieldByName('id').AsString)
  else
    ShowEmptyState;
end;

procedure TFormMain.ButtonSearchClick(Sender: TObject);
var
  SearchTerm: string;
begin
  SearchTerm := EditSearch.Text.Trim.ToLower;

  if SearchTerm.IsEmpty then
  begin
    MemTableContacts.Filtered := False;
    PopulateGrid;
    SetStatus(Format('Showing all %d contacts', [MemTableContacts.RecordCount]));
    Exit;
  end;

  MemTableContacts.OnFilterRecord := procedure(DataSet: TDataSet; var Accept: Boolean)
  begin
    Accept :=
      DataSet.FieldByName('first_name').AsString.ToLower.Contains(SearchTerm) or
      DataSet.FieldByName('last_name').AsString.ToLower.Contains(SearchTerm) or
      DataSet.FieldByName('email').AsString.ToLower.Contains(SearchTerm) or
      DataSet.FieldByName('company').AsString.ToLower.Contains(SearchTerm);
  end;

  MemTableContacts.Filtered := True;
  PopulateGrid;
  SetStatus(Format('Found %d contacts for "%s"', [MemTableContacts.RecordCount, SearchTerm]));
end;

procedure TFormMain.EditSearchKeyDown(Sender: TObject; var Key: Word;
  var KeyChar: Char; Shift: TShiftState);
begin
  if Key = vkReturn then
    ButtonSearchClick(Sender)
  else if Key = vkEscape then
  begin
    EditSearch.Text := '';
    MemTableContacts.Filtered := False;
    PopulateGrid;
  end;
end;

procedure TFormMain.ShowEmptyState;
begin
  HTMLRenderDetail.HTML.Text :=
    '<div style="font-family: Arial, sans-serif; padding: 40px; text-align: center; color: #999;
```

```
    '  <h2>No Contact Selected</h2>' +
    '  <p>Select a contact from the list, or click "New" to create one.</p>' +
    '</div>';
end;

procedure TFormMain.SetStatus(const AMessage: string);
begin
  LabelStatus.Text := FormatDateTime('hh:nn:ss', Now) + '  ' + AMessage;
end;

end.
```

---

14. Template Files

Create a `templates` folder next to your executable. Place `contact-detail.html` and `contact-form.html` there (shown in sections 7 and 8 above).

```
ContactManager.exe
templates/
  contact-detail.html
  contact-form.html
```

Exercise: Extend the Contact Manager

Add these features to the contact manager:

- **Categories/Tags** -- Add a `category` field to contacts (e.g., "Work", "Personal", "Client"). Add a dropdown filter above the grid to filter by category. Add a category select to the form.
- **Export to CSV** -- Add an "Export" button that saves the current contact list (filtered or unfiltered) to a CSV file using `TSaveDialog`.

Solution: CSV Export

```
procedure TFormMain.ButtonExportClick(Sender: TObject);
var
  SaveDialog: TSaveDialog;
  CSV: TStringList;
begin
  SaveDialog := TSaveDialog.Create(Self);
  CSV := TStringList.Create;
  try
    SaveDialog.Filter := 'CSV files|*.csv';
    SaveDialog.DefaultExt := 'csv';
    SaveDialog.FileName := 'contacts_' + FormatDateTime('yyyymmdd', Now) + '.csv';

    if not SaveDialog.Execute then Exit;

    // Header row
    CSV.Add('"First Name","Last Name","Email","Phone","Company","Notes"');

    // Data rows
    MemTableContacts.First;
    while not MemTableContacts.Eof do
    begin
      CSV.Add(Format('"%s","%s","%s","%s","%s","%s"', [
        MemTableContacts.FieldByName('first_name').AsString.Replace('"', '""'),
        MemTableContacts.FieldByName('last_name').AsString.Replace('"', '""'),
        MemTableContacts.FieldByName('email').AsString.Replace('"', '""'),
        MemTableContacts.FieldByName('phone').AsString.Replace('"', '""'),
        MemTableContacts.FieldByName('company').AsString.Replace('"', '""'),
        MemTableContacts.FieldByName('notes').AsString.Replace('"', '""')
      ]));

      MemTableContacts.Next;
    end;

    CSV.SaveToFile(SaveDialog.FileName, TEncoding.UTF8);
    SetStatus(Format('Exported %d contacts to %s',
      [MemTableContacts.RecordCount, ExtractFileName(SaveDialog.FileName)]));
  finally
    CSV.Free;
    SaveDialog.Free;
  end;
end;
```

Solution: Category Filter

Add a `TComboBox` named `ComboCategory` above the grid:

```
procedure TFormMain.FormCreate(Sender: TObject);
begin
  // ... existing setup ...

  ComboCategory.Items.Add('All Categories');
  ComboCategory.Items.Add('Work');
  ComboCategory.Items.Add('Personal');
  ComboCategory.Items.Add('Client');
  ComboCategory.Items.Add('Vendor');
  ComboCategory.ItemIndex := 0;
  ComboCategory.OnChange := ComboCategoryChange;
end;
```

```
procedure TFormMain.ComboCategoryChange(Sender: TObject);
begin
  if ComboCategory.ItemIndex = 0 then
  begin
    MemTableContacts.Filtered := False;
    PopulateGrid;
    SetStatus(Format('Showing all %d contacts', [MemTableContacts.RecordCount]));
  end
  else
  begin
    var Category := ComboCategory.Items[ComboCategory.ItemIndex];

    MemTableContacts.OnFilterRecord := procedure(DataSet: TDataSet; var Accept: Boolean)
    begin
      Accept := DataSet.FieldByName('category').AsString = Category;
    end;

    MemTableContacts.Filtered := True;
    PopulateGrid;
    SetStatus(Format('Showing %d "%s" contacts',
      [MemTableContacts.RecordCount, Category]));
  end;
end;
```

Add the category field to `contact-form.html`:

```
<div style="margin-bottom: 15px;">
  <label style="display: block; color: #666; margin-bottom: 4px;">Category</label>
  <select name="category" class="form-control"
          style="width: 100%; padding: 8px; border: 1px solid #ddd; border-radius: 4px;">
    <option value="">-- Select --</option>
    <option value="Work" {% if category == 'Work' %}selected{% endif %}>Work</option>
    <option value="Personal" {% if category == 'Personal' %}selected{% endif %}>Personal</option
    <option value="Client" {% if category == 'Client' %}selected{% endif %}>Client</option>
    <option value="Vendor" {% if category == 'Vendor' %}selected{% endif %}>Vendor</option>
  </select>
</div>
```

Real-World Integration

Making the Components Talk

You have learned each Tina4 Delphi component in isolation. TTina4REST makes HTTP calls. TTina4RESTRequest populates MemTables. TTina4JSONAdapter fans out JSON into multiple tables. TTina4HTMLRender displays HTML. TTina4HTMLPages navigates between views. TTina4Twig renders templates. TTina4WebSocketClient receives real-time data.

Individually, they are useful. Together, they are a full-stack desktop application framework. This chapter shows you the patterns for connecting them -- the repeatable architectures that turn seven components into a cohesive application.

---

1. Philosophy: Components as a Pipeline

Think of Tina4 Delphi as a data pipeline:

```
Data Source -> Transform -> Display
```

Every pattern in this chapter follows this flow. The data source is usually a REST API or WebSocket feed. The transform converts JSON into structured Delphi data (MemTables, objects). The display renders that data as grids, HTML, or template-driven UI.

The components snap together at defined connection points:

- **TTina4REST** feeds **TTina4RESTRequest**
- **TTina4RESTRequest** feeds **TFDMemTable**
- **TFDMemTable** feeds **TStringGrid** or **TTina4JSONAdapter**
- **TTina4RESTRequest** feeds **TTina4JSONAdapter** (via MasterSource)
- **TTina4HTMLRender** displays data from any source via Twig variables
- **TTina4HTMLPages** organizes multiple views on a single renderer
- **TTina4WebSocketClient** pushes data into any of the above

---

2. Pattern 1: REST to MemTable to Grid

The most common pattern. Fetch data from an API, populate a MemTable, display in a grid.

```
REST API -> TTina4RESTRequest -> TFDMemTable -> TStringGrid
```

Implementation

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  // Configure the pipeline
  Tina4REST1.BaseUrl := 'https://api.example.com/v1';

  RESTRequestProducts.Tina4REST := Tina4REST1;
  RESTRequestProducts.EndPoint := '/products';
  RESTRequestProducts.RequestType := TTina4RequestType.Get;
  RESTRequestProducts.DataKey := 'records';
  RESTRequestProducts.MemTable := FDMemTableProducts;
  RESTRequestProducts.SyncMode := TTina4RestSyncMode.Clear;

  // One call does everything
  RESTRequestProducts.ExecuteRESTCall;
  // FDMemTableProducts is now populated
  // Bind it to a grid with DataSourceProducts
end;
```

With Periodic Refresh

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  // ... setup as above ...

  // Refresh every 30 seconds
  Timer1.Interval := 30000;
  Timer1.OnTimer := procedure(Sender: TObject)
  begin
    RESTRequestProducts.SyncMode := TTina4RestSyncMode.Sync;
    RESTRequestProducts.ExecuteRESTCallAsync;
  end;
  Timer1.Enabled := True;
end;
```

Using `Sync` mode on refresh preserves the user's scroll position and selection, since existing records are updated in place rather than cleared and reloaded.

---

3. Pattern 2: REST to JSON Adapter to Multiple MemTables

When a single API response contains multiple datasets, use TTina4JSONAdapter to fan them out.

```
REST API -> TTina4RESTRequest -> TTina4JSONAdapter (categories)
                              -> TTina4JSONAdapter (tags)
                              -> TTina4JSONAdapter (stats)
```

Scenario

Your API returns a dashboard payload:

```
{
  "categories": [{"id": "1", "name": "Electronics"}, ...],
  "recentOrders": [{"id": "101", "total": "59.99"}, ...],
  "stats": {"totalProducts": 150, "totalOrders": 42, "revenue": 12500}
}
```

Implementation

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  Tina4REST1.BaseUrl := 'https://api.example.com/v1';

  // Main request fetches the dashboard payload
  RESTRequestDashboard.Tina4REST := Tina4REST1;
  RESTRequestDashboard.EndPoint := '/dashboard';
  RESTRequestDashboard.RequestType := TTina4RequestType.Get;
  RESTRequestDashboard.DataKey := '';  // We handle the keys in adapters
  RESTRequestDashboard.MemTable := FDMemTableRaw;

  // Adapter 1: Categories
  JSONAdapterCategories.MasterSource := RESTRequestDashboard;
  JSONAdapterCategories.DataKey := 'categories';
  JSONAdapterCategories.MemTable := FDMemTableCategories;

  // Adapter 2: Recent Orders
  JSONAdapterOrders.MasterSource := RESTRequestDashboard;
  JSONAdapterOrders.DataKey := 'recentOrders';
  JSONAdapterOrders.MemTable := FDMemTableOrders;

  // When the dashboard request completes, all adapters auto-execute
  RESTRequestDashboard.ExecuteRESTCall;

  // FDMemTableCategories and FDMemTableOrders are now populated
end;
```

The key insight: set `MasterSource` on each adapter to the same `TTina4RESTRequest`. When that request completes, all adapters fire automatically.

---

4. Pattern 3: HTML Render + Twig + REST Data

Rich UI rendering by combining API data with Twig templates.

```
REST API -> TJSONObject -> SetTwigVariable -> TTina4HTMLRender.Twig
```

Implementation

```
procedure TForm1.ShowProductCard(const AProductID: string);
var
  StatusCode: Integer;
  Response: TJSONObject;
begin
  Response := Tina4REST1.Get(StatusCode, '/products/' + AProductID);
  try
    if (StatusCode <> 200) or not Assigned(Response) then Exit;

    // Pass each field as a Twig variable
    HTMLRender1.SetTwigVariable('name', Response.GetValue<String>('name', ''));
    HTMLRender1.SetTwigVariable('price', Response.GetValue<String>('price', '0'));
    HTMLRender1.SetTwigVariable('description', Response.GetValue<String>('description', ''));
    HTMLRender1.SetTwigVariable('imageUrl', Response.GetValue<String>('imageUrl', ''));
    HTMLRender1.SetTwigVariable('stock', Response.GetValue<String>('stock', '0'));
    HTMLRender1.SetTwigVariable('category', Response.GetValue<String>('category', ''));

    // Load the template -- Twig renders, HTMLRender displays
    HTMLRender1.Twig.LoadFromFile('C:\MyApp\templates\product-card.html');
  finally
    Response.Free;
  end;
end;
```

The Template (product-card.html)

```
<div style="font-family: Arial, sans-serif; padding: 20px;">
  <div style="display: flex; gap: 20px;">
    {% if imageUrl %}
      <img src="{{ imageUrl }}" style="width: 200px; height: 200px; object-fit: cover; border-ra
    {% endif %}
    <div>
      <h2 style="color: #2c3e50; margin: 0 0 10px;">{{ name }}</h2>
      <p style="color: #1abc9c; font-size: 1.4em; font-weight: bold;">
        {{ price|format_currency('USD') }}
      </p>
      <p style="color: #666;">{{ category|title }}</p>
      <p>
        {% if stock|number_format > 10 %}
          <span style="color: #27ae60;">In Stock ({{ stock }})</span>
        {% elseif stock|number_format > 0 %}
          <span style="color: #f39c12;">Low Stock ({{ stock }} left)</span>
        {% else %}
          <span style="color: #e74c3c;">Out of Stock</span>
        {% endif %}
      </p>
    </div>
  </div>
  <div style="margin-top: 15px; padding: 15px; background: #f9f9f9; border-radius: 4px;">
    <p>{{ description|nl2br }}</p>
  </div>
</div>
```

---

5. Pattern 4: Master-Detail Chains

Linked data views where selecting a record in one view loads related data in another.

```
Customers -> Orders -> Order Items
(MemTable1)  (MemTable2)  (MemTable3)
```

Implementation

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  Tina4REST1.BaseUrl := 'https://api.example.com/v1';

  // Level 1: Customers
  RESTRequestCustomers.Tina4REST := Tina4REST1;
  RESTRequestCustomers.EndPoint := '/customers';
  RESTRequestCustomers.DataKey := 'records';
  RESTRequestCustomers.MemTable := FDMemTableCustomers;

  // Level 2: Orders for selected customer
  RESTRequestOrders.Tina4REST := Tina4REST1;
  RESTRequestOrders.MasterSource := RESTRequestCustomers;
  RESTRequestOrders.EndPoint := '/customers/{id}/orders';
  RESTRequestOrders.DataKey := 'records';
  RESTRequestOrders.MemTable := FDMemTableOrders;

  // Level 3: Items for selected order
  RESTRequestItems.Tina4REST := Tina4REST1;
  RESTRequestItems.MasterSource := RESTRequestOrders;
  RESTRequestItems.EndPoint := '/orders/{id}/items';
  RESTRequestItems.DataKey := 'records';
  RESTRequestItems.MemTable := FDMemTableItems;

  // Load customers -- orders and items auto-load on selection change
  RESTRequestCustomers.ExecuteRESTCall;
end;
```

The `{id}` placeholder in the endpoint is replaced with the `id` field from the master's current MemTable record. When the user selects a different customer, the orders request fires automatically with the new customer ID. When they select a different order, the items request fires with the new order ID.

Displaying the Chain

```
// Grid1 shows customers (bound to FDMemTableCustomers)
// Grid2 shows orders for selected customer (bound to FDMemTableOrders)
// HTMLRender1 shows order items in a formatted table

procedure TForm1.GridOrdersCellClick(const Column: TColumn; const Row: Integer);
begin
  FDMemTableOrders.First;
  FDMemTableOrders.MoveBy(Row);
  // RESTRequestItems fires automatically via MasterSource
  // Update the HTML display
  RenderOrderItems;
end;

procedure TForm1.RenderOrderItems;
var
  HTML: TStringBuilder;
begin
  HTML := TStringBuilder.Create;
  try
    HTML.AppendLine('<table style="width: 100%; border-collapse: collapse;">');
    HTML.AppendLine('<tr style="background: #2c3e50; color: white;">');
    HTML.AppendLine('<th style="padding: 8px;">Product</th>');
    HTML.AppendLine('<th style="padding: 8px; text-align: right;">Qty</th>');
    HTML.AppendLine('<th style="padding: 8px; text-align: right;">Price</th>');
    HTML.AppendLine('</tr>');

    FDMemTableItems.First;
    while not FDMemTableItems.Eof do
    begin
      HTML.AppendFormat(
        '<tr><td style="padding: 8px;">%s</td>' +
        '<td style="padding: 8px; text-align: right;">%s</td>' +
        '<td style="padding: 8px; text-align: right;">$%s</td></tr>',
        [FDMemTableItems.FieldByName('product_name').AsString,
         FDMemTableItems.FieldByName('quantity').AsString,
         FDMemTableItems.FieldByName('unit_price').AsString]);
      FDMemTableItems.Next;
    end;

    HTML.AppendLine('</table>');
    HTMLRender1.HTML.Text := HTML.ToString;
  finally
    HTML.Free;
  end;
end;
```

---

6. Pattern 5: Two-Way Sync

Read data from the API, let the user edit it locally, then push changes back.

```
GET /items -> MemTable (user edits) -> POST/PATCH /items
```

Implementation

```
procedure TForm1.LoadData;
begin
  RESTRequestItems.EndPoint := '/items';
  RESTRequestItems.RequestType := TTina4RequestType.Get;
  RESTRequestItems.DataKey := 'records';
  RESTRequestItems.MemTable := FDMemTableItems;
  RESTRequestItems.ExecuteRESTCall;
end;

procedure TForm1.SaveChanges;
var
  StatusCode: Integer;
  Response: TJSONObject;
begin
  // Iterate changed records and push them back
  FDMemTableItems.First;
  while not FDMemTableItems.Eof do
  begin
    var ItemID := FDMemTableItems.FieldByName('id').AsString;
    var ItemJSON := TJSONObject.Create;
    try
      // Build JSON from current MemTable row
      for var I := 0 to FDMemTableItems.FieldCount - 1 do
      begin
        var Field := FDMemTableItems.Fields[I];
        var FieldName := CamelCase(Field.FieldName);
        ItemJSON.AddPair(FieldName, Field.AsString);
      end;

      if ItemID.IsEmpty then
      begin
        // New record -- POST
        Response := Tina4REST1.Post(StatusCode, '/items', '', ItemJSON.ToString);
        Response.Free;
      end
      else
      begin
        // Existing record -- PATCH
        Response := Tina4REST1.Patch(StatusCode, '/items/' + ItemID, '', ItemJSON.ToString);
        Response.Free;
      end;
    finally
      ItemJSON.Free;
    end;

    FDMemTableItems.Next;
  end;
end;
```

Using SourceMemTable for Bulk POST

For simpler cases, use the `SourceMemTable` property to POST all rows at once:

```
procedure TForm1.BulkUpload;
begin
  RESTRequestImport.Tina4REST := Tina4REST1;
  RESTRequestImport.RequestType := TTina4RequestType.Post;
  RESTRequestImport.EndPoint := '/items/import';
```

```
  RESTRequestImport.SourceMemTable := FDMemTableItems;
  RESTRequestImport.SourceIgnoreFields := 'internal_flag,temp_id';
  RESTRequestImport.SourceIgnoreBlanks := True;
  RESTRequestImport.ExecuteRESTCall;
end;
```

---

7. Pattern 6: HTML Pages + REST

Each page in a TTina4HTMLPages collection loads its own data when navigated to.

```
HTMLPages
  Page "dashboard" -> OnAfterNavigate -> fetch /dashboard data
  Page "products"  -> OnAfterNavigate -> fetch /products data
  Page "settings"  -> OnAfterNavigate -> fetch /settings data
```

Implementation

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  Tina4REST1.BaseUrl := 'https://api.example.com/v1';
  HTMLPages1.Renderer := HTMLRender1;
  HTMLPages1.TwigTemplatePath := ExtractFilePath(ParamStr(0)) + 'templates';

  // Define pages
  var PageDash := HTMLPages1.Pages.Add;
  PageDash.PageName := 'dashboard';
  PageDash.IsDefault := True;
  PageDash.TwigContent.Text := '{% include "pages/dashboard.html" %}';

  var PageProducts := HTMLPages1.Pages.Add;
  PageProducts.PageName := 'products';
  PageProducts.TwigContent.Text := '{% include "pages/products.html" %}';

  var PageSettings := HTMLPages1.Pages.Add;
  PageSettings.PageName := 'settings';
  PageSettings.TwigContent.Text := '{% include "pages/settings.html" %}';

  // Load data when pages change
  HTMLPages1.OnAfterNavigate := HandlePageNavigate;
end;

procedure TForm1.HandlePageNavigate(Sender: TObject);
var
  StatusCode: Integer;
  Response: TJSONObject;
begin
  var PageName := HTMLPages1.ActivePage;

  if PageName = 'dashboard' then
  begin
    Response := Tina4REST1.Get(StatusCode, '/dashboard/stats');
    try
      if Assigned(Response) then
      begin
        HTMLPages1.SetTwigVariable('totalUsers',
          Response.GetValue<String>('totalUsers', '0'));
        HTMLPages1.SetTwigVariable('totalOrders',
          Response.GetValue<String>('totalOrders', '0'));
        HTMLPages1.SetTwigVariable('revenue',
          Response.GetValue<String>('revenue', '0'));
      end;
    finally
      Response.Free;
    end;
    // Re-render the current page with new data
    HTMLPages1.NavigateTo('dashboard');
  end
  else if PageName = 'products' then
  begin
    Response := Tina4REST1.Get(StatusCode, '/products', 'limit=50');
    try
      if Assigned(Response) then
        HTMLPages1.SetTwigVariable('products', Response.ToString);
    finally
```

```
      Response.Free;
    end;
    HTMLPages1.NavigateTo('products');
  end;
end;
```

Navigation Template

```
<!-- templates/nav.html -->
<nav style="background: #2c3e50; padding: 10px;">
  <a href="#dashboard" style="color: white; margin-right: 15px;">Dashboard</a>
  <a href="#products" style="color: white; margin-right: 15px;">Products</a>
  <a href="#settings" style="color: white;">Settings</a>
</nav>
```

---

8. Pattern 7: WebSocket + HTML Render

Real-time data pushed from a WebSocket updates the HTML display immediately.

```
WebSocket -> OnMessage -> Update Twig variables -> Re-render HTML
```

Implementation

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  WebSocket1.URL := 'wss://api.example.com/ws/updates';
  WebSocket1.AutoReconnect := True;
  WebSocket1.OnMessage := HandleWSUpdate;
  WebSocket1.Connect;

  // Initial render
  RenderStatusPanel;
end;

procedure TForm1.HandleWSUpdate(Sender: TObject; const AMessage: string);
begin
  TThread.Synchronize(nil, procedure
  var
    JSON: TJSONObject;
  begin
    JSON := StrToJSONObject(AMessage);
    if not Assigned(JSON) then Exit;
    try
      var UpdateType := JSON.GetValue<String>('type', '');

      if UpdateType = 'metric' then
      begin
        var MetricName := JSON.GetValue<String>('name', '');
        var MetricValue := JSON.GetValue<String>('value', '');

        // Update the specific Twig variable
        HTMLRender1.SetTwigVariable(MetricName, MetricValue);

        // Re-render the template with updated data
        HTMLRender1.Twig.LoadFromFile(
          ExtractFilePath(ParamStr(0)) + 'templates\status-panel.html');
      end
      else if UpdateType = 'alert' then
      begin
        var AlertMsg := JSON.GetValue<String>('message', '');
        ShowNotification(AlertMsg);
      end;
    finally
      JSON.Free;
    end;
  end);
end;

procedure TForm1.RenderStatusPanel;
begin
  // Set initial values
  HTMLRender1.SetTwigVariable('cpuUsage', '0');
  HTMLRender1.SetTwigVariable('memoryUsage', '0');
  HTMLRender1.SetTwigVariable('diskUsage', '0');
  HTMLRender1.SetTwigVariable('activeUsers', '0');
  HTMLRender1.SetTwigVariable('requestsPerSec', '0');

  HTMLRender1.Twig.LoadFromFile(
    ExtractFilePath(ParamStr(0)) + 'templates\status-panel.html');
end;
```

Status Panel Template

```
<div style="font-family: Arial, sans-serif; padding: 15px;">
  <h2 style="color: #2c3e50;">System Status</h2>

  <div style="display: flex; gap: 15px; flex-wrap: wrap;">
    {% macro metric(label, value, unit, color) %}
      <div style="flex: 1; min-width: 150px; background: white; padding: 15px;
                  border-radius: 8px; border-left: 4px solid {{ color }};">
        <p style="color: #999; margin: 0; font-size: 0.85em;">{{ label }}</p>
        <p style="color: {{ color }}; font-size: 1.8em; font-weight: bold; margin: 5px 0;">
          {{ value }}{{ unit }}
        </p>
      </div>
    {% endmacro %}

    {{ metric('CPU Usage', cpuUsage, '%', '#3498db') }}
    {{ metric('Memory', memoryUsage, '%', '#2ecc71') }}
    {{ metric('Disk', diskUsage, '%', '#e67e22') }}
    {{ metric('Active Users', activeUsers, '', '#9b59b6') }}
    {{ metric('Requests/s', requestsPerSec, '', '#1abc9c') }}
  </div>
</div>
```

---

9. Complete Example: Product Management Dashboard

This brings every pattern together into a single application.

Architecture

```
Left Sidebar:   HTMLPages navigation (categories from API)
Main Area:      StringGrid (products filtered by category)
Detail Panel:   HTMLRender + Twig (product card with image)
Edit Form:      HTMLRender (HTML form -> POST/PATCH)
Live Updates:   WebSocket (product changes from other users)
```

Form Components

Component	Name	Purpose
TTina4REST	REST1	Base configuration
TTina4RESTRequest	RESTCategories	Fetch categories
TTina4RESTRequest	RESTProducts	Fetch products by category
TTina4WebSocketClient	WebSocket1	Live product updates
TTina4HTMLPages	HTMLPagesNav	Sidebar navigation
TTina4HTMLRender	HTMLRenderNav	Sidebar renderer
TTina4HTMLRender	HTMLRenderDetail	Product detail / edit form
TFDMemTable	MemTableCategories	Categories data
TFDMemTable	MemTableProducts	Products data
TStringGrid	GridProducts	Product list

Implementation

```
unit DashboardForm;

interface

uses
  System.SysUtils, System.Types, System.Classes, System.JSON,
  FMX.Types, FMX.Controls, FMX.Forms, FMX.StdCtrls, FMX.Edit,
  FMX.Layouts, FMX.Grid, FMX.Grid.Style, FMX.ScrollBox,
  FMX.Dialogs,
  FireDAC.Comp.Client,
  Data.DB,
  Tina4REST, Tina4RESTRequest, Tina4HTMLRender, Tina4HTMLPages,
  Tina4WebSocket, Tina4Core;

type
  TFormDashboard = class(TForm)
    LayoutMain: TLayout;
    LayoutSidebar: TLayout;
    LayoutCenter: TLayout;
    LayoutDetail: TLayout;
    SplitterLeft: TSplitter;
    SplitterRight: TSplitter;

    REST1: TTina4REST;
    RESTCategories: TTina4RESTRequest;
    RESTProducts: TTina4RESTRequest;
    WebSocket1: TTina4WebSocketClient;

    HTMLRenderNav: TTina4HTMLRender;
    HTMLPagesNav: TTina4HTMLPages;
    HTMLRenderDetail: TTina4HTMLRender;

    MemTableCategories: TFDMemTable;
    MemTableProducts: TFDMemTable;
    GridProducts: TStringGrid;
    LabelStatus: TLabel;

    procedure FormCreate(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
  private
    FCurrentCategory: string;
    FCurrentProductID: string;

    procedure SetupREST;
    procedure SetupNavigation;
    procedure SetupGrid;
    procedure SetupWebSocket;

    procedure LoadCategories;
    procedure LoadProducts(const ACategoryID: string);
    procedure ShowProductDetail(const AProductID: string);
    procedure ShowEditForm(const AProductID: string);

    procedure OnWSMessage(Sender: TObject; const AMessage: string);
    procedure OnWSOpen(Sender: TObject);
    procedure GridCellClick(const Column: TColumn; const Row: Integer);
    procedure HandleFormSubmit(Sender: TObject;
```

```
      const FormName: string; FormData: TStrings);

    procedure SetStatus(const AMessage: string);
    procedure PopulateGrid;
  public
    procedure SelectCategory(const ACategoryID: string);
    procedure EditProduct(const AProductID: string);
    procedure DeleteProduct(const AProductID: string);
  end;

var
  FormDashboard: TFormDashboard;

implementation

{$R *.fmx}

procedure TFormDashboard.FormCreate(Sender: TObject);
begin
  SetupREST;
  SetupNavigation;
  SetupGrid;
  SetupWebSocket;

  LoadCategories;
  SetStatus('Ready');
end;

procedure TFormDashboard.FormDestroy(Sender: TObject);
begin
  WebSocket1.AutoReconnect := False;
  if WebSocket1.Connected then
    WebSocket1.Disconnect;
end;

// ---- REST Setup ----

procedure TFormDashboard.SetupREST;
begin
  REST1.BaseUrl := 'https://api.example.com/v1';
  // REST1.SetBearer('your-token');

  RESTCategories.Tina4REST := REST1;
  RESTCategories.EndPoint := '/categories';
  RESTCategories.RequestType := TTina4RequestType.Get;
  RESTCategories.DataKey := 'records';
  RESTCategories.MemTable := MemTableCategories;

  RESTProducts.Tina4REST := REST1;
  RESTProducts.RequestType := TTina4RequestType.Get;
  RESTProducts.DataKey := 'records';
  RESTProducts.MemTable := MemTableProducts;
end;

// ---- Sidebar Navigation ----

procedure TFormDashboard.SetupNavigation;
begin
  HTMLPagesNav.Renderer := HTMLRenderNav;
```

```
  HTMLPagesNav.TwigTemplatePath := ExtractFilePath(ParamStr(0)) + 'templates';
  HTMLRenderNav.RegisterObject('App', Self);
end;

procedure TFormDashboard.LoadCategories;
begin
  SetStatus('Loading categories...');

  RESTCategories.OnExecuteDone := procedure(Sender: TObject)
  begin
    TThread.Synchronize(nil, procedure
    var
      HTML: TStringBuilder;
    begin
      // Build the sidebar navigation from category data
      HTML := TStringBuilder.Create;
      try
        HTML.AppendLine('<div style="font-family: Arial, sans-serif; padding: 10px;">');
        HTML.AppendLine('<h3 style="color: #2c3e50; padding: 0 10px;">Categories</h3>');

        MemTableCategories.First;
        while not MemTableCategories.Eof do
        begin
          var CatID := MemTableCategories.FieldByName('id').AsString;
          var CatName := MemTableCategories.FieldByName('name').AsString;
          var IsActive := (CatID = FCurrentCategory);

          HTML.AppendFormat(
            '<div onclick="App:SelectCategory(''%s'')" ' +
            '     style="padding: 10px 15px; cursor: pointer; border-radius: 4px; ' +
            '     margin: 2px 5px; background: %s; color: %s;">' +
            '  %s' +
            '</div>',
            [CatID,
             IfThen(IsActive, '#1abc9c', 'transparent'),
             IfThen(IsActive, 'white', '#333'),
             CatName]);

          MemTableCategories.Next;
        end;

        HTML.AppendLine('</div>');
        HTMLRenderNav.HTML.Text := HTML.ToString;
      finally
        HTML.Free;
      end;

      // Auto-select first category
      if (FCurrentCategory.IsEmpty) and (MemTableCategories.RecordCount > 0) then
      begin
        MemTableCategories.First;
        SelectCategory(MemTableCategories.FieldByName('id').AsString);
      end;

      SetStatus(Format('Loaded %d categories', [MemTableCategories.RecordCount]));
    end);
  end;

  RESTCategories.ExecuteRESTCallAsync;
```

```
end;

procedure TFormDashboard.SelectCategory(const ACategoryID: string);
begin
  FCurrentCategory := ACategoryID;
  LoadProducts(ACategoryID);
  // Refresh sidebar to show active state
  LoadCategories;
end;

// ---- Product Grid ----

procedure TFormDashboard.SetupGrid;
begin
  GridProducts.ColumnCount := 4;
  GridProducts.Columns[0].Header := 'Product';
  GridProducts.Columns[0].Width := 180;
  GridProducts.Columns[1].Header := 'Price';
  GridProducts.Columns[1].Width := 80;
  GridProducts.Columns[2].Header := 'Stock';
  GridProducts.Columns[2].Width := 60;
  GridProducts.Columns[3].Header := 'Status';
  GridProducts.Columns[3].Width := 80;
  GridProducts.OnCellClick := GridCellClick;
end;

procedure TFormDashboard.LoadProducts(const ACategoryID: string);
begin
  SetStatus('Loading products...');

  RESTProducts.EndPoint := '/categories/' + ACategoryID + '/products';

  RESTProducts.OnExecuteDone := procedure(Sender: TObject)
  begin
    TThread.Synchronize(nil, procedure
    begin
      PopulateGrid;
      SetStatus(Format('Loaded %d products', [MemTableProducts.RecordCount]));
    end);
  end;

  RESTProducts.ExecuteRESTCallAsync;
end;

procedure TFormDashboard.PopulateGrid;
begin
  GridProducts.RowCount := MemTableProducts.RecordCount;
  MemTableProducts.First;
  var Row := 0;

  while not MemTableProducts.Eof do
  begin
    GridProducts.Cells[0, Row] := MemTableProducts.FieldByName('name').AsString;
    GridProducts.Cells[1, Row] := '$' + MemTableProducts.FieldByName('price').AsString;

    var Stock := MemTableProducts.FieldByName('stock').AsInteger;
    GridProducts.Cells[2, Row] := Stock.ToString;

    if Stock > 10 then
```

```
      GridProducts.Cells[3, Row] := 'In Stock'
    else if Stock > 0 then
      GridProducts.Cells[3, Row] := 'Low'
    else
      GridProducts.Cells[3, Row] := 'Out';

    MemTableProducts.Next;
    Inc(Row);
  end;
end;

procedure TFormDashboard.GridCellClick(const Column: TColumn; const Row: Integer);
begin
  MemTableProducts.First;
  MemTableProducts.MoveBy(Row);
  var ProductID := MemTableProducts.FieldByName('id').AsString;
  ShowProductDetail(ProductID);
end;

// ---- Product Detail ----

procedure TFormDashboard.ShowProductDetail(const AProductID: string);
var
  StatusCode: Integer;
  Response: TJSONObject;
begin
  FCurrentProductID := AProductID;
  SetStatus('Loading product...');

  Response := REST1.Get(StatusCode, '/products/' + AProductID);
  try
    if (StatusCode <> 200) or not Assigned(Response) then
    begin
      SetStatus('Error loading product');
      Exit;
    end;

    HTMLRenderDetail.SetTwigVariable('id', Response.GetValue<String>('id', ''));
    HTMLRenderDetail.SetTwigVariable('name', Response.GetValue<String>('name', ''));
    HTMLRenderDetail.SetTwigVariable('price', Response.GetValue<String>('price', '0'));
    HTMLRenderDetail.SetTwigVariable('description', Response.GetValue<String>('description', '')
    HTMLRenderDetail.SetTwigVariable('imageUrl', Response.GetValue<String>('imageUrl', ''));
    HTMLRenderDetail.SetTwigVariable('stock', Response.GetValue<String>('stock', '0'));
    HTMLRenderDetail.SetTwigVariable('category', Response.GetValue<String>('category', ''));
    HTMLRenderDetail.SetTwigVariable('sku', Response.GetValue<String>('sku', ''));

    HTMLRenderDetail.Twig.LoadFromFile(
      ExtractFilePath(ParamStr(0)) + 'templates\product-detail.html');

    SetStatus('Viewing: ' + Response.GetValue<String>('name', ''));
  finally
    Response.Free;
  end;
end;

// ---- Edit Form ----

procedure TFormDashboard.EditProduct(const AProductID: string);
begin
```

```
  ShowEditForm(AProductID);
end;

procedure TFormDashboard.ShowEditForm(const AProductID: string);
var
  StatusCode: Integer;
  Response: TJSONObject;
begin
  Response := REST1.Get(StatusCode, '/products/' + AProductID);
  try
    if (StatusCode = 200) and Assigned(Response) then
    begin
      HTMLRenderDetail.SetTwigVariable('id', Response.GetValue<String>('id', ''));
      HTMLRenderDetail.SetTwigVariable('name', Response.GetValue<String>('name', ''));
      HTMLRenderDetail.SetTwigVariable('price', Response.GetValue<String>('price', ''));
      HTMLRenderDetail.SetTwigVariable('description', Response.GetValue<String>('description', '
      HTMLRenderDetail.SetTwigVariable('stock', Response.GetValue<String>('stock', ''));
      HTMLRenderDetail.SetTwigVariable('sku', Response.GetValue<String>('sku', ''));

      HTMLRenderDetail.Twig.LoadFromFile(
        ExtractFilePath(ParamStr(0)) + 'templates\product-form.html');

      SetStatus('Editing: ' + Response.GetValue<String>('name', ''));
    end;
  finally
    Response.Free;
  end;
end;

procedure TFormDashboard.HandleFormSubmit(Sender: TObject;
  const FormName: string; FormData: TStrings);
var
  StatusCode: Integer;
  Response: TJSONObject;
  Body: TJSONObject;
begin
  if FormName <> 'productForm' then Exit;

  var ProductID := FormData.Values['id'];
  Body := TJSONObject.Create;
  try
    Body.AddPair('name', FormData.Values['name'].Trim);
    Body.AddPair('price', TJSONNumber.Create(StrToFloatDef(FormData.Values['price'], 0)));
    Body.AddPair('description', FormData.Values['description'].Trim);
    Body.AddPair('stock', TJSONNumber.Create(StrToIntDef(FormData.Values['stock'], 0)));
    Body.AddPair('sku', FormData.Values['sku'].Trim);

    if ProductID.IsEmpty then
      Response := REST1.Post(StatusCode, '/products', '', Body.ToString)
    else
      Response := REST1.Patch(StatusCode, '/products/' + ProductID, '', Body.ToString);
  finally
    Body.Free;
  end;

  try
    if StatusCode in [200, 201] then
    begin
      SetStatus('Product saved');
```

```
      LoadProducts(FCurrentCategory);
      if Assigned(Response) then
        ShowProductDetail(Response.GetValue<String>('id', ProductID));
    end
    else
    begin
      SetStatus('Save failed');
      ShowMessage('Could not save product');
    end;
  finally
    Response.Free;
  end;
end;

procedure TFormDashboard.DeleteProduct(const AProductID: string);
begin
  MessageDlg('Delete this product?', TMsgDlgType.mtConfirmation,
    [TMsgDlgBtn.mbYes, TMsgDlgBtn.mbNo], 0,
    procedure(const AResult: TModalResult)
    var
      StatusCode: Integer;
      Response: TJSONObject;
    begin
      if AResult <> mrYes then Exit;

      Response := REST1.Delete(StatusCode, '/products/' + AProductID);
      try
        if StatusCode in [200, 204] then
        begin
          SetStatus('Product deleted');
          LoadProducts(FCurrentCategory);
          HTMLRenderDetail.HTML.Text :=
            '<p style="padding: 40px; text-align: center; color: #999;">Select a product</p>';
        end;
      finally
        Response.Free;
      end;
    end);
end;

// ---- WebSocket Live Updates ----

procedure TFormDashboard.SetupWebSocket;
begin
  WebSocket1.URL := 'wss://api.example.com/ws/products';
  WebSocket1.AutoReconnect := True;
  WebSocket1.ReconnectInterval := 5000;
  WebSocket1.PingInterval := 30000;
  WebSocket1.OnOpen := OnWSOpen;
  WebSocket1.OnMessage := OnWSMessage;
  WebSocket1.Connect;
end;

procedure TFormDashboard.OnWSOpen(Sender: TObject);
begin
  TThread.Synchronize(nil, procedure
  begin
    // Subscribe to product updates
    var Sub := TJSONObject.Create;
```

```
    try
      Sub.AddPair('type', 'subscribe');
      Sub.AddPair('channel', 'products');
      WebSocket1.Send(Sub.ToString);
    finally
      Sub.Free;
    end;
  end);
end;

procedure TFormDashboard.OnWSMessage(Sender: TObject; const AMessage: string);
begin
  TThread.Synchronize(nil, procedure
  var
    JSON: TJSONObject;
  begin
    JSON := StrToJSONObject(AMessage);
    if not Assigned(JSON) then Exit;
    try
      var MsgType := JSON.GetValue<String>('type', '');

      if MsgType = 'product_updated' then
      begin
        var ProductID := JSON.GetValue<String>('productId', '');
        var CategoryID := JSON.GetValue<String>('categoryId', '');

        // Refresh the grid if we are viewing the affected category
        if CategoryID = FCurrentCategory then
        begin
          LoadProducts(FCurrentCategory);
          SetStatus('Product updated by another user');
        end;

        // Refresh detail if viewing the affected product
        if ProductID = FCurrentProductID then
          ShowProductDetail(ProductID);
      end
      else if MsgType = 'product_deleted' then
      begin
        var CategoryID := JSON.GetValue<String>('categoryId', '');
        if CategoryID = FCurrentCategory then
        begin
          LoadProducts(FCurrentCategory);
          SetStatus('Product deleted by another user');
        end;
      end;
    finally
      JSON.Free;
    end;
  end);
end;

// ---- Utilities ----

procedure TFormDashboard.SetStatus(const AMessage: string);
begin
  LabelStatus.Text := FormatDateTime('hh:nn:ss', Now) + '  ' + AMessage;
end;
```

```
end.
```

Template Files

templates/product-detail.html:

```
<div style="font-family: Arial, sans-serif; padding: 20px;">
  <div style="display: flex; justify-content: space-between; align-items: center;">
    <h2 style="color: #2c3e50; margin: 0;">{{ name }}</h2>
    <div>
      <button onclick="App:EditProduct('{{ id }}')"
              style="background: #3498db; color: white; border: none; padding: 8px 16px;
                     border-radius: 4px; margin-right: 5px; cursor: pointer;">Edit</button>
      <button onclick="App:DeleteProduct('{{ id }}')"
              style="background: #e74c3c; color: white; border: none; padding: 8px 16px;
                     border-radius: 4px; cursor: pointer;">Delete</button>
    </div>
  </div>

  <div style="display: flex; gap: 20px; margin-top: 20px;">
    {% if imageUrl %}
      <img src="{{ imageUrl }}" style="width: 250px; height: 250px; object-fit: cover; border-ra
    {% else %}
      <div style="width: 250px; height: 250px; background: #ecf0f1; border-radius: 8px;
                  display: flex; align-items: center; justify-content: center; color: #bdc3c7;">
        No Image
      </div>
    {% endif %}

    <div style="flex: 1;">
      <p style="font-size: 1.4em; color: #1abc9c; font-weight: bold;">
        {{ price|format_currency('USD') }}
      </p>
      <table style="width: 100%;">
        <tr>
          <td style="padding: 6px 0; color: #999; width: 80px;">SKU</td>
          <td style="padding: 6px 0;">{{ sku|upper }}</td>
        </tr>
        <tr>
          <td style="padding: 6px 0; color: #999;">Category</td>
          <td style="padding: 6px 0;">{{ category|title }}</td>
        </tr>
        <tr>
          <td style="padding: 6px 0; color: #999;">Stock</td>
          <td style="padding: 6px 0;">
            {% if stock > 10 %}
              <span style="color: #27ae60;">In Stock ({{ stock }})</span>
            {% elseif stock > 0 %}
              <span style="color: #f39c12;">Low Stock ({{ stock }})</span>
            {% else %}
              <span style="color: #e74c3c;">Out of Stock</span>
            {% endif %}
          </td>
        </tr>
      </table>
    </div>
  </div>

  {% if description %}
```

```
    <div style="margin-top: 20px; padding: 15px; background: #f9f9f9; border-radius: 4px;">
      <h3 style="margin: 0 0 10px; color: #2c3e50;">Description</h3>
      <p>{{ description|nl2br }}</p>
    </div>
  {% endif %}
</div>
```

templates/product-form.html:

```
<div style="font-family: Arial, sans-serif; padding: 20px;">
  <h2 style="color: #2c3e50;">{% if id %}Edit Product{% else %}New Product{% endif %}</h2>

  <form name="productForm">
    {% if id %}<input type="hidden" name="id" value="{{ id }}">{% endif %}

    <div style="margin-bottom: 12px;">
      <label style="display: block; color: #666; margin-bottom: 4px;">Name *</label>
      <input type="text" name="name" value="{{ name|default('') }}"
             style="width: 100%; padding: 8px; border: 1px solid #ddd; border-radius: 4px;">
    </div>

    <div style="display: flex; gap: 15px; margin-bottom: 12px;">
      <div style="flex: 1;">
        <label style="display: block; color: #666; margin-bottom: 4px;">Price *</label>
        <input type="text" name="price" value="{{ price|default('') }}"
               style="width: 100%; padding: 8px; border: 1px solid #ddd; border-radius: 4px;">
      </div>
      <div style="flex: 1;">
        <label style="display: block; color: #666; margin-bottom: 4px;">Stock</label>
        <input type="text" name="stock" value="{{ stock|default('0') }}"
               style="width: 100%; padding: 8px; border: 1px solid #ddd; border-radius: 4px;">
      </div>
      <div style="flex: 1;">
        <label style="display: block; color: #666; margin-bottom: 4px;">SKU</label>
        <input type="text" name="sku" value="{{ sku|default('') }}"
               style="width: 100%; padding: 8px; border: 1px solid #ddd; border-radius: 4px;">
      </div>
    </div>

    <div style="margin-bottom: 15px;">
      <label style="display: block; color: #666; margin-bottom: 4px;">Description</label>
      <textarea name="description" rows="5"
                style="width: 100%; padding: 8px; border: 1px solid #ddd; border-radius: 4px;">{
    </div>

    <div style="display: flex; gap: 10px;">
      <button type="submit"
              style="background: #1abc9c; color: white; border: none; padding: 10px 24px;
                     border-radius: 4px; cursor: pointer;">Save</button>
      <button type="button" onclick="App:ShowProductDetail('{{ id }}')"
              style="background: #95a5a6; color: white; border: none; padding: 10px 24px;
                     border-radius: 4px; cursor: pointer;">Cancel</button>
    </div>
  </form>
</div>
```

---

10. Integration with Tina4 Backend

The Delphi components are framework-agnostic -- they work with any REST API. But they are designed to pair naturally with Tina4 backends.

Tina4 Python Backend

```
from tina4_python import get, post, patch, delete

@get("/api/products")
async def get_products(request, response):
    products = DBI.fetch("SELECT * FROM products")
    return response(products)

@post("/api/products")
async def create_product(request, response):
    product = request.body
    DBI.insert("products", product)
    return response(product, 201)
```

Tina4 PHP Backend

```
\Tina4\Get::add("/api/products", function(\Tina4\Response $response) {
    return $response((new Product())->select("*")->asArray());
});

\Tina4\Post::add("/api/products", function(\Tina4\Response $response, \Tina4\Request $request) {
    $product = new Product($request->data);
    $product->save();
    return $response($product, 201);
});
```

Tina4 Node.js Backend

```
Router.get("/api/products", async (req, res) => {
    const products = await DBI.fetch("SELECT * FROM products");
    res.json(products);
});

Router.post("/api/products", async (req, res) => {
    const product = await DBI.insert("products", req.body);
    res.status(201).json(product);
});
```

The `GetJSONFromDB` output format matches what Delphi's `TTina4RESTRequest` expects. The `DataKey` of `"records"` works with the default array key. Field names are automatically converted between `snake_case` (database) and `camelCase` (JSON) on both sides.

Exercise: Monitoring Dashboard

Build a monitoring dashboard that combines all three data sources:

- **REST polling** -- Fetch server metrics every 30 seconds from `/api/metrics`
- **WebSocket alerts** -- Receive real-time alerts from `wss://api.example.com/ws/alerts`

- **HTML-rendered status panels** -- Display metrics as colored cards, alerts as a scrollable list

Requirements

- Four metric cards: CPU, Memory, Disk, Network (updated via REST polling)
- Alert list: show the last 20 alerts with timestamp, severity, and message
- Color coding: green (< 60%), yellow (60-80%), red (> 80%) for metric values
- Sound or visual flash when a critical alert arrives
- A "Clear Alerts" button

Solution Outline

```
procedure TFormMonitor.FormCreate(Sender: TObject);
begin
  // REST polling for metrics
  REST1.BaseUrl := 'https://api.example.com/v1';
  TimerMetrics.Interval := 30000;
  TimerMetrics.OnTimer := FetchMetrics;
  TimerMetrics.Enabled := True;

  // WebSocket for alerts
  WebSocket1.URL := 'wss://api.example.com/ws/alerts';
  WebSocket1.AutoReconnect := True;
  WebSocket1.OnMessage := HandleAlert;
  WebSocket1.Connect;

  // Initial fetch
  FetchMetrics(nil);
end;

procedure TFormMonitor.FetchMetrics(Sender: TObject);
var
  StatusCode: Integer;
  Response: TJSONObject;
begin
  Response := REST1.Get(StatusCode, '/api/metrics');
  try
    if (StatusCode = 200) and Assigned(Response) then
    begin
      HTMLRenderMetrics.SetTwigVariable('cpu',
        Response.GetValue<String>('cpu', '0'));
      HTMLRenderMetrics.SetTwigVariable('memory',
        Response.GetValue<String>('memory', '0'));
      HTMLRenderMetrics.SetTwigVariable('disk',
        Response.GetValue<String>('disk', '0'));
      HTMLRenderMetrics.SetTwigVariable('network',
        Response.GetValue<String>('network', '0'));

      HTMLRenderMetrics.Twig.LoadFromFile(
        ExtractFilePath(ParamStr(0)) + 'templates\metrics.html');
    end;
  finally
    Response.Free;
  end;
end;

procedure TFormMonitor.HandleAlert(Sender: TObject; const AMessage: string);
begin
  TThread.Synchronize(nil, procedure
  var
    JSON: TJSONObject;
  begin
    JSON := StrToJSONObject(AMessage);
    if not Assigned(JSON) then Exit;
    try
      var Severity := JSON.GetValue<String>('severity', 'info');
      var Message := JSON.GetValue<String>('message', '');
      var Timestamp := JSON.GetValue<String>('timestamp', GetJSONDate(Now));

      // Add to alert list (keep last 20)
```

```
      FAlerts.Insert(0, Format('%s|%s|%s', [Timestamp, Severity, Message]));
      while FAlerts.Count > 20 do
        FAlerts.Delete(FAlerts.Count - 1);

      RefreshAlertDisplay;

      // Visual flash for critical alerts
      if Severity = 'critical' then
      begin
        HTMLRenderAlerts.SetElementStyle('alertPanel', 'background-color', '#e74c3c');
        // Reset after 500ms
        TThread.CreateAnonymousThread(procedure
        begin
          Sleep(500);
          TThread.Synchronize(nil, procedure
          begin
            HTMLRenderAlerts.SetElementStyle('alertPanel', 'background-color', 'white');
          end);
        end).Start;
      end;
    finally
      JSON.Free;
    end;
  end);
end;
```

The metrics template uses the same macro pattern from Pattern 7:

```
{% macro metricCard(label, value, unit) %}
  {% set color = '#27ae60' %}
  {% if value > 80 %}{% set color = '#e74c3c' %}
  {% elseif value > 60 %}{% set color = '#f39c12' %}{% endif %}

  <div style="flex: 1; background: white; padding: 15px; border-radius: 8px;
              border-top: 4px solid {{ color }}; text-align: center;">
    <p style="color: #999; margin: 0;">{{ label }}</p>
    <p style="color: {{ color }}; font-size: 2em; font-weight: bold; margin: 5px 0;">
      {{ value }}{{ unit }}
    </p>
  </div>
{% endmacro %}

<div style="display: flex; gap: 15px; padding: 15px;">
  {{ metricCard('CPU', cpu, '%') }}
  {{ metricCard('Memory', memory, '%') }}
  {{ metricCard('Disk', disk, '%') }}
  {{ metricCard('Network', network, ' Mbps') }}
</div>
```

This exercise combines everything: REST for periodic data, WebSocket for real-time events, Twig for templated rendering, and HTML Render for interactive display. It is the distillation of every pattern in this chapter into a single, practical application.

Claude Code Integration

Let AI Write Your Delphi

You describe a form. Claude writes the Pascal. It compiles the project. It launches the app. It clicks a button to test the logic. It sees the result on screen. It fixes a bug in the event handler, recompiles, and tests again. You never left the terminal.

This is not theoretical. Tina4 Delphi ships with an MCP server that gives Claude Code direct access to the Free Pascal and Delphi compilers, the ability to generate proper project structures, and tools to interact with running desktop applications. This chapter shows you how to set it up and how to use it effectively.

---

1. What Is MCP

MCP stands for Model Context Protocol. It is an open standard that lets AI assistants like Claude Code interact with external tools. Instead of just reading and writing files, Claude can call tool functions -- compile code, run programs, take screenshots, click buttons.

Think of MCP as a plugin system. You install an MCP server. Claude discovers the tools it provides. When Claude needs to compile your Pascal code, it calls the `compile_pascal` tool. When it needs to see the running app, it calls the `preview_screenshot` tool. The protocol handles the communication.

Without MCP, Claude can write Delphi code. With MCP, Claude can write, compile, run, see, interact with, and debug Delphi code. The difference is the gap between writing a recipe and cooking the meal.

---

2. The Pascal MCP Server

The MCP server is in a separate repository at github.com/tina4stack/claude-pascal-mcp. It is a Python-based server that exposes Pascal/Delphi development tools to any MCP-compatible AI assistant (Claude Code, Cursor, Copilot, etc.).

What the Server Provides

Tool	What It Does
`compile_pascal`	Compile a single `.pas` file or a full Delphi project (DPR + PAS + DFM)
`project_template`	Generate a complete Delphi project structure with components and event handlers
`run_program`	Compile and execute a program, capture stdout and stderr
`launch_gui`	Compile and run a VCL/FMX application in the background
`preview_screenshot`	Capture what the running desktop app looks like via HTTP bridge
`click_button`	Enumerate child windows and send click messages to controls
`move_window`	Reposition and resize application windows
`type_text`	Enter text into the focused control
`send_keys`	Send keyboard shortcuts (Ctrl+S, Alt+F4, etc.)
`parse_form`	Read `.dfm`, `.fmx`, or `.lfm` files and return component structure
`detect_compilers`	Find Free Pascal, Delphi 7 (dcc32), and RAD Studio (dcc64) on the system

Installation

The server uses `uv`, a fast Python package manager. If you do not have `uv` installed:

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

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

Then install the MCP server dependencies:

```
cd /path/to/claude-pascal-mcp
uv sync
```

That is it. No virtual environment management, no pip install chains, no requirements.txt conflicts. `uv sync` reads the project file and installs everything.

---

3. Configuration

Project-Level Configuration

Create a `.mcp.json` file in your Delphi project root:

```
{
  "mcpServers": {
    "pascal-dev": {
      "command": "uv",
      "args": [
        "run",
        "--directory",
        "/path/to/claude-pascal-mcp",
        "pascal-mcp"
      ]
    }
  }
}
```

Replace `/path/to/tina4delphi` with the actual path where you cloned the tina4delphi repository.

Global Configuration

To make the MCP server available in all Claude Code sessions, add it to your global Claude settings:

```
claude mcp add pascal-dev -- uv run --directory /path/to/claude-pascal-mcp pascal-mcp
```

Verifying the Setup

Start Claude Code in your project directory and check that the tools are available:

```
cd /path/to/your/delphi-project
claude
```

Claude will show connected MCP servers on startup. You should see `pascal-dev` listed. Ask Claude to detect compilers:

```
What Pascal/Delphi compilers are available on this machine?
```

Claude calls `detect_compilers` and reports what it finds -- Free Pascal, Delphi 7, RAD Studio, or any combination.

---

4. Specifying a Compiler

The MCP server auto-detects compilers by scanning common installation paths. On Windows, it checks:

- `C:\FPC\*\bin\*\fpc.exe` for Free Pascal
- `C:\Program Files (x86)\Borland\Delphi7\Bin\dcc32.exe` for Delphi 7
- `C:\Program Files (x86)\Embarcadero\Studio\*\bin\dcc64.exe` for RAD Studio

If your compiler is installed in a non-standard location, tell Claude:

```
Compile my project using the compiler at D:\Tools\FPC\3.2.2\bin\x86_64-win64\fpc.exe
```

Claude passes the path directly to the `compile_pascal` tool. No configuration file needed.

---

5. Preview Bridge Setup

The preview bridge lets Claude see your running desktop application through its built-in preview panel. It works by capturing screenshots of the app window and serving them over HTTP.

Create `.claude/launch.json` in your project root:

```
{
  "version": "0.0.1",
  "configurations": [
    {
      "name": "pascal-preview",
      "runtimeExecutable": "/path/to/claude-pascal-mcp/.venv/Scripts/pythonw.exe",
      "runtimeArgs": ["-m", "pascal_mcp.preview_bridge"],
      "port": 18080,
      "autoPort": true
    }
  ]
}
```

On macOS or Linux, use the Python path from the virtual environment:

```
{
  "version": "0.0.1",
  "configurations": [
    {
      "name": "pascal-preview",
      "runtimeExecutable": "/path/to/claude-pascal-mcp/.venv/bin/python",
      "runtimeArgs": ["-m", "pascal_mcp.preview_bridge"],
      "port": 18080,
      "autoPort": true
    }
  ]
}
```

When Claude launches a GUI application, it starts the preview bridge automatically. The bridge captures screenshots of the running app and serves them at `http://localhost:18080`. Claude sees these screenshots in its preview panel and can make decisions based on what the UI looks like.

---

6. What Claude Can Do

Compile Single Files

Claude can compile a standalone Pascal program:

```
// hello.pas
program Hello;
begin
  WriteLn('Hello from Claude!');
```

```
end.
```

Ask Claude: "Compile and run hello.pas." Claude calls `compile_pascal` with the file path, then `run_program` to execute it. You see the output in the conversation.

Compile Full Delphi Projects

Claude understands the Delphi project structure. Given a `.dpr` file, it compiles the entire project with all units, forms, and resources:

```
// MyApp.dpr
program MyApp;

uses
  FMX.Forms,
  MainUnit in 'MainUnit.pas' {MainForm};

{$R *.res}

begin
  Application.Initialize;
  Application.CreateForm(TMainForm, MainForm);
  Application.Run;
end.
```

Claude resolves unit dependencies, includes form files, and passes the correct compiler flags.

Generate Project Templates

When you ask Claude to create a new application, it uses the `project_template` tool to generate the correct project structure:

```
Create a new FMX application with a form that has a TTina4HTMLRender and a TTina4REST component.
```

Claude generates:

```
MyNewApp/
  MyNewApp.dpr          -- Project file
  MyNewApp.dproj        -- IDE project file
  MainUnit.pas          -- Main form unit
  MainUnit.fmx          -- FMX form file
```

The generated `.fmx` form includes the components you requested, correctly parented and configured. The `.pas` unit includes the corresponding field declarations and event handler stubs.

Launch GUI Applications

Claude can compile and run GUI applications in the background:

```
// MainUnit.pas
unit MainUnit;

interface

uses
  System.SysUtils, System.Classes, FMX.Types, FMX.Controls,
  FMX.Forms, FMX.StdCtrls, FMX.Layouts;

type
  TMainForm = class(TForm)
```

```
    Button1: TButton;
    Label1: TLabel;
    procedure Button1Click(Sender: TObject);
  private
    FClickCount: Integer;
  end;

var
  MainForm: TMainForm;

implementation

{$R *.fmx}

procedure TMainForm.Button1Click(Sender: TObject);
begin
  Inc(FClickCount);
  Label1.Text := 'Clicked ' + FClickCount.ToString + ' times';
end;

end.
```

Claude compiles this, launches the app, and confirms it is running. With the preview bridge active, Claude can see the window.

Interact with Running Applications

This is where the MCP server earns its keep. Claude does not just launch your app -- it uses it:

```
Claude, click Button1 in the running app and tell me what happens.
```

Claude calls `click_button` to find and click the button. It takes a screenshot to see the result. It reads the label text. It reports back: "The label now shows 'Clicked 1 times'."

Claude can also type into text fields:

```
Type "admin@example.com" into the email input field.
```

And send keyboard shortcuts:

```
Press Ctrl+S to trigger the save action.
```

Parse Form Files

Claude can read and understand Delphi form files:

```
Parse the form file MainUnit.fmx and tell me what components it contains.
```

Claude calls `parse_form` and reports the component tree:

```
TMainForm (TForm)
  +-- Panel1 (TPanel)
  |   +-- Label1 (TLabel) - Text: 'Welcome'
  |   +-- Button1 (TButton) - Text: 'Click Me'
  +-- Tina4HTMLRender1 (TTina4HTMLRender)
  +-- Tina4REST1 (TTina4REST) - BaseUrl: 'https://api.example.com'
```

This lets Claude understand existing projects before modifying them.

---

7. Walkthrough: Building a Weather App

Let us walk through a complete vibe coding session. You will ask Claude to build a weather app from scratch, and Claude will do all the work -- writing code, compiling, running, and testing.

Step 1: Describe What You Want

```
Build an FMX application that shows the current weather for a city.
Use TTina4HTMLRender to display the weather data as styled HTML.
Use TTina4REST to fetch data from wttr.in (a free weather API).
Add a text input for the city name and a button to fetch the weather.
```

Step 2: Claude Generates the Project

Claude creates the project structure and writes the code:

```
// WeatherApp.dpr
program WeatherApp;

uses
  FMX.Forms,
  WeatherMain in 'WeatherMain.pas' {FormWeather};

{$R *.res}

begin
  Application.Initialize;
  Application.CreateForm(TFormWeather, FormWeather);
  Application.Run;
end.
```

```
// WeatherMain.pas
unit WeatherMain;

interface

uses
  System.SysUtils, System.Classes, System.JSON,
  FMX.Types, FMX.Controls, FMX.Forms, FMX.StdCtrls,
  FMX.Edit, FMX.Layouts,
  Tina4REST, Tina4HTMLRender;

type
  TFormWeather = class(TForm)
    LayoutTop: TLayout;
    EditCity: TEdit;
    ButtonFetch: TButton;
    Tina4REST1: TTina4REST;
    Tina4HTMLRender1: TTina4HTMLRender;
    procedure FormCreate(Sender: TObject);
    procedure ButtonFetchClick(Sender: TObject);
  private
    procedure ShowWeather(const City, Condition, TempC: string);
    procedure ShowError(const Msg: string);
  end;

var
  FormWeather: TFormWeather;

implementation
```

```
{$R *.fmx}

procedure TFormWeather.FormCreate(Sender: TObject);
begin
  Tina4REST1.BaseUrl := 'https://wttr.in';
  EditCity.Text := 'Cape Town';

  Tina4HTMLRender1.HTML.Text :=
    '<div style="padding: 20px; font-family: Arial;">' +
    '  <h2 style="color: #2c3e50;">Enter a city and click Fetch</h2>' +
    '</div>';
end;

procedure TFormWeather.ButtonFetchClick(Sender: TObject);
var
  StatusCode: Integer;
  Response: TJSONObject;
  Current: TJSONObject;
  Condition, TempC: string;
begin
  if EditCity.Text.Trim.IsEmpty then
  begin
    ShowError('Please enter a city name.');
    Exit;
  end;

  Tina4HTMLRender1.HTML.Text :=
    '<div style="padding: 20px; text-align: center;">' +
    '  <p style="color: #7f8c8d;">Loading weather data...</p>' +
    '</div>';

  try
    Response := Tina4REST1.Get(StatusCode,
      '/' + EditCity.Text.Trim + '?format=j1');
    try
      if StatusCode = 200 then
      begin
        Current := Response.GetValue<TJSONObject>('current_condition')
          .GetValue<TJSONObject>(0);
        Condition := Current.GetValue<TJSONArray>('weatherDesc')
          .Items[0].GetValue<string>('value');
        TempC := Current.GetValue<string>('temp_C');
        ShowWeather(EditCity.Text.Trim, Condition, TempC);
      end
      else
        ShowError('City not found. Status: ' + StatusCode.ToString);
    finally
      Response.Free;
    end;
  except
    on E: Exception do
      ShowError('Network error: ' + E.Message);
  end;
end;

procedure TFormWeather.ShowWeather(const City, Condition, TempC: string);
begin
  Tina4HTMLRender1.HTML.Text :=
    '<div style="padding: 20px; font-family: Arial;">' +
```

```
    '  <h2 style="color: #2c3e50;">' + City + '</h2>' +
    '  <div style="background: #ecf0f1; border-radius: 8px; ' +
    '    padding: 20px; margin-top: 10px;">' +
    '    <p style="font-size: 48px; margin: 0; color: #2980b9;">' +
    '      ' + TempC + '&deg;C</p>' +
    '    <p style="font-size: 18px; color: #7f8c8d; margin-top: 8px;">' +
    '      ' + Condition + '</p>' +
    '  </div>' +
    '</div>';
end;

procedure TFormWeather.ShowError(const Msg: string);
begin
  Tina4HTMLRender1.HTML.Text :=
    '<div style="padding: 20px; font-family: Arial;">' +
    '  <div style="background: #e74c3c; color: white; ' +
    '    padding: 15px; border-radius: 8px;">' +
    '    <p>' + Msg + '</p>' +
    '  </div>' +
    '</div>';
end;

end.
```

Step 3: Claude Compiles and Runs

Claude compiles the project. If there are errors, it reads them, fixes the code, and recompiles. Once compilation succeeds, it launches the app.

Step 4: Claude Tests the App

Claude types "London" into the city field, clicks Fetch, takes a screenshot, and reports: "The app shows London at 14 degrees Celsius, Partly Cloudy."

If the label layout is off or the button does not respond, Claude sees the problem, edits the code, recompiles, and tests again.

Step 5: Iterate

```
Make the temperature larger and add a humidity reading.
Also add a dropdown with preset cities: Cape Town, London, Tokyo, New York.
```

Claude modifies the code, recompiles, relaunches, and verifies. Each iteration takes seconds.

---

8. Walkthrough: Modifying an Existing Form

You have an existing project. You want Claude to add features without breaking what works.

Step 1: Let Claude Understand the Project

```
Read the form file MainUnit.fmx and the unit MainUnit.pas.
Tell me what the app currently does.
```

Claude parses both files and gives you a summary: which components exist, what events are wired, what the app does when you click each button.

Step 2: Request a Change

```
Add a new button called ButtonExport that exports the MemTable data to a CSV file.
Wire up the click event. The file should be saved to the user's Documents folder.
```

Step 3: Claude Modifies the Files

Claude edits the `.fmx` form to add the button component:

```
object ButtonExport: TButton
  Position.X = 220.000000000000000000
  Position.Y = 10.000000000000000000
  Width = 120.000000000000000000
  Height = 32.000000000000000000
  Text = 'Export CSV'
  OnClick = ButtonExportClick
end
```

Claude edits the `.pas` unit to add the field declaration and event handler:

```
procedure TMainForm.ButtonExportClick(Sender: TObject);
var
  FileName: string;
  SL: TStringList;
  I, J: Integer;
begin
  FileName := TPath.Combine(TPath.GetDocumentsPath, 'export.csv');
  SL := TStringList.Create;
  try
    // Header row
    var Header := '';
    for I := 0 to FDMemTable1.FieldCount - 1 do
    begin
      if I > 0 then Header := Header + ',';
      Header := Header + FDMemTable1.Fields[I].FieldName;
    end;
    SL.Add(Header);

    // Data rows
    FDMemTable1.First;
    while not FDMemTable1.Eof do
    begin
      var Row := '';
      for J := 0 to FDMemTable1.FieldCount - 1 do
      begin
        if J > 0 then Row := Row + ',';
        Row := Row + '"' + FDMemTable1.Fields[J].AsString.Replace('"', '""') + '"';
      end;
      SL.Add(Row);
      FDMemTable1.Next;
    end;

    SL.SaveToFile(FileName);
    ShowMessage('Exported to ' + FileName);
  finally
    SL.Free;
  end;
end;
```

Step 4: Claude Compiles and Tests

Claude compiles the modified project. It launches the app, loads some data, clicks the Export button, and confirms the CSV file was created.

---

9. Tips for Effective AI-Assisted Delphi Development

Be Specific About Components

Bad prompt:

```
Make a form that shows data.
```

Good prompt:

```
Create an FMX form with TTina4REST connected to https://api.example.com.
Add a TTina4RESTRequest that fetches /products into an FDMemTable.
Display the data in a TTina4HTMLRender using a Twig template
with a table showing name, price, and category columns.
```

The more specific you are about which Tina4 components to use, the better the generated code.

Reference Existing Code

```
Look at how CustomerForm.pas uses TTina4RESTRequest and apply
the same pattern to build a ProductForm.
```

Claude reads your existing code and replicates the patterns. Your codebase stays consistent.

Let Claude Fix Its Own Mistakes

When compilation fails, do not fix it yourself. Paste the error or just say:

```
Fix the compilation errors.
```

Claude reads the compiler output, understands the error, and fixes it. This is faster than you doing it because Claude remembers every line it wrote.

Use the Preview Bridge for Visual Feedback

With the preview bridge running, Claude can verify visual layout:

```
The stats cards should be in a horizontal row, not stacked vertically.
Fix the layout.
```

Claude takes a screenshot, sees the problem, adjusts the HTML/CSS in the TTina4HTMLRender, recompiles, and checks again.

Keep Your CLAUDE.md Updated

Create a `CLAUDE.md` in your project root that describes:

- Which compiler to use
- Where your templates live
- What API endpoints your app connects to

- Any project-specific conventions

```
# My Delphi Project

## Build
- Compiler: RAD Studio 12 (dcc64)
- Platform: Win64
- Project file: src/MyApp.dpr

## Conventions
- All REST endpoints are in DataModule1
- HTML templates are in the templates/ directory
- Use TTina4HTMLRender for all UI rendering
- Use Twig templates, not raw HTML strings
```

Claude reads this file first and follows your conventions from the start.

---

10. Exercise: Build a Calculator from Scratch

Set up the MCP server and use Claude Code to build a calculator application. Do not write any code yourself. Use only natural language prompts.

Requirements

- FMX application with a TTina4HTMLRender for the display
- HTML buttons for digits 0-9, operators (+, -, *, /), equals, and clear
- Use `onclick` RTTI calls from HTML buttons to Pascal methods
- Support chained operations (2 + 3 * 4 should work left-to-right)
- Display the current expression and result

Suggested Prompts

Start with:

```
Create a new FMX calculator app. Use TTina4HTMLRender for the entire UI.
Render calculator buttons as an HTML grid. Wire onclick events using
RTTI to call Pascal methods. Show the current expression at the top.
```

Then iterate:

```
The buttons are too small. Make them 60x60 pixels with larger font.
Add a decimal point button.
Handle division by zero gracefully.
```

Solution

The solution is not code you write. The solution is the conversation with Claude. By the end, you should have:

- A working calculator app compiled and running
- HTML-rendered buttons that call Pascal methods via RTTI
- Clean separation between the UI (HTML/CSS) and logic (Pascal)

- Claude having fixed at least one bug it introduced during development

The exercise is complete when the calculator handles `12.5 + 7.5 = 20` correctly and Claude has verified it by interacting with the running app.

---

11. Common MCP Issues

Problem	Cause	Fix
Claude does not see pascal-dev tools	`.mcp.json` not in project root	Move `.mcp.json` to the directory where you run `claude`
`uv: command not found`	uv not installed or not in PATH	Install uv and restart your terminal
Compiler not found	Auto-detect failed	Tell Claude the compiler path explicitly
Preview bridge not starting	Wrong Python path in `launch.json`	Use the `.venv` Python, not system Python
GUI app launches but Claude cannot see it	Preview bridge not configured	Add `launch.json` configuration as shown in Section 5
Click tool reports "window not found"	App window title changed or app closed	Relaunch the app and try again
Compilation fails with missing units	Search path not configured	Tell Claude which directories to include in the search path

---

Summary

The MCP server bridges the gap between AI code generation and AI code verification. Claude does not just write Delphi code and hope it works. It compiles, runs, sees, interacts, and iterates. The feedback loop that used to require a human switching between IDE and terminal now happens inside a single conversation.

Set up the MCP server once. Use it on every project. The time you invest in the five-minute setup pays back on the first non-trivial feature Claude builds for you.

Building a Complete Application

The Admin Dashboard

Picture the screen. A login form rendered in HTML with styled inputs and a submit button. You type credentials, click Login, and the form posts to an API endpoint. The bearer token comes back. The dashboard appears -- stat cards showing user counts and active sessions, a sidebar menu, a user table with search and pagination. A WebSocket connection pushes live notifications. A red badge increments on the bell icon.

This chapter builds that application from scratch using every Tina4 Delphi component: TTina4REST, TTina4RESTRequest, TTina4HTMLRender, TTina4HTMLPages, TTina4Twig, TTina4JSONAdapter, and TTina4WebSocketClient. By the end, you will have a complete, working desktop admin dashboard -- and you will have seen how all the pieces from previous chapters fit together in a real application.

---

1. What We Are Building

The admin dashboard has four pages:

- **Login** -- HTML form, POST to API, store bearer token
- **Dashboard** -- Stat cards with user count, active sessions, recent activity
- **Users** -- List, search, create, edit, delete users via REST
- **Settings** -- Application configuration form

Plus cross-cutting concerns:

- **Authentication** -- Bearer token stored in TTina4REST, checked on every page
- **WebSocket notifications** -- Live updates pushed from the server
- **Error handling** -- Network errors, expired tokens, validation failures

---

2. Project Setup

Create a new FMX application in Delphi. Name the project `AdminDashboard`.

File Structure

```
AdminDashboard/
  AdminDashboard.dpr          -- Project file
  MainUnit.pas                -- Main form with all components
  MainUnit.fmx                -- FMX form definition
  templates/
    login.html                -- Login page template
    dashboard.html            -- Dashboard page template
    users.html                -- Users list template
    user-detail.html          -- User detail/edit template
    settings.html             -- Settings page template
    layout.html               -- Shared layout with sidebar
```

The Main Form

Drop these components on the form:

Component	Name	Purpose
TTina4REST	REST	Base URL and authentication
TTina4RESTRequest	RESTUsers	Fetch user list
TTina4RESTRequest	RESTStats	Fetch dashboard stats
TFDMemTable	MemUsers	Store user data
TFDMemTable	MemStats	Store stats data
TTina4HTMLRender	Renderer	Display HTML content
TTina4HTMLPages	Pages	Page navigation
TDataSource	DSUsers	Bridge MemTable to data-aware controls

Project File

```
// AdminDashboard.dpr
program AdminDashboard;

uses
  FMX.Forms,
  MainUnit in 'MainUnit.pas' {FormMain};

{$R *.res}

begin
  Application.Initialize;
  Application.CreateForm(TFormMain, FormMain);
  Application.Run;
end.
```

---

3. Main Unit -- Declarations

```
unit MainUnit;

interface

uses
  System.SysUtils, System.Classes, System.JSON, System.IOUtils,
  System.Generics.Collections, System.Threading,
  FMX.Types, FMX.Controls, FMX.Forms, FMX.StdCtrls, FMX.Layouts,
  FMX.Dialogs,
  FireDAC.Stan.Intf, FireDAC.Stan.Option, FireDAC.Stan.Param,
  FireDAC.Stan.Error, FireDAC.DatS, FireDAC.Phys.Intf,
  FireDAC.DApt.Intf, FireDAC.Comp.DataSet, FireDAC.Comp.Client,
  Data.DB,
  Tina4REST, Tina4RESTRequest, Tina4HTMLRender, Tina4HTMLPages,
  Tina4Twig, Tina4JSONAdapter, Tina4Core;

type
  TFormMain = class(TForm)
    REST: TTina4REST;
    RESTUsers: TTina4RESTRequest;
    RESTStats: TTina4RESTRequest;
    MemUsers: TFDMemTable;
    MemStats: TFDMemTable;
    Renderer: TTina4HTMLRender;
    Pages: TTina4HTMLPages;
    DSUsers: TDataSource;
    procedure FormCreate(Sender: TObject);
  private
    FBearerToken: string;
    FCurrentUserId: string;
    FSearchFilter: string;
    FCurrentPage: Integer;
    FPageSize: Integer;
    FNotificationCount: Integer;

    { Authentication }
    procedure DoLogin(const Username, Password: string);
    procedure DoLogout;
    function IsAuthenticated: Boolean;

    { Page setup }
    procedure SetupPages;
    procedure SetupLoginPage;
    procedure SetupDashboardPage;
    procedure SetupUsersPage;
    procedure SetupSettingsPage;

    { REST operations }
    procedure FetchStats;
    procedure FetchUsers;
    procedure CreateUser(const Name, Email, Role: string);
    procedure UpdateUser(const Id, Name, Email, Role: string);
    procedure DeleteUser(const Id: string);

    { Event handlers for HTML }
    procedure HandleLogin(const FormData: string);
    procedure HandleSearch(const Query: string);
```

```
    procedure HandlePageChange(const Page: string);
    procedure HandleEditUser(const UserId: string);
    procedure HandleDeleteUser(const UserId: string);
    procedure HandleSaveUser(const FormData: string);
    procedure HandleSaveSettings(const FormData: string);

    { UI helpers }
    procedure ShowNotification(const Msg: string; IsError: Boolean = False);
    procedure RefreshCurrentPage;
    function BuildUserTableHTML: string;
    function BuildStatsHTML: string;
    function ParseFormData(const Raw: string): TStringList;
  public
    { RTTI-callable methods for HTML onclick }
    procedure NavigateTo(const PageName: string);
    procedure OnLoginSubmit(const Username, Password: string);
    procedure OnSearchSubmit(const Query: string);
    procedure OnEditClick(const UserId: string);
    procedure OnDeleteClick(const UserId: string);
    procedure OnPageClick(const Page: string);
  end;

var
  FormMain: TFormMain;

implementation

{$R *.fmx}
```

---

4. Initialization

```
procedure TFormMain.FormCreate(Sender: TObject);
begin
  FCurrentPage := 1;
  FPageSize := 10;
  FNotificationCount := 0;
  FSearchFilter := '';

  { REST configuration }
  REST.BaseUrl := 'https://api.example.com/v1';

  { Users request }
  RESTUsers.Tina4REST := REST;
  RESTUsers.EndPoint := '/users';
  RESTUsers.RequestType := TTina4RequestType.Get;
  RESTUsers.DataKey := 'records';
  RESTUsers.MemTable := MemUsers;
  RESTUsers.SyncMode := TTina4RestSyncMode.Clear;

  { Stats request }
  RESTStats.Tina4REST := REST;
  RESTStats.EndPoint := '/stats';
  RESTStats.RequestType := TTina4RequestType.Get;
  RESTStats.DataKey := 'stats';
  RESTStats.MemTable := MemStats;

  { HTML renderer }
  Renderer.CacheEnabled := True;
  Renderer.CacheDir := TPath.Combine(TPath.GetDocumentsPath, 'AdminCache');
  Renderer.RegisterObject('App', Self);

  { Pages }
  Pages.Renderer := Renderer;
  Pages.TwigTemplatePath := TPath.Combine(ExtractFilePath(ParamStr(0)), 'templates');

  SetupPages;

  { Page navigation guard }
  Pages.OnBeforeNavigate := procedure(Sender: TObject;
    const FromPage, ToPage: string; var Allow: Boolean)
  begin
    if (ToPage <> 'login') and (not IsAuthenticated) then
    begin
      Allow := False;
      Pages.NavigateTo('login');
    end;
  end;
end;
```

---

5. Authentication

```
function TFormMain.IsAuthenticated: Boolean;
begin
  Result := not FBearerToken.IsEmpty;
end;

procedure TFormMain.DoLogin(const Username, Password: string);
var
  StatusCode: Integer;
  Response: TJSONObject;
  Body: string;
begin
  Body := Format('{"username": "%s", "password": "%s"}',
    [Username, Password]);

  Response := REST.Post(StatusCode, '/auth/login', '', Body);
  try
    if StatusCode = 200 then
    begin
      FBearerToken := Response.GetValue<string>('token');
      REST.SetBearer(FBearerToken);

      { Navigate to dashboard }
      Pages.NavigateTo('dashboard');
      FetchStats;
      FetchUsers;
    end
    else
    begin
      var Msg := 'Login failed.';
      if Assigned(Response) then
        Msg := Response.GetValue<string>('message', 'Invalid credentials.');
      ShowNotification(Msg, True);
    end;
  finally
    Response.Free;
  end;
end;

procedure TFormMain.DoLogout;
begin
  FBearerToken := '';
  REST.SetBearer('');
  Pages.NavigateTo('login');
end;

{ RTTI-callable from HTML onclick }
procedure TFormMain.OnLoginSubmit(const Username, Password: string);
begin
  DoLogin(Username, Password);
end;
```

---

6. Page Definitions

```
procedure TFormMain.SetupPages;
begin
  SetupLoginPage;
  SetupDashboardPage;
  SetupUsersPage;
  SetupSettingsPage;
end;

procedure TFormMain.SetupLoginPage;
var
  Page: TTina4Page;
begin
  Page := Pages.Pages.Add;
  Page.PageName := 'login';
  Page.IsDefault := True;
  Page.HTMLContent.Text :=
    '<div style="max-width: 400px; margin: 80px auto; ' +
    '  font-family: Arial, sans-serif;">' +
    '  <h1 style="text-align: center; color: #2c3e50;">Admin Dashboard</h1>' +
    '  <div style="background: white; padding: 30px; border-radius: 8px; ' +
    '    box-shadow: 0 2px 10px rgba(0,0,0,0.1);">' +
    '    <div style="margin-bottom: 15px;">' +
    '      <label style="display: block; margin-bottom: 5px; ' +
    '        color: #555; font-size: 14px;">Username</label>' +
    '      <input type="text" id="loginUser" ' +
    '        style="width: 100%; padding: 10px; border: 1px solid #ddd; ' +
    '        border-radius: 4px; font-size: 14px;" ' +
    '        placeholder="Enter username">' +
    '    </div>' +
    '    <div style="margin-bottom: 20px;">' +
    '      <label style="display: block; margin-bottom: 5px; ' +
    '        color: #555; font-size: 14px;">Password</label>' +
    '      <input type="password" id="loginPass" ' +
    '        style="width: 100%; padding: 10px; border: 1px solid #ddd; ' +
    '        border-radius: 4px; font-size: 14px;" ' +
    '        placeholder="Enter password">' +
    '    </div>' +
    '    <button onclick="App:OnLoginSubmit(' +
    '      document.getElementById(''loginUser'').value, ' +
    '      document.getElementById(''loginPass'').value)" ' +
    '      style="width: 100%; padding: 12px; background: #3498db; ' +
    '      color: white; border: none; border-radius: 4px; ' +
    '      font-size: 16px; cursor: pointer;">Login</button>' +
    '    <div id="loginError" style="display: none; margin-top: 15px; ' +
    '      padding: 10px; background: #e74c3c; color: white; ' +
    '      border-radius: 4px;"></div>' +
    '  </div>' +
    '</div>';
end;
```

The login page uses HTML inputs rendered by TTina4HTMLRender. The button's `onclick` calls `App:OnLoginSubmit(...)` which resolves via RTTI to the `TFormMain.OnLoginSubmit` method. The parameters are extracted from the input fields using `document.getElementById`.

---

7. Dashboard Page with Stats

```
procedure TFormMain.SetupDashboardPage;
var
  Page: TTina4Page;
begin
  Page := Pages.Pages.Add;
  Page.PageName := 'dashboard';
  { Content is set dynamically after stats load }
  Page.HTMLContent.Text :=
    '<div style="padding: 20px; font-family: Arial;">' +
    '  <p>Loading dashboard...</p>' +
    '</div>';
end;

function TFormMain.BuildStatsHTML: string;
var
  UserCount, ActiveSessions, Revenue, OrderCount: string;
begin
  UserCount := '0';
  ActiveSessions := '0';
  Revenue := '$0';
  OrderCount := '0';

  if MemStats.Active and (MemStats.RecordCount > 0) then
  begin
    MemStats.First;
    UserCount := MemStats.FieldByName('user_count').AsString;
    ActiveSessions := MemStats.FieldByName('active_sessions').AsString;
    Revenue := '$' + MemStats.FieldByName('revenue').AsString;
    OrderCount := MemStats.FieldByName('order_count').AsString;
  end;

  Result :=
    '<div style="font-family: Arial; padding: 20px;">' +

    { Navigation bar }
    '  <div style="display: flex; justify-content: space-between; ' +
    '    align-items: center; margin-bottom: 30px; padding-bottom: 15px; ' +
    '    border-bottom: 2px solid #ecf0f1;">' +
    '    <h1 style="margin: 0; color: #2c3e50;">Dashboard</h1>' +
    '    <div>' +
    '      <span onclick="App:NavigateTo(''dashboard'')" ' +
    '        style="margin-right: 15px; color: #3498db; ' +
    '        cursor: pointer; font-weight: bold;">Dashboard</span>' +
    '      <span onclick="App:NavigateTo(''users'')" ' +
    '        style="margin-right: 15px; color: #7f8c8d; cursor: pointer;">Users</span>' +
    '      <span onclick="App:NavigateTo(''settings'')" ' +
    '        style="margin-right: 15px; color: #7f8c8d; cursor: pointer;">Settings</span>' +
    '      <span onclick="App:DoLogout" ' +
    '        style="color: #e74c3c; cursor: pointer;">Logout</span>' +
    '    </div>' +
    '  </div>' +

    { Stat cards row }
    '  <div style="display: flex; gap: 20px; margin-bottom: 30px;">' +

    '    <div style="flex: 1; background: #3498db; color: white; ' +
    '      padding: 20px; border-radius: 8px;">' +
```

```
    '      <p style="font-size: 14px; margin: 0; opacity: 0.8;">Total Users</p>' +
    '      <p style="font-size: 36px; margin: 5px 0 0 0; font-weight: bold;">' +
    UserCount + '</p>' +
    '    </div>' +

    '    <div style="flex: 1; background: #2ecc71; color: white; ' +
    '      padding: 20px; border-radius: 8px;">' +
    '      <p style="font-size: 14px; margin: 0; opacity: 0.8;">Active Sessions</p>' +
    '      <p style="font-size: 36px; margin: 5px 0 0 0; font-weight: bold;">' +
    ActiveSessions + '</p>' +
    '    </div>' +

    '    <div style="flex: 1; background: #9b59b6; color: white; ' +
    '      padding: 20px; border-radius: 8px;">' +
    '      <p style="font-size: 14px; margin: 0; opacity: 0.8;">Revenue</p>' +
    '      <p style="font-size: 36px; margin: 5px 0 0 0; font-weight: bold;">' +
    Revenue + '</p>' +
    '    </div>' +

    '    <div style="flex: 1; background: #e67e22; color: white; ' +
    '      padding: 20px; border-radius: 8px;">' +
    '      <p style="font-size: 14px; margin: 0; opacity: 0.8;">Orders</p>' +
    '      <p style="font-size: 36px; margin: 5px 0 0 0; font-weight: bold;">' +
    OrderCount + '</p>' +
    '    </div>' +

    '  </div>' +

    { Notification area }
    '  <div id="notificationArea" style="display: none; ' +
    '    padding: 10px; border-radius: 4px; margin-bottom: 20px;"></div>' +

    '</div>';
end;

procedure TFormMain.FetchStats;
begin
  RESTStats.OnExecuteDone := procedure(Sender: TObject)
  begin
    TThread.Synchronize(nil, procedure
    begin
      var Page := Pages.Pages.FindPage('dashboard');
      if Assigned(Page) then
      begin
        Page.HTMLContent.Text := BuildStatsHTML;
        if Pages.ActivePage = 'dashboard' then
          Pages.NavigateTo('dashboard');
      end;
    end);
  end;
  RESTStats.ExecuteRESTCallAsync;
end;
```

The dashboard fetches stats asynchronously. When the REST call completes, it rebuilds the HTML and refreshes the page. The stat cards use colored backgrounds with large numbers -- a common dashboard pattern.

---

8. Users Page -- CRUD Operations

```
procedure TFormMain.SetupUsersPage;
var
  Page: TTina4Page;
begin
  Page := Pages.Pages.Add;
  Page.PageName := 'users';
  Page.HTMLContent.Text :=
    '<div style="padding: 20px; font-family: Arial;">' +
    '  <p>Loading users...</p>' +
    '</div>';
end;

function TFormMain.BuildUserTableHTML: string;
var
  Rows: string;
  RowClass: string;
  I: Integer;
begin
  Rows := '';
  if MemUsers.Active then
  begin
    MemUsers.First;
    I := 0;
    while not MemUsers.Eof do
    begin
      if I mod 2 = 0 then
        RowClass := 'background: #f9f9f9;'
      else
        RowClass := 'background: white;';

      Rows := Rows +
        '<tr style="' + RowClass + '">' +
        '  <td style="padding: 10px; border-bottom: 1px solid #eee;">' +
             MemUsers.FieldByName('id').AsString + '</td>' +
        '  <td style="padding: 10px; border-bottom: 1px solid #eee;">' +
             MemUsers.FieldByName('name').AsString + '</td>' +
        '  <td style="padding: 10px; border-bottom: 1px solid #eee;">' +
             MemUsers.FieldByName('email').AsString + '</td>' +
        '  <td style="padding: 10px; border-bottom: 1px solid #eee;">' +
             MemUsers.FieldByName('role').AsString + '</td>' +
        '  <td style="padding: 10px; border-bottom: 1px solid #eee;">' +
        '    <span onclick="App:OnEditClick(''' +
               MemUsers.FieldByName('id').AsString + ''')" ' +
        '      style="color: #3498db; cursor: pointer; margin-right: 10px;">Edit</span>' +
        '    <span onclick="App:OnDeleteClick(''' +
               MemUsers.FieldByName('id').AsString + ''')" ' +
        '      style="color: #e74c3c; cursor: pointer;">Delete</span>' +
        '  </td>' +
        '</tr>';

      MemUsers.Next;
      Inc(I);
    end;
  end;

  if Rows.IsEmpty then
    Rows := '<tr><td colspan="5" style="padding: 20px; text-align: center; ' +
```

```
    'color: #999;">No users found.</td></tr>';

Result :=
  '<div style="font-family: Arial; padding: 20px;">' +

  { Navigation }
  '  <div style="display: flex; justify-content: space-between; ' +
  '    align-items: center; margin-bottom: 30px; padding-bottom: 15px; ' +
  '    border-bottom: 2px solid #ecf0f1;">' +
  '    <h1 style="margin: 0; color: #2c3e50;">Users</h1>' +
  '    <div>' +
  '      <span onclick="App:NavigateTo(''dashboard'')" ' +
  '        style="margin-right: 15px; color: #7f8c8d; cursor: pointer;">Dashboard</span>' +
  '      <span onclick="App:NavigateTo(''users'')" ' +
  '        style="margin-right: 15px; color: #3498db; ' +
  '        cursor: pointer; font-weight: bold;">Users</span>' +
  '      <span onclick="App:NavigateTo(''settings'')" ' +
  '        style="margin-right: 15px; color: #7f8c8d; cursor: pointer;">Settings</span>' +
  '      <span onclick="App:DoLogout" ' +
  '        style="color: #e74c3c; cursor: pointer;">Logout</span>' +
  '    </div>' +
  '  </div>' +

  { Search bar }
  '  <div style="display: flex; gap: 10px; margin-bottom: 20px;">' +
  '    <input type="text" id="searchInput" placeholder="Search users..." ' +
  '      style="flex: 1; padding: 10px; border: 1px solid #ddd; ' +
  '      border-radius: 4px; font-size: 14px;" ' +
  '      value="' + FSearchFilter + '">' +
  '    <button onclick="App:OnSearchSubmit(' +
  '      document.getElementById(''searchInput'').value)" ' +
  '      style="padding: 10px 20px; background: #3498db; color: white; ' +
  '      border: none; border-radius: 4px; cursor: pointer;">Search</button>' +
  '    <button onclick="App:NavigateTo(''user-create'')" ' +
  '      style="padding: 10px 20px; background: #2ecc71; color: white; ' +
  '      border: none; border-radius: 4px; cursor: pointer;">+ New User</button>' +
  '  </div>' +

  { User table }
  '  <table style="width: 100%; border-collapse: collapse;">' +
  '    <thead>' +
  '      <tr style="background: #2c3e50; color: white;">' +
  '        <th style="padding: 12px; text-align: left;">ID</th>' +
  '        <th style="padding: 12px; text-align: left;">Name</th>' +
  '        <th style="padding: 12px; text-align: left;">Email</th>' +
  '        <th style="padding: 12px; text-align: left;">Role</th>' +
  '        <th style="padding: 12px; text-align: left;">Actions</th>' +
  '      </tr>' +
  '    </thead>' +
  '    <tbody>' + Rows + '</tbody>' +
  '  </table>' +

  { Pagination }
  '  <div style="margin-top: 20px; text-align: center;">' +
  '    <span onclick="App:OnPageClick(''' + (FCurrentPage - 1).ToString + ''')" ' +
  '      style="padding: 8px 16px; margin: 0 5px; background: #ecf0f1; ' +
  '      border-radius: 4px; cursor: pointer;">Previous</span>' +
  '    <span style="padding: 8px 16px; margin: 0 5px; background: #3498db; ' +
  '      color: white; border-radius: 4px;">Page ' + FCurrentPage.ToString + '</span>' +
```

```
    '    <span onclick="App:OnPageClick(''' + (FCurrentPage + 1).ToString + ''')" ' +
    '      style="padding: 8px 16px; margin: 0 5px; background: #ecf0f1; ' +
    '      border-radius: 4px; cursor: pointer;">Next</span>' +
    '  </div>' +

    '</div>';
end;

procedure TFormMain.FetchUsers;
var
  QueryParams: string;
begin
  QueryParams := Format('page=%d&limit=%d', [FCurrentPage, FPageSize]);
  if not FSearchFilter.IsEmpty then
    QueryParams := QueryParams + '&search=' + FSearchFilter;

  RESTUsers.EndPoint := '/users';
  RESTUsers.QueryParams := QueryParams;
  RESTUsers.OnExecuteDone := procedure(Sender: TObject)
  begin
    TThread.Synchronize(nil, procedure
    begin
      var Page := Pages.Pages.FindPage('users');
      if Assigned(Page) then
      begin
        Page.HTMLContent.Text := BuildUserTableHTML;
        if Pages.ActivePage = 'users' then
          Pages.NavigateTo('users');
      end;
    end);
  end;
  RESTUsers.ExecuteRESTCallAsync;
end;
```

Creating a User

```
procedure TFormMain.CreateUser(const Name, Email, Role: string);
var
  StatusCode: Integer;
  Body: string;
  Response: TJSONObject;
begin
  Body := Format('{"name": "%s", "email": "%s", "role": "%s"}',
    [Name, Email, Role]);

  Response := REST.Post(StatusCode, '/users', '', Body);
  try
    if StatusCode in [200, 201] then
    begin
      ShowNotification('User created successfully.');
      FetchUsers;
      Pages.NavigateTo('users');
    end
    else
      ShowNotification('Failed to create user.', True);
  finally
    Response.Free;
  end;
end;
```

Updating a User

```
procedure TFormMain.UpdateUser(const Id, Name, Email, Role: string);
var
  StatusCode: Integer;
  Body: string;
  Response: TJSONObject;
begin
  Body := Format('{"name": "%s", "email": "%s", "role": "%s"}',
    [Name, Email, Role]);

  Response := REST.Patch(StatusCode, '/users/' + Id, '', Body);
  try
    if StatusCode = 200 then
    begin
      ShowNotification('User updated successfully.');
      FetchUsers;
      Pages.NavigateTo('users');
    end
    else
      ShowNotification('Failed to update user.', True);
  finally
    Response.Free;
  end;
end;
```

Deleting a User

```
procedure TFormMain.DeleteUser(const Id: string);
var
  StatusCode: Integer;
  Response: TJSONObject;
begin
  Response := REST.Delete(StatusCode, '/users/' + Id, '');
  try
    if StatusCode in [200, 204] then
    begin
      ShowNotification('User deleted.');
      FetchUsers;
    end
    else
      ShowNotification('Failed to delete user.', True);
  finally
    Response.Free;
  end;
end;
```

---

9. Settings Page

```
procedure TFormMain.SetupSettingsPage;
var
  Page: TTina4Page;
begin
  Page := Pages.Pages.Add;
  Page.PageName := 'settings';
  Page.HTMLContent.Text :=
    '<div style="font-family: Arial; padding: 20px;">' +

    { Navigation }
    '  <div style="display: flex; justify-content: space-between; ' +
    '    align-items: center; margin-bottom: 30px; padding-bottom: 15px; ' +
    '    border-bottom: 2px solid #ecf0f1;">' +
    '    <h1 style="margin: 0; color: #2c3e50;">Settings</h1>' +
    '    <div>' +
    '      <span onclick="App:NavigateTo(''dashboard'')" ' +
    '        style="margin-right: 15px; color: #7f8c8d; cursor: pointer;">Dashboard</span>' +
    '      <span onclick="App:NavigateTo(''users'')" ' +
    '        style="margin-right: 15px; color: #7f8c8d; cursor: pointer;">Users</span>' +
    '      <span onclick="App:NavigateTo(''settings'')" ' +
    '        style="margin-right: 15px; color: #3498db; ' +
    '        cursor: pointer; font-weight: bold;">Settings</span>' +
    '      <span onclick="App:DoLogout" ' +
    '        style="color: #e74c3c; cursor: pointer;">Logout</span>' +
    '    </div>' +
    '  </div>' +

    { Settings form }
    '  <div style="max-width: 600px;">' +
    '    <div style="margin-bottom: 15px;">' +
    '      <label style="display: block; margin-bottom: 5px; color: #555;">App Name</label>' +
    '      <input type="text" id="settAppName" value="Admin Dashboard" ' +
    '        style="width: 100%; padding: 10px; border: 1px solid #ddd; ' +
    '        border-radius: 4px;">' +
    '    </div>' +
    '    <div style="margin-bottom: 15px;">' +
    '      <label style="display: block; margin-bottom: 5px; color: #555;">API URL</label>' +
    '      <input type="text" id="settApiUrl" value="https://api.example.com/v1" ' +
    '        style="width: 100%; padding: 10px; border: 1px solid #ddd; ' +
    '        border-radius: 4px;">' +
    '    </div>' +
    '    <div style="margin-bottom: 15px;">' +
    '      <label style="display: block; margin-bottom: 5px; color: #555;">' +
    '        Items Per Page</label>' +
    '      <select id="settPageSize" style="width: 100%; padding: 10px; ' +
    '        border: 1px solid #ddd; border-radius: 4px;">' +
    '        <option value="10">10</option>' +
    '        <option value="25">25</option>' +
    '        <option value="50">50</option>' +
    '      </select>' +
    '    </div>' +
    '    <div style="margin-bottom: 20px;">' +
    '      <label style="display: block; margin-bottom: 5px; color: #555;">' +
    '        Enable Notifications</label>' +
    '      <input type="checkbox" id="settNotify" checked ' +
    '        style="margin-right: 8px;">' +
    '      <span style="color: #555;">Receive real-time notifications</span>' +
```

```
    '    </div>' +
    '    <button onclick="App:HandleSaveSettings(' +
    '      document.getElementById(''settApiUrl'').value)" ' +
    '      style="padding: 12px 30px; background: #3498db; color: white; ' +
    '      border: none; border-radius: 4px; font-size: 16px; ' +
    '      cursor: pointer;">Save Settings</button>' +
    '  </div>' +
    '</div>';
end;

procedure TFormMain.HandleSaveSettings(const FormData: string);
begin
  REST.BaseUrl := FormData;  // The API URL passed from the form
  ShowNotification('Settings saved.');
end;
```

---

10. RTTI Event Handlers

These methods are called from HTML `onclick` attributes via the RTTI mechanism. They bridge the HTML UI to the Pascal logic.

```
procedure TFormMain.NavigateTo(const PageName: string);
begin
  if not IsAuthenticated and (PageName <> 'login') then
  begin
    Pages.NavigateTo('login');
    Exit;
  end;

  Pages.NavigateTo(PageName);

  { Refresh data when entering certain pages }
  if PageName = 'dashboard' then
    FetchStats
  else if PageName = 'users' then
    FetchUsers;
end;

procedure TFormMain.OnSearchSubmit(const Query: string);
begin
  FSearchFilter := Query;
  FCurrentPage := 1;
  FetchUsers;
end;

procedure TFormMain.OnEditClick(const UserId: string);
var
  Page: TTina4Page;
begin
  FCurrentUserId := UserId;

  { Find the user in MemTable }
  if MemUsers.Locate('id', UserId) then
  begin
    Page := Pages.Pages.FindPage('user-edit');
    if not Assigned(Page) then
```

```
    begin
      Page := Pages.Pages.Add;
      Page.PageName := 'user-edit';
    end;

    Page.HTMLContent.Text :=
      '<div style="font-family: Arial; padding: 20px; max-width: 600px;">' +
      '  <h2 style="color: #2c3e50;">Edit User</h2>' +
      '  <div style="margin-bottom: 15px;">' +
      '    <label style="display: block; margin-bottom: 5px;">Name</label>' +
      '    <input type="text" id="editName" ' +
      '      value="' + MemUsers.FieldByName('name').AsString + '" ' +
      '      style="width: 100%; padding: 10px; border: 1px solid #ddd; ' +
      '      border-radius: 4px;">' +
      '  </div>' +
      '  <div style="margin-bottom: 15px;">' +
      '    <label style="display: block; margin-bottom: 5px;">Email</label>' +
      '    <input type="text" id="editEmail" ' +
      '      value="' + MemUsers.FieldByName('email').AsString + '" ' +
      '      style="width: 100%; padding: 10px; border: 1px solid #ddd; ' +
      '      border-radius: 4px;">' +
      '  </div>' +
      '  <div style="margin-bottom: 20px;">' +
      '    <label style="display: block; margin-bottom: 5px;">Role</label>' +
      '    <select id="editRole" style="width: 100%; padding: 10px; ' +
      '      border: 1px solid #ddd; border-radius: 4px;">' +
      '      <option value="user">User</option>' +
      '      <option value="admin">Admin</option>' +
      '      <option value="editor">Editor</option>' +
      '    </select>' +
      '  </div>' +
      '  <button onclick="App:HandleSaveUser(' +
      '    document.getElementById(''editName'').value)" ' +
      '    style="padding: 10px 20px; background: #3498db; color: white; ' +
      '    border: none; border-radius: 4px; cursor: pointer; ' +
      '    margin-right: 10px;">Save</button>' +
      '  <button onclick="App:NavigateTo(''users'')" ' +
      '    style="padding: 10px 20px; background: #95a5a6; color: white; ' +
      '    border: none; border-radius: 4px; cursor: pointer;">Cancel</button>' +
      '</div>';

    Pages.NavigateTo('user-edit');
  end;
end;

procedure TFormMain.OnDeleteClick(const UserId: string);
begin
  { In a real app, show a confirmation dialog first }
  DeleteUser(UserId);
end;

procedure TFormMain.OnPageClick(const Page: string);
var
  NewPage: Integer;
begin
  NewPage := StrToIntDef(Page, 1);
  if NewPage < 1 then NewPage := 1;
  FCurrentPage := NewPage;
  FetchUsers;
```

```
end;

procedure TFormMain.HandleSaveUser(const FormData: string);
begin
  { FormData contains the name; in a real app, read all fields }
  var Name := Renderer.GetElementValue('editName');
  var Email := Renderer.GetElementValue('editEmail');
  var Role := Renderer.GetElementValue('editRole');

  if FCurrentUserId.IsEmpty then
    CreateUser(Name, Email, Role)
  else
    UpdateUser(FCurrentUserId, Name, Email, Role);
end;
```

---

11. Notifications

```
procedure TFormMain.ShowNotification(const Msg: string; IsError: Boolean);
var
  BgColor: string;
begin
  if IsError then
    BgColor := '#e74c3c'
  else
    BgColor := '#2ecc71';

  { Update the notification area on the current page }
  Renderer.SetElementVisible('notificationArea', True);
  Renderer.SetElementStyle('notificationArea', 'background-color', BgColor);
  Renderer.SetElementStyle('notificationArea', 'color', 'white');
  Renderer.SetElementText('notificationArea', Msg);

  { Auto-hide after 3 seconds }
  TTask.Run(procedure
  begin
    Sleep(3000);
    TThread.Synchronize(nil, procedure
    begin
      Renderer.SetElementVisible('notificationArea', False);
    end);
  end);
end;
```

The notification uses DOM manipulation to show and hide a div. No page rebuild needed. The `SetElementVisible`, `SetElementStyle`, and `SetElementText` methods update the rendered HTML in place. A background task hides it after three seconds.

---

12. WebSocket Notifications

Add a WebSocket connection for real-time updates. If your API supports WebSocket, add this to `FormCreate`:

```
procedure TFormMain.SetupWebSocket;
```

```
begin
  { WebSocket for real-time notifications }
  { Note: TTina4WebSocketClient is a separate component }
  FWebSocket := TTina4WebSocketClient.Create(Self);
  FWebSocket.URL := 'wss://api.example.com/ws/notifications';

  FWebSocket.OnMessage := procedure(Sender: TObject; const Msg: string)
  begin
    TThread.Synchronize(nil, procedure
    var
      JSON: TJSONObject;
    begin
      JSON := StrToJSONObject(Msg);
      try
        if Assigned(JSON) then
        begin
          Inc(FNotificationCount);
          var NotifText := JSON.GetValue<string>('message', 'New notification');
          ShowNotification(NotifText);

          { If on dashboard, refresh stats }
          if Pages.ActivePage = 'dashboard' then
            FetchStats;
        end;
      finally
        JSON.Free;
      end;
    end);
  end;

  FWebSocket.OnDisconnect := procedure(Sender: TObject)
  begin
    TThread.Synchronize(nil, procedure
    begin
      ShowNotification('Connection lost. Reconnecting...', True);
    end);
  end;

  FWebSocket.AutoReconnect := True;
  FWebSocket.ReconnectInterval := 5000;
end;
```

After successful login, connect the WebSocket:

```
procedure TFormMain.DoLogin(const Username, Password: string);
begin
  { ... existing login code ... }

  if StatusCode = 200 then
  begin
    FBearerToken := Response.GetValue<string>('token');
    REST.SetBearer(FBearerToken);

    { Connect WebSocket after auth }
    FWebSocket.Headers.Values['Authorization'] := 'Bearer ' + FBearerToken;
    FWebSocket.Connect;

    Pages.NavigateTo('dashboard');
    FetchStats;
```

```
    FetchUsers;
  end;
end;
```

---

13. Error Handling

Every REST call in this application handles errors. Here is the complete error handling pattern used throughout:

```
procedure TFormMain.SafeRESTCall(const Method: string; const EndPoint: string;
  const Body: string; OnSuccess: TProc<TJSONObject>);
var
  StatusCode: Integer;
  Response: TJSONObject;
begin
  try
    case Method.ToUpper.Chars[0] of
      'G': Response := REST.Get(StatusCode, EndPoint, '');
      'P': Response := REST.Post(StatusCode, EndPoint, '', Body);
      'A': Response := REST.Patch(StatusCode, EndPoint, '', Body);
      'D': Response := REST.Delete(StatusCode, EndPoint, '');
    else
      Response := REST.Get(StatusCode, EndPoint, '');
    end;

    try
      case StatusCode of
        200, 201, 204:
          begin
            if Assigned(OnSuccess) then
              OnSuccess(Response);
          end;
        401:
          begin
            ShowNotification('Session expired. Please log in again.', True);
            DoLogout;
          end;
        403:
          ShowNotification('You do not have permission for this action.', True);
        404:
          ShowNotification('The requested resource was not found.', True);
        422:
          begin
            var Msg := 'Validation error.';
            if Assigned(Response) then
              Msg := Response.GetValue<string>('message', Msg);
            ShowNotification(Msg, True);
          end;
      else
        ShowNotification('Server error: ' + StatusCode.ToString, True);
      end;
    finally
      Response.Free;
    end;
  except
    on E: ENetHTTPClientException do
```

```
      ShowNotification('Network error: Could not reach server.', True);
    on E: Exception do
      ShowNotification('Unexpected error: ' + E.Message, True);
  end;
end;
```

Usage:

```
procedure TFormMain.FetchStatsWithSafeCall;
begin
  SafeRESTCall('GET', '/stats', '', procedure(Response: TJSONObject)
  begin
    PopulateMemTableFromJSON(MemStats, 'stats', Response.ToString);
    var Page := Pages.Pages.FindPage('dashboard');
    if Assigned(Page) then
      Page.HTMLContent.Text := BuildStatsHTML;
  end);
end;
```

---

14. Using Twig Templates Instead of String Concatenation

The inline HTML strings above work but become unwieldy. For production applications, use Twig template files. Here is the dashboard page as a Twig template:

```
{# templates/dashboard.html #}
{% extends 'layout.html' %}

{% block title %}Dashboard{% endblock %}

{% block content %}
<div style="display: flex; gap: 20px; margin-bottom: 30px;">
  {% for stat in stats %}
  <div style="flex: 1; background: {{ stat.color }}; color: white;
    padding: 20px; border-radius: 8px;">
    <p style="font-size: 14px; margin: 0; opacity: 0.8;">{{ stat.label }}</p>
    <p style="font-size: 36px; margin: 5px 0 0 0; font-weight: bold;">
      {{ stat.value }}
    </p>
  </div>
  {% endfor %}
</div>

<div id="notificationArea" style="display: none;
  padding: 10px; border-radius: 4px; margin-bottom: 20px;">
</div>
{% endblock %}
```

```
{# templates/layout.html #}
<div style="font-family: Arial; padding: 20px;">
  <div style="display: flex; justify-content: space-between;
    align-items: center; margin-bottom: 30px; padding-bottom: 15px;
    border-bottom: 2px solid #ecf0f1;">
    <h1 style="margin: 0; color: #2c3e50;">{% block title %}{% endblock %}</h1>
    <div>
      <span onclick="App:NavigateTo('dashboard')"
        style="margin-right: 15px; color: #7f8c8d; cursor: pointer;">Dashboard</span>
      <span onclick="App:NavigateTo('users')"
```

```
        style="margin-right: 15px; color: #7f8c8d; cursor: pointer;">Users</span>
      <span onclick="App:NavigateTo('settings')"
        style="margin-right: 15px; color: #7f8c8d; cursor: pointer;">Settings</span>
      <span onclick="App:DoLogout"
        style="color: #e74c3c; cursor: pointer;">Logout</span>
    </div>
  </div>
  {% block content %}{% endblock %}
</div>
```

Use Twig pages in your TTina4HTMLPages:

```
procedure TFormMain.SetupDashboardPageWithTwig;
var
  Page: TTina4Page;
begin
  Page := Pages.Pages.Add;
  Page.PageName := 'dashboard';
  Page.TwigContent.LoadFromFile(
    TPath.Combine(Pages.TwigTemplatePath, 'dashboard.html'));
end;

procedure TFormMain.FetchStatsWithTwig;
begin
  RESTStats.OnExecuteDone := procedure(Sender: TObject)
  begin
    TThread.Synchronize(nil, procedure
    begin
      { Set Twig variables from MemTable data }
      Pages.SetTwigVariable('stats',
        '[' +
        '{"label": "Total Users", "value": "' +
          MemStats.FieldByName('user_count').AsString +
          '", "color": "#3498db"},' +
        '{"label": "Active Sessions", "value": "' +
          MemStats.FieldByName('active_sessions').AsString +
          '", "color": "#2ecc71"},' +
        '{"label": "Revenue", "value": "$' +
          MemStats.FieldByName('revenue').AsString +
          '", "color": "#9b59b6"},' +
        '{"label": "Orders", "value": "' +
          MemStats.FieldByName('order_count').AsString +
          '", "color": "#e67e22"}' +
        ']');

      if Pages.ActivePage = 'dashboard' then
        Pages.NavigateTo('dashboard');
    end);
  end;
  RESTStats.ExecuteRESTCallAsync;
end;
```

---

15. Project Organization

For a production application, organize your project like this:

```
AdminDashboard/
```

```
  AdminDashboard.dpr
  AdminDashboard.dproj
  src/
    MainUnit.pas              -- Main form, page setup, navigation
    MainUnit.fmx              -- FMX form with all components
    DataModule.pas            -- REST components, MemTables
    DataModule.dfm            -- DataModule design file
    Auth.pas                  -- Authentication logic
    UserOperations.pas        -- User CRUD operations
  templates/
    layout.html               -- Shared layout with navigation
    login.html                -- Login page
    dashboard.html            -- Dashboard with stat cards
    users.html                -- User list with table
    user-edit.html            -- User edit form
    settings.html             -- Settings form
  ssl/
    libeay32.dll              -- 32-bit SSL (for IDE)
    ssleay32.dll
    libcrypto-3-x64.dll       -- 64-bit SSL (for compiled app)
    libssl-3-x64.dll
```

Move REST components to a data module so the main form stays focused on UI:

```
// DataModule.pas
unit DataModule;

interface

uses
  System.SysUtils, System.Classes,
  FireDAC.Comp.Client, Data.DB,
  Tina4REST, Tina4RESTRequest;

type
  TDM = class(TDataModule)
    REST: TTina4REST;
    RESTUsers: TTina4RESTRequest;
    RESTStats: TTina4RESTRequest;
    MemUsers: TFDMemTable;
    MemStats: TFDMemTable;
    DSUsers: TDataSource;
    procedure DataModuleCreate(Sender: TObject);
  private
    FToken: string;
  public
    procedure SetToken(const AToken: string);
    property Token: string read FToken;
  end;

var
  DM: TDM;

implementation

{$R *.dfm}

procedure TDM.DataModuleCreate(Sender: TObject);
begin
```

```
  REST.BaseUrl := 'https://api.example.com/v1';
  RESTUsers.Tina4REST := REST;
  RESTUsers.EndPoint := '/users';
  RESTUsers.DataKey := 'records';
  RESTUsers.MemTable := MemUsers;
  RESTStats.Tina4REST := REST;
  RESTStats.EndPoint := '/stats';
  RESTStats.DataKey := 'stats';
  RESTStats.MemTable := MemStats;
  DSUsers.DataSet := MemUsers;
end;

procedure TDM.SetToken(const AToken: string);
begin
  FToken := AToken;
  REST.SetBearer(AToken);
end;

end.
```

Summary

This chapter built a complete admin dashboard using:

- **TTina4REST** for API authentication and HTTP calls
- **TTina4RESTRequest** for declarative data fetching into MemTables
- **TTina4HTMLRender** for rendering the entire UI as styled HTML
- **TTina4HTMLPages** for SPA-style page navigation
- **Twig templates** for maintainable, dynamic HTML
- **RTTI onclick** for bridging HTML events to Pascal methods
- **WebSocket** for real-time notifications
- **DOM manipulation** for in-place UI updates without full re-renders

The data flows in one direction: API response goes into a MemTable, MemTable data is rendered into HTML, user actions in HTML call Pascal methods via RTTI, and those methods make API calls that restart the cycle. This unidirectional flow makes the application predictable and debuggable.

Every pattern shown here -- authentication guards, async data loading, error handling, notification toasts -- is reusable in any Tina4 Delphi application. The admin dashboard is not just an example. It is a template for how to build desktop applications that talk to REST APIs.

Design Patterns & Best Practices

What We Learned the Hard Way

You call `REST.Get` and forget to free the response. The app leaks memory. You parse JSON in the main thread and the UI freezes for two seconds. You hardcode the API key in the source and push it to GitHub. You rebuild the entire HTML page when only one label needs updating.

Every pattern in this chapter was discovered through real-world mistakes. Every best practice exists because someone did it the wrong way first. This is the chapter you read before you ship -- and the chapter you return to when something breaks and you cannot figure out why.

---

1. Memory Management

Delphi does not have a garbage collector. Every object you create must be freed. The Tina4 components create `TJSONObject` instances that you own. If you do not free them, the memory leaks silently until the application crashes.

The try/finally Pattern

Every `TJSONObject` returned by `TTina4REST` must be freed:

```
// WRONG -- leaks memory if an exception occurs between Get and Free
var Response := REST.Get(StatusCode, '/users');
ShowMessage(Response.ToString);
Response.Free;

// RIGHT -- guaranteed cleanup
var Response := REST.Get(StatusCode, '/users');
try
  ShowMessage(Response.ToString);
finally
  Response.Free;
end;
```

This applies to every REST method: `Get`, `Post`, `Patch`, `Put`, `Delete`. They all return a `TJSONObject` that you own.

Parsing JSON Safely

`StrToJSONObject` returns `nil` on failure. Always check:

```
// WRONG -- access violation if JSON is malformed
var Obj := StrToJSONObject(SomeString);
ShowMessage(Obj.GetValue<string>('name'));
Obj.Free;

// RIGHT -- nil check before use
var Obj := StrToJSONObject(SomeString);
try
  if Assigned(Obj) then
    ShowMessage(Obj.GetValue<string>('name'))
  else
```

```
    ShowMessage('Invalid JSON');
finally
  Obj.Free;  // Free is safe to call on nil
end;
```

Component Ownership

Components dropped on a form at design time are owned by the form. The form frees them automatically. Components created at runtime need explicit ownership:

```
// Owned by Self (the form) -- freed automatically
var Adapter := TTina4JSONAdapter.Create(Self);
Adapter.MemTable := FDMemTable1;

// No owner -- you must free it yourself
var Twig := TTina4Twig.Create(nil);
try
  Twig.Render('template.html', Variables);
finally
  Twig.Free;
end;
```

Rule of thumb: if the component lives for the form's lifetime, pass `Self` as owner. If it is temporary, pass `nil` and use try/finally.

MemTable Lifecycle

`TFDMemTable` does not connect to a database, so it does not need explicit cleanup of connections. But be aware of when data is valid:

```
// WRONG -- MemTable might not be active
ShowMessage(MemUsers.FieldByName('name').AsString);

// RIGHT -- check Active and RecordCount
if MemUsers.Active and (MemUsers.RecordCount > 0) then
  ShowMessage(MemUsers.FieldByName('name').AsString)
else
  ShowMessage('No data loaded');
```

---

2. Async Patterns

The Problem with Synchronous Calls

`ExecuteRESTCall` blocks the main thread. The UI freezes until the response arrives. On a slow network, the user sees a frozen window and thinks the app crashed.

```
// WRONG -- blocks UI thread
RESTUsers.ExecuteRESTCall;  // 2-second freeze
UpdateUI;
```

The Async Solution

```
// RIGHT -- non-blocking
RESTUsers.OnExecuteDone := procedure(Sender: TObject)
begin
  TThread.Synchronize(nil, procedure
  begin
    UpdateUI;
  end);
end;
RESTUsers.ExecuteRESTCallAsync;
```

`ExecuteRESTCallAsync` runs the HTTP call on a background thread. When it completes, `OnExecuteDone` fires. But `OnExecuteDone` fires on the background thread, not the main thread. You must use `TThread.Synchronize` to update UI controls.

Loading Indicators

Show a loading state while the async call runs:

```
procedure TFormMain.FetchUsersAsync;
begin
  { Show loading }
  Renderer.SetElementText('userTableBody', 'Loading...');
  Renderer.SetElementVisible('loadingSpinner', True);

  RESTUsers.OnExecuteDone := procedure(Sender: TObject)
  begin
    TThread.Synchronize(nil, procedure
    begin
      { Hide loading, show data }
      Renderer.SetElementVisible('loadingSpinner', False);
      RefreshUserTable;
    end);
  end;
  RESTUsers.ExecuteRESTCallAsync;
end;
```

Multiple Concurrent Requests

When you need data from multiple endpoints simultaneously:

```
procedure TFormMain.LoadDashboard;
var
  StatsLoaded, UsersLoaded: Boolean;
begin
  StatsLoaded := False;
  UsersLoaded := False;

  Renderer.HTML.Text :=
    '<div style="padding: 20px;">' +
    '  <p id="loadStatus">Loading dashboard data...</p>' +
    '</div>';

  RESTStats.OnExecuteDone := procedure(Sender: TObject)
  begin
    TThread.Synchronize(nil, procedure
    begin
      StatsLoaded := True;
```

```
      if StatsLoaded and UsersLoaded then
        RenderFullDashboard;
    end);
  end;

  RESTUsers.OnExecuteDone := procedure(Sender: TObject)
  begin
    TThread.Synchronize(nil, procedure
    begin
      UsersLoaded := True;
      if StatsLoaded and UsersLoaded then
        RenderFullDashboard;
    end);
  end;

  RESTStats.ExecuteRESTCallAsync;
  RESTUsers.ExecuteRESTCallAsync;
end;
```

Both requests fire simultaneously. The dashboard renders when both complete. This is faster than sequential calls.

Cancellation and Timeouts

The underlying HTTP client supports timeouts. Set them on the TTina4REST component:

```
REST.Timeout := 10000;  // 10 seconds

RESTUsers.OnExecuteDone := procedure(Sender: TObject)
begin
  TThread.Synchronize(nil, procedure
  begin
    if RESTUsers.LastStatusCode = 0 then
      ShowNotification('Request timed out.', True)
    else
      RefreshUserTable;
  end);
end;
RESTUsers.ExecuteRESTCallAsync;
```

---

3. Error Handling Patterns

Status Code Checking

Every REST response has a status code. Check it before using the data:

```
procedure TFormMain.HandleResponse(StatusCode: Integer; Response: TJSONObject);
begin
  case StatusCode of
    200:
      ProcessData(Response);
    201:
      ShowNotification('Created successfully.');
    204:
      ShowNotification('Deleted successfully.');
    400:
      ShowNotification('Bad request: check your input.', True);
```

```
    401:
      begin
        ShowNotification('Session expired. Please log in again.', True);
        DoLogout;
      end;
    403:
      ShowNotification('Access denied.', True);
    404:
      ShowNotification('Resource not found.', True);
    422:
      begin
        { Extract validation errors from response }
        if Assigned(Response) then
        begin
          var Errors := Response.GetValue<TJSONArray>('errors');
          if Assigned(Errors) then
          begin
            var Msg := '';
            for var I := 0 to Errors.Count - 1 do
              Msg := Msg + Errors.Items[I].GetValue<string>('message') + #13#10;
            ShowNotification(Msg.Trim, True);
          end;
        end
        else
          ShowNotification('Validation failed.', True);
      end;
    500..599:
      ShowNotification('Server error. Please try again later.', True);
  else
    ShowNotification('Unexpected response: ' + StatusCode.ToString, True);
  end;
end;
```

JSON Parse Failure

Never assume JSON parsing will succeed:

```
// WRONG -- crashes on malformed JSON
var Users := Response.GetValue<TJSONArray>('records');
for var I := 0 to Users.Count - 1 do
  ProcessUser(Users.Items[I] as TJSONObject);

// RIGHT -- defensive parsing
var UsersValue := Response.FindValue('records');
if (UsersValue <> nil) and (UsersValue is TJSONArray) then
begin
  var Users := UsersValue as TJSONArray;
  for var I := 0 to Users.Count - 1 do
  begin
    if Users.Items[I] is TJSONObject then
      ProcessUser(Users.Items[I] as TJSONObject);
  end;
end
else
  ShowNotification('Unexpected response format.', True);
```

Network Error Handling

Wrap REST calls in exception handlers:

```
procedure TFormMain.SafeFetch(const EndPoint: string; OnDone: TProc<TJSONObject>);
var
  StatusCode: Integer;
  Response: TJSONObject;
begin
  try
    Response := REST.Get(StatusCode, EndPoint, '');
    try
      if StatusCode = 200 then
        OnDone(Response)
      else
        HandleResponse(StatusCode, Response);
    finally
      Response.Free;
    end;
  except
    on E: ENetHTTPClientException do
    begin
      if E.Message.Contains('SSL') then
        ShowNotification('SSL error: check your SSL DLL configuration.', True)
      else if E.Message.Contains('Timeout') then
        ShowNotification('Request timed out. Check your network.', True)
      else
        ShowNotification('Network error: ' + E.Message, True);
    end;
    on E: Exception do
      ShowNotification('Error: ' + E.Message, True);
  end;
end;
```

User-Friendly Error Messages in HTML

Display errors in the HTML renderer instead of using ShowMessage dialogs:

```
procedure TFormMain.ShowError(const Msg: string);
begin
  Renderer.SetElementVisible('errorBanner', True);
  Renderer.SetElementText('errorBanner', Msg);
  Renderer.SetElementStyle('errorBanner', 'background-color', '#e74c3c');
  Renderer.SetElementStyle('errorBanner', 'color', 'white');
  Renderer.SetElementStyle('errorBanner', 'padding', '12px');
  Renderer.SetElementStyle('errorBanner', 'border-radius', '4px');

  TTask.Run(procedure
  begin
    Sleep(5000);
    TThread.Synchronize(nil, procedure
    begin
      Renderer.SetElementVisible('errorBanner', False);
    end);
  end);
end;
```

---

4. Design-Time vs Runtime

What to Configure in Object Inspector

Set these properties at design time -- they rarely change:

Component	Property	Why
`TTina4REST`	`BaseUrl`	The API base URL is fixed per deployment
`TTina4RESTRequest`	`Tina4REST`	Links never change at runtime
`TTina4RESTRequest`	`DataKey`	JSON key is defined by the API
`TTina4RESTRequest`	`MemTable`	Target MemTable is fixed
`TTina4HTMLPages`	`Renderer`	Renderer link is permanent
`TTina4HTMLRender`	`CacheEnabled`	Caching policy is global

What to Configure in Code

Set these at runtime -- they depend on user actions or environment:

```
// Auth tokens change on login
REST.SetBearer(Token);

// Endpoints change based on context
RESTUsers.EndPoint := '/users/' + UserId + '/orders';

// Query params change with search/pagination
RESTUsers.QueryParams := 'page=2&search=admin';

// HTML content changes with data
Renderer.HTML.Text := BuildDynamicHTML;

// Twig variables change per render
Pages.SetTwigVariable('userName', CurrentUser.Name);
```

When to Create Components Dynamically

Create components at runtime when:

- The number of instances depends on data (e.g., one TTina4RESTRequest per API entity)
- The component is temporary (e.g., a one-time report renderer)
- You need a pool of workers

```
function TFormMain.CreateRESTRequest(const EndPoint, DataKey: string;
  AMemTable: TFDMemTable): TTina4RESTRequest;
begin
  Result := TTina4RESTRequest.Create(Self);
  Result.Tina4REST := REST;
  Result.EndPoint := EndPoint;
```

```
  Result.DataKey := DataKey;
  Result.MemTable := AMemTable;
  Result.RequestType := TTina4RequestType.Get;
  Result.SyncMode := TTina4RestSyncMode.Clear;
end;

// Usage
var ReqOrders := CreateRESTRequest('/orders', 'records', MemOrders);
try
  ReqOrders.ExecuteRESTCall;
  // process MemOrders
finally
  ReqOrders.Free;
end;
```

---

5. Data Flow Patterns

Unidirectional: API to MemTable to UI

The most common pattern. Data flows in one direction:

```
API Response --> PopulateMemTableFromJSON --> MemTable --> HTML Render
```

```
procedure TFormMain.LoadProducts;
begin
  RESTProducts.OnExecuteDone := procedure(Sender: TObject)
  begin
    TThread.Synchronize(nil, procedure
    begin
      { MemTable is populated automatically by RESTRequest }
      { Now render it }
      RenderProductTable;
    end);
  end;
  RESTProducts.ExecuteRESTCallAsync;
end;

procedure TFormMain.RenderProductTable;
var
  HTML: string;
begin
  HTML := '<table>';
  MemProducts.First;
  while not MemProducts.Eof do
  begin
    HTML := HTML +
      '<tr>' +
      '  <td>' + MemProducts.FieldByName('name').AsString + '</td>' +
      '  <td>' + MemProducts.FieldByName('price').AsString + '</td>' +
      '</tr>';
    MemProducts.Next;
  end;
  HTML := HTML + '</table>';
  Renderer.HTML.Text := HTML;
end;
```

Bidirectional: UI to API to MemTable to UI

User fills a form, submits it, the API processes it, the response refreshes the data:

```
HTML Form --> onclick/OnFormSubmit --> POST to API --> Refresh MemTable --> Re-render
```

```
procedure TFormMain.HandleFormSubmit(Sender: TObject;
  const FormName: string; FormData: TStrings);
var
  StatusCode: Integer;
  Body: TJSONObject;
  Response: TJSONObject;
begin
  if FormName = 'productForm' then
  begin
    Body := TJSONObject.Create;
    try
      Body.AddPair('name', FormData.Values['name']);
      Body.AddPair('price', FormData.Values['price']);

      Response := REST.Post(StatusCode, '/products', '', Body.ToString);
      try
        if StatusCode = 201 then
        begin
          { Refresh the list to include the new product }
          RESTProducts.ExecuteRESTCall;
          RenderProductTable;
          ShowNotification('Product created.');
        end
        else
          ShowError('Failed to create product.');
      finally
        Response.Free;
      end;
    finally
      Body.Free;
    end;
  end;
end;
```

Event-Driven: WebSocket to UI

Real-time updates bypass the request/response cycle:

```
WebSocket Message --> Parse JSON --> Update MemTable or UI directly
```

```
FWebSocket.OnMessage := procedure(Sender: TObject; const Msg: string)
begin
  TThread.Synchronize(nil, procedure
  var
    JSON: TJSONObject;
  begin
    JSON := StrToJSONObject(Msg);
    try
      if Assigned(JSON) then
      begin
        var EventType := JSON.GetValue<string>('type', '');

        if EventType = 'user.created' then
        begin
```

```
          { Add to MemTable without full refresh }
          MemUsers.Append;
          MemUsers.FieldByName('id').AsString :=
            JSON.GetValue<string>('data.id');
          MemUsers.FieldByName('name').AsString :=
            JSON.GetValue<string>('data.name');
          MemUsers.Post;
          RenderUserTable;
        end
        else if EventType = 'stats.updated' then
        begin
          { Refresh stats }
          FetchStats;
        end;
      end;
    finally
      JSON.Free;
    end;
  end);
end;
```

---

6. Performance

MemTable SyncMode for Incremental Updates

When polling an API for updates, use `Sync` mode instead of `Clear`:

```
// WRONG for polling -- clears and rebuilds the entire table
RESTUsers.SyncMode := TTina4RestSyncMode.Clear;

// RIGHT for polling -- updates existing rows, adds new ones
RESTUsers.SyncMode := TTina4RestSyncMode.Sync;
RESTUsers.IndexFieldNames := 'id';
```

In `Sync` mode, the MemTable matches rows by `IndexFieldNames`. Existing rows are updated in place. New rows are inserted. Bound controls (grids, lists) update smoothly without flickering.

Minimizing HTML Re-renders

Do not rebuild the entire HTML when only one element changes:

```
// WRONG -- rebuilds entire page, resets scroll position, flickers
Renderer.HTML.Text := BuildFullPageHTML;

// RIGHT -- update just the element that changed
Renderer.SetElementText('userCount', NewCount.ToString);
Renderer.SetElementStyle('statusDot', 'background-color', '#2ecc71');
Renderer.SetElementVisible('loadingSpinner', False);
```

Use `SetElementText`, `SetElementStyle`, `SetElementVisible`, and `SetElementValue` for surgical updates. Full re-renders should only happen when the page structure changes.

Image Caching

Enable disk-based caching for images loaded in TTina4HTMLRender:

```
Renderer.CacheEnabled := True;
```

```
Renderer.CacheDir := TPath.Combine(TPath.GetDocumentsPath, 'MyAppCache');
```

Without caching, every page render re-downloads all images. With caching, images are loaded from disk after the first download. This makes page transitions near-instant.

Lazy Loading with Pagination

Do not load all records at once. Use pagination:

```
procedure TFormMain.FetchPage(APage: Integer);
begin
  RESTUsers.QueryParams := Format('page=%d&limit=20', [APage]);
  RESTUsers.OnExecuteDone := procedure(Sender: TObject)
  begin
    TThread.Synchronize(nil, procedure
    begin
      RenderUserTable;
      RenderPaginationControls(APage);
    end);
  end;
  RESTUsers.ExecuteRESTCallAsync;
end;
```

Twenty records per page. The user clicks Next to load more. The API does the heavy lifting.

---

7. Security

Never Hardcode Credentials

```
// WRONG -- credentials in source code
REST.Username := 'admin';
REST.Password := 'P@ssw0rd123';

// RIGHT -- read from config file or environment
var Config := TIniFile.Create(TPath.Combine(
  TPath.GetDocumentsPath, 'myapp.ini'));
try
  REST.BaseUrl := Config.ReadString('API', 'BaseUrl', '');
  // Do not store credentials in config either
  // Use login flow with bearer tokens instead
finally
  Config.Free;
end;
```

Token Refresh Pattern

Bearer tokens expire. Handle 401 responses with a refresh flow:

```
procedure TFormMain.ExecuteWithRefresh(const EndPoint: string;
  OnSuccess: TProc<TJSONObject>);
var
  StatusCode: Integer;
  Response: TJSONObject;
begin
  Response := REST.Get(StatusCode, EndPoint, '');
  try
    if StatusCode = 401 then
```

```
    begin
      { Try to refresh the token }
      var RefreshResponse := REST.Post(StatusCode, '/auth/refresh', '',
        '{"refresh_token": "' + FRefreshToken + '"}');
      try
        if StatusCode = 200 then
        begin
          FBearerToken := RefreshResponse.GetValue<string>('token');
          FRefreshToken := RefreshResponse.GetValue<string>('refresh_token');
          REST.SetBearer(FBearerToken);

          { Retry the original request }
          Response.Free;
          Response := REST.Get(StatusCode, EndPoint, '');
          if StatusCode = 200 then
            OnSuccess(Response)
          else
            DoLogout;
        end
        else
          DoLogout;
      finally
        RefreshResponse.Free;
      end;
    end
    else if StatusCode = 200 then
      OnSuccess(Response)
    else
      HandleResponse(StatusCode, Response);
  finally
    Response.Free;
  end;
end;
```

HTTPS Only

Never send credentials or tokens over HTTP. Always use HTTPS:

```
// WRONG
REST.BaseUrl := 'http://api.example.com';

// RIGHT
REST.BaseUrl := 'https://api.example.com';
```

If your development server does not have SSL, use a self-signed certificate and configure the HTTP client to accept it during development only.

---

8. Project Organization

Separate Data Modules

Keep REST components and MemTables in a data module, not on the main form:

```
// DataModule.pas
unit DataModule;

interface
```

```
uses
  System.Classes, FireDAC.Comp.Client, Data.DB,
  Tina4REST, Tina4RESTRequest;

type
  TDM = class(TDataModule)
    REST: TTina4REST;
    RESTUsers: TTina4RESTRequest;
    RESTProducts: TTina4RESTRequest;
    MemUsers: TFDMemTable;
    MemProducts: TFDMemTable;
  end;

var
  DM: TDM;

// In MainUnit:
uses DataModule;

procedure TFormMain.LoadUsers;
begin
  DM.RESTUsers.ExecuteRESTCall;
  RenderUserTable(DM.MemUsers);
end;
```

Template File Organization

Store Twig templates in a structured directory:

```
templates/
  layouts/
    base.html              -- Base layout with <head> and structure
    sidebar.html           -- Sidebar navigation component
  pages/
    dashboard.html         -- Dashboard page
    users/
      list.html            -- User list
      detail.html          -- User detail card
      form.html            -- User create/edit form
  components/
    stat-card.html         -- Reusable stat card
    data-table.html        -- Reusable data table
    notification.html      -- Notification toast
```

Resource Management

For deployed applications, embed templates as resources or deploy them alongside the executable:

```
procedure TFormMain.FormCreate(Sender: TObject);
begin
  { Templates are relative to the executable }
  Pages.TwigTemplatePath := TPath.Combine(
    ExtractFilePath(ParamStr(0)), 'templates');

  { Or use a configurable path }
  var ConfigPath := TPath.Combine(TPath.GetDocumentsPath, 'MyApp');
  ForceDirectories(ConfigPath);
end;
```

---

9. Complete Example: Refactoring a Poorly Written App

Here is a bad implementation. Read it and find the problems:

```
// BAD CODE -- how many issues can you spot?
procedure TForm1.Button1Click(Sender: TObject);
var
  Resp: TJSONObject;
  SC: Integer;
begin
  Tina4REST1.BaseUrl := 'http://api.example.com';
  Tina4REST1.Username := 'admin';
  Tina4REST1.Password := 'admin123';

  Resp := Tina4REST1.Get(SC, '/users');
  var Users := Resp.GetValue<TJSONArray>('records');
  var HTML := '<table>';
  for var I := 0 to Users.Count - 1 do
  begin
    var U := Users.Items[I] as TJSONObject;
    HTML := HTML + '<tr><td>' + U.GetValue<string>('name') + '</td></tr>';
  end;
  HTML := HTML + '</table>';
  Tina4HTMLRender1.HTML.Text := HTML;
end;
```

Problems found:

- **HTTP, not HTTPS** -- credentials sent in plain text
- **Hardcoded credentials** -- username and password in source code
- **BaseUrl set in click handler** -- should be set once in FormCreate
- **No try/finally** -- `Resp` is never freed, memory leak
- **No nil check on Resp** -- crashes if request fails
- **No status code check** -- processes data even on 404 or 500
- **No exception handling** -- network errors crash the app
- **Synchronous call in UI thread** -- freezes the interface
- **No type checking on JSON values** -- crashes on unexpected structure
- **Component names are defaults** -- `Button1`, `Tina4REST1` tell you nothing

Here is the refactored version:

```
unit UserListUnit;

interface

uses
  System.SysUtils, System.Classes, System.JSON,
  FMX.Types, FMX.Controls, FMX.Forms, FMX.StdCtrls,
  Tina4REST, Tina4RESTRequest, Tina4HTMLRender,
  FireDAC.Comp.Client;
```

```
type
  TFormUserList = class(TForm)
    REST: TTina4REST;
    RESTUsers: TTina4RESTRequest;
    MemUsers: TFDMemTable;
    Renderer: TTina4HTMLRender;
    ButtonLoad: TButton;
    procedure FormCreate(Sender: TObject);
    procedure ButtonLoadClick(Sender: TObject);
  private
    procedure RenderUserTable;
    procedure ShowError(const Msg: string);
  end;

implementation

{$R *.fmx}

procedure TFormUserList.FormCreate(Sender: TObject);
begin
  REST.BaseUrl := 'https://api.example.com';
  { Token set after login -- never hardcode credentials }

  RESTUsers.Tina4REST := REST;
  RESTUsers.EndPoint := '/users';
  RESTUsers.DataKey := 'records';
  RESTUsers.MemTable := MemUsers;
end;

procedure TFormUserList.ButtonLoadClick(Sender: TObject);
begin
  ButtonLoad.Enabled := False;
  Renderer.HTML.Text := '<p>Loading...</p>';

  RESTUsers.OnExecuteDone := procedure(Sender: TObject)
  begin
    TThread.Synchronize(nil, procedure
    begin
      ButtonLoad.Enabled := True;

      if RESTUsers.LastStatusCode = 200 then
        RenderUserTable
      else if RESTUsers.LastStatusCode = 401 then
        ShowError('Please log in first.')
      else
        ShowError('Failed to load users.');
    end);
  end;
  RESTUsers.ExecuteRESTCallAsync;
end;

procedure TFormUserList.RenderUserTable;
var
  HTML: string;
begin
  HTML := '<table style="width: 100%; border-collapse: collapse;">';
  HTML := HTML + '<tr style="background: #2c3e50; color: white;">' +
    '<th style="padding: 10px;">Name</th></tr>';
```

```
  if MemUsers.Active and (MemUsers.RecordCount > 0) then
  begin
    MemUsers.First;
    while not MemUsers.Eof do
    begin
      HTML := HTML + '<tr><td style="padding: 10px; border-bottom: 1px solid #eee;">' +
        MemUsers.FieldByName('name').AsString + '</td></tr>';
      MemUsers.Next;
    end;
  end
  else
    HTML := HTML + '<tr><td style="padding: 20px; text-align: center; ' +
      'color: #999;">No users found.</td></tr>';

  HTML := HTML + '</table>';
  Renderer.HTML.Text := HTML;
end;

procedure TFormUserList.ShowError(const Msg: string);
begin
  Renderer.HTML.Text :=
    '<div style="padding: 15px; background: #e74c3c; color: white; ' +
    '  border-radius: 4px;">' + Msg + '</div>';
end;

end.
```

What changed:

- HTTPS, not HTTP
- No hardcoded credentials -- token set after login
- BaseUrl set in FormCreate, not in click handler
- TTina4RESTRequest handles the REST call and MemTable population
- Async execution with `ExecuteRESTCallAsync`
- Status code checking
- Loading indicator while data fetches
- Button disabled during load to prevent double-clicks
- Meaningful component and method names
- MemTable Active and RecordCount checked before iteration
- Error messages displayed in the renderer, not as dialogs

---

10. Exercise: Code Review

Review the following code and identify five issues. Then fix each one.

```
procedure TForm1.LoadOrders;
var
  R: TJSONObject;
  S: Integer;
```

```
begin
  R := Tina4REST1.Get(S, '/orders');
  var Data := R.GetValue<TJSONArray>('orders');
  PopulateMemTableFromJSON(FDMemTable1, 'orders', R.ToString);

  var Total := 0.0;
  FDMemTable1.First;
  while not FDMemTable1.Eof do
  begin
    Total := Total + FDMemTable1.FieldByName('amount').AsFloat;
    FDMemTable1.Next;
  end;

  Tina4HTMLRender1.HTML.Text :=
    '<h1>Orders</h1><p>Total: $' + FloatToStr(Total) + '</p>';
end;
```

Solution

Issue 1: `R` (TJSONObject) is never freed -- memory leak.

Issue 2: No `try/except` -- network errors crash the app.

Issue 3: No status code check -- processes a 500 error response as valid data.

Issue 4: `FloatToStr` uses locale-specific formatting -- on some systems `1234.5` becomes `1234,5`.

Issue 5: Synchronous call blocks the UI thread.

Fixed version:

```
procedure TFormOrders.LoadOrders;
begin
  Renderer.HTML.Text := '<p>Loading orders...</p>';

  RESTOrders.OnExecuteDone := procedure(Sender: TObject)
  begin
    TThread.Synchronize(nil, procedure
    begin
      if RESTOrders.LastStatusCode <> 200 then
      begin
        Renderer.HTML.Text :=
          '<div style="color: red;">Failed to load orders.</div>';
        Exit;
      end;

      var Total := 0.0;
      if MemOrders.Active then
      begin
        MemOrders.First;
        while not MemOrders.Eof do
        begin
          Total := Total + MemOrders.FieldByName('amount').AsFloat;
          MemOrders.Next;
        end;
      end;

      Renderer.HTML.Text := _____
```

```
        '<h1>Orders</h1><p>Total: $' +
        FormatFloat('#,##0.00', Total) + '</p>';
    end);
  end;
  RESTOrders.ExecuteRESTCallAsync;
end;
```

The TTina4RESTRequest handles JSON parsing and MemTable population. `FormatFloat` gives consistent formatting. The async call keeps the UI responsive. The status code is checked before processing data.

Summary

The patterns in this chapter come down to four principles:

- **Free what you create.** Every `TJSONObject` in a try/finally. Every dynamic component with an owner or explicit Free.
- **Never block the main thread.** Use `ExecuteRESTCallAsync` with `TThread.Synchronize` for UI updates.
- **Check before you use.** Status codes before processing responses. `Assigned()` before accessing objects. `Active` and `RecordCount` before iterating MemTables.
- **Update surgically.** Use `SetElementText` and `SetElementVisible` instead of rebuilding the entire HTML. Use `Sync` mode instead of `Clear` when polling.

These four principles prevent the five most common bugs in Tina4 Delphi applications: memory leaks, frozen UIs, access violations, missing error messages, and flickering displays. Follow them, and the bugs stop before they start.

Troubleshooting

When Things Go Wrong

The app compiles. It launches. You click the button to fetch users. Nothing happens. No error message. No crash. Just silence. The UI sits there. You check the endpoint -- it is correct. You check the auth -- it is set. You add a `ShowMessage` after the REST call and discover the status code is 0. Zero. The request never completed. You look at the SSL DLLs and realize you put 32-bit DLLs in System32 and 64-bit DLLs in SysWOW64. Backwards.

Every problem in this chapter has been hit by a real developer. Every fix has been verified. The format is the same throughout: Problem, Cause, Fix. Find your symptom, read the cause, apply the fix.

---

1. SSL Errors

"Could not load SSL library"

Problem: REST calls fail with the error "Could not load SSL library" or the HTTP client raises an `ENetHTTPClientException` mentioning SSL.

Cause: The OpenSSL DLLs are missing or have the wrong bitness. Delphi's HTTP client needs platform-matched DLLs:

- The IDE is 32-bit, so it needs 32-bit DLLs
- Your compiled 64-bit app needs 64-bit DLLs
- Windows has confusing folder names: `SysWOW64` is for 32-bit, `System32` is for 64-bit

Fix:

```
32-bit DLLs (libeay32.dll, ssleay32.dll)
  --> C:\Windows\SysWOW64\         (for the IDE, design-time testing)
  --> Your app's output directory   (if compiling as Win32)

64-bit DLLs (libcrypto-3-x64.dll, libssl-3-x64.dll)
  --> C:\Windows\System32\         (for compiled 64-bit apps)
  --> Your app's output directory   (alternative)
```

Verify you have the correct files:

```
// Add this to FormCreate to diagnose SSL at startup
procedure TFormMain.CheckSSL;
begin
  {$IFDEF WIN64}
  if not FileExists('libcrypto-3-x64.dll') and
     not FileExists('C:\Windows\System32\libcrypto-3-x64.dll') then
    ShowMessage('WARNING: 64-bit SSL DLLs not found');
  {$ENDIF}
  {$IFDEF WIN32}
  if not FileExists('libeay32.dll') and
     not FileExists('C:\Windows\SysWOW64\libeay32.dll') then
```

```
    ShowMessage('WARNING: 32-bit SSL DLLs not found');
  {$ENDIF}
end;
```

Certificate Verification Failures

Problem: HTTPS calls fail with "certificate verify failed" or "unable to get local issuer certificate."

Cause: The server's SSL certificate cannot be verified against a trusted CA bundle. Common with internal APIs, self-signed certificates, or corporate proxies.

Fix for development -- disable certificate validation (never do this in production):

```
uses
  System.Net.HttpClient;

procedure TFormMain.FormCreate(Sender: TObject);
begin
  { Development only -- accepts any certificate }
  REST.ValidateServerCertificate := False;
end;
```

Fix for production -- ensure the server has a valid certificate from a trusted CA (Let's Encrypt, DigiCert, etc.).

Self-Signed Certificate Handling

Problem: You need to connect to an internal API that uses a self-signed certificate.

Cause: The HTTP client rejects certificates not signed by a trusted CA.

Fix: Accept the specific certificate by validating its thumbprint:

```
procedure TFormMain.OnValidateServerCertificate(
  const Sender: TObject;
  const ARequest: TURLRequest;
  const Certificate: TCertificate;
  var Accepted: Boolean);
begin
  { Accept only your known self-signed certificate }
  if Certificate.Subject.Contains('CN=myapi.internal') then
    Accepted := True
  else
    Accepted := Certificate.IsValid;
end;
```

---

2. REST Issues

401 Unauthorized

Problem: Every REST call returns status 401.

Cause 1: Bearer token expired.

```
// Check if token is set
if REST.GetBearer.IsEmpty then
  ShowMessage('No bearer token set -- user needs to log in');
```

Cause 2: Wrong authentication type. The API expects Basic Auth but you are sending Bearer, or vice versa.

```
// Basic Auth
REST.Username := 'admin';
REST.Password := 'secret';

// Bearer Auth -- do NOT set Username/Password
REST.SetBearer('eyJhbGciOiJIUzI1NiJ9...');

// Both set? Bearer takes priority, but the server might reject
// the extra Authorization header. Clear one:
REST.Username := '';
REST.Password := '';
REST.SetBearer(Token);
```

Cause 3: Token format wrong. Some APIs want `Bearer <token>`, others want just the token. TTina4REST adds the `Bearer` prefix automatically via `SetBearer`. Do not add it yourself:

```
// WRONG -- double prefix: "Bearer Bearer eyJ..."
REST.SetBearer('Bearer eyJhbGciOiJIUzI1NiJ9...');

// RIGHT
REST.SetBearer('eyJhbGciOiJIUzI1NiJ9...');
```

404 Not Found

Problem: Status code 404 on a valid endpoint.

Cause 1: Trailing slash mismatch. The API expects `/users` but you send `/users/`.

```
// Try both
RESTUsers.EndPoint := '/users';   // no trailing slash
RESTUsers.EndPoint := '/users/';  // with trailing slash
```

Cause 2: BaseUrl includes a path that is duplicated in the EndPoint.

```
// WRONG -- requests /v1/v1/users
REST.BaseUrl := 'https://api.example.com/v1';
RESTUsers.EndPoint := '/v1/users';

// RIGHT
REST.BaseUrl := 'https://api.example.com/v1';
RESTUsers.EndPoint := '/users';
```

500 Internal Server Error

Problem: Server returns 500 on POST/PATCH requests.

Cause: The request body is malformed or missing required fields.

Fix: Log the request body before sending:

```
var Body := '{"name": "Andre"}';
// Log it
Memo1.Lines.Add('Sending: ' + Body);

var Response := REST.Post(StatusCode, '/users', '', Body);
```

Common body issues:

- Missing `Content-Type: application/json` header (TTina4REST sets this automatically)
- Unescaped quotes in string values
- Wrong field names (camelCase vs snake_case)

Timeout Errors

Problem: REST calls hang for 30+ seconds, then fail.

Cause: Default timeout is too long, or the server is unresponsive.

Fix:

```
// Set a reasonable timeout (milliseconds)
REST.Timeout := 10000;  // 10 seconds

// Handle timeout in async calls
RESTUsers.OnExecuteDone := procedure(Sender: TObject)
begin
  TThread.Synchronize(nil, procedure
  begin
    if RESTUsers.LastStatusCode = 0 then
      ShowMessage('Request timed out or network error')
    else
      ProcessResponse;
  end);
end;
```

CORS Issues When Calling Web APIs

Problem: Browser-based APIs return CORS errors. Your Delphi app gets unexpected responses.

Cause: CORS is a browser security feature. Desktop applications do not have CORS restrictions. If you are getting errors, the issue is something else (wrong URL, authentication, SSL).

Fix: CORS is not relevant for Delphi desktop apps. Check the actual error message -- it is likely a network, SSL, or auth issue disguised by a generic error message.

---

3. JSON Issues

Access Violation Parsing Malformed JSON

Problem: `StrToJSONObject` crashes with an access violation.

Cause: The input string is not valid JSON, and you are using the result without checking for nil.

Fix:

```
// WRONG
var Obj := StrToJSONObject(ResponseText);
ShowMessage(Obj.GetValue<string>('name'));  // AV if Obj is nil

// RIGHT
var Obj := StrToJSONObject(ResponseText);
try
```

```
  if Assigned(Obj) then
    ShowMessage(Obj.GetValue<string>('name'))
  else
    ShowMessage('Failed to parse JSON: ' + ResponseText.Substring(0, 100));
finally
  Obj.Free;
end;
```

Field Names Not Matching

Problem: `FieldByName('firstName')` raises "Field not found" but the data is there.

Cause: `PopulateMemTableFromJSON` and `GetFieldDefsFromJSONObject` convert camelCase to snake_case when the `ASnakeCase` parameter is `True` (which is the default in some contexts).

Fix: Use the correct field name:

```
// If JSON has "firstName" and snake_case conversion is on:
MemTable.FieldByName('first_name').AsString;  // RIGHT
MemTable.FieldByName('firstName').AsString;    // WRONG -- raises exception

// Check what fields exist:
for var I := 0 to MemTable.FieldCount - 1 do
  Memo1.Lines.Add(MemTable.Fields[I].FieldName);
```

Nested Objects Becoming ftMemo

Problem: A nested JSON object like `{"address": {"city": "Cape Town"}}` becomes a single `ftMemo` field instead of separate fields.

Cause: `GetFieldDefsFromJSONObject` stores nested objects and arrays as `ftMemo` fields containing the JSON string. This is by design -- MemTables do not support nested structures.

Fix: Parse the nested JSON from the memo field:

```
var AddressJSON := MemTable.FieldByName('address').AsString;
var Addr := StrToJSONObject(AddressJSON);
try
  if Assigned(Addr) then
    ShowMessage('City: ' + Addr.GetValue<string>('city'));
finally
  Addr.Free;
end;
```

Or use a separate TTina4JSONAdapter to extract the nested data into its own MemTable.

Large JSON Causing Out of Memory

Problem: Parsing a very large JSON response (100MB+) causes the app to run out of memory.

Cause: Delphi's `TJSONObject` loads the entire document into memory. Combined with string copies during parsing, memory usage can be 3-5x the JSON size.

Fix: Use pagination to limit response size:

```
RESTUsers.QueryParams := 'page=1&limit=100';  // Fetch 100 at a time
```

If the API does not support pagination, process the response in chunks or use streaming JSON parsers.

---

4. MemTable Issues

"Field not found"

Problem: `MemTable.FieldByName('user_name')` raises "Field 'user_name' not found."

Cause 1: The API changed its response format and the field name is different.

Fix: List all fields to see what actually exists:

```
procedure TFormMain.DebugMemTableFields(ATable: TFDMemTable);
begin
  if not ATable.Active then
  begin
    ShowMessage('MemTable is not active');
    Exit;
  end;

  var Fields := '';
  for var I := 0 to ATable.FieldCount - 1 do
    Fields := Fields + ATable.Fields[I].FieldName + ' (' +
      ATable.Fields[I].ClassName + ')' + #13#10;
  ShowMessage(Fields);
end;
```

Cause 2: The MemTable was never populated -- `ExecuteRESTCall` failed silently.

Fix: Check `LastStatusCode` before accessing MemTable data:

```
RESTUsers.ExecuteRESTCall;
if RESTUsers.LastStatusCode = 200 then
begin
  // Safe to access MemTable
  MemUsers.First;
end
else
  ShowMessage('REST call failed: ' + RESTUsers.LastStatusCode.ToString);
```

Duplicate Key Violations in Sync Mode

Problem: `PopulateMemTableFromJSON` raises an index violation error when using Sync mode.

Cause: `IndexFieldNames` is set to a field that has duplicate values in the data, or the index was not created properly.

Fix:

```
// Make sure the index field is unique in the data
RESTUsers.SyncMode := TTina4RestSyncMode.Sync;
RESTUsers.IndexFieldNames := 'id';  // 'id' must be unique

// If you need a compound key:
RESTUsers.IndexFieldNames := 'user_id;order_id';
```

IndexFieldNames Not Set for Sync Mode

Problem: Sync mode does not update existing records -- it keeps appending duplicates.

Cause: `IndexFieldNames` is empty. Without it, the MemTable cannot match existing rows.

Fix:

```
// WRONG -- Sync mode without index just appends
RESTUsers.SyncMode := TTina4RestSyncMode.Sync;
// IndexFieldNames is empty

// RIGHT
RESTUsers.SyncMode := TTina4RestSyncMode.Sync;
RESTUsers.IndexFieldNames := 'id';
```

Data Type Mismatches

Problem: `FieldByName('price').AsFloat` returns 0 even though the JSON has `"price": "29.99"`.

Cause: The JSON value is a string `"29.99"`, not a number `29.99`. The field was created as `ftString`.

Fix: Convert explicitly:

```
var PriceStr := MemTable.FieldByName('price').AsString;
var Price := StrToFloatDef(PriceStr, 0.0);
```

Or use `AsFloat` which does automatic conversion for most cases, but verify the field type:

```
// Debug the field type
ShowMessage(MemTable.FieldByName('price').DataType.ToString);
// If it shows ftString, the JSON sent "29.99" as a string
```

---

5. HTML Renderer Issues

Elements Not Displaying

Problem: You set `HTML.Text` but nothing appears in the renderer.

Cause 1: The renderer has zero width or height.

```
// Check dimensions
ShowMessage(Format('Renderer size: %dx%d',
  [Round(Renderer.Width), Round(Renderer.Height)]));
```

Cause 2: The HTML has a syntax error that prevents rendering.

Fix: Start with minimal HTML and build up:

```
// Test with minimal HTML first
Renderer.HTML.Text := '<p>Test</p>';
// If this works, add more content gradually
```

CSS Not Applying

Problem: CSS classes or styles are ignored.

Cause: The renderer supports a subset of CSS. Some properties are not implemented.

Fix: Check the supported CSS list in the documentation. Use inline styles as a fallback:

```
// If a CSS class does not work:
'<div class="my-custom-class">Text</div>'  // might not work

// Use inline styles instead:
'<div style="color: red; font-size: 18px;">Text</div>'  // works
```

Supported CSS includes: `color`, `background-color`, `font-size`, `font-family`, `font-weight`, `padding`, `margin`, `border`, `border-radius`, `width`, `height`, `display`, `text-align`, and more. Complex selectors like `:nth-child` or `@media` queries are not supported.

Form Controls Not Appearing

Problem: `<input>` or `<select>` elements are not visible.

Cause: The form control type is not supported, or the element is hidden by CSS.

Fix: Check supported form elements:

```
// Supported input types:
'<input type="text">'       // works
'<input type="password">'   // works
'<input type="email">'      // works
'<input type="checkbox">'   // works
'<input type="radio">'      // works
'<input type="submit">'     // works
'<input type="button">'     // works
'<input type="file">'       // works
'<input type="date">'       // might not render as date picker
'<input type="range">'      // might not render as slider
'<textarea>'                // works
'<select><option>'          // works
```

onclick Not Firing

Problem: Clicking an element with `onclick="App:MyMethod('test')"` does nothing.

Cause 1: The object was not registered with `RegisterObject`.

```
// Must be called before setting HTML
Renderer.RegisterObject('App', Self);
```

Cause 2: The method does not exist or has the wrong signature.

```
// The method must be published or public
// Parameter types must match what the onclick passes

// WRONG -- private method, not found by RTTI
private
  procedure MyMethod(const Value: string);

// RIGHT -- public method
public
  procedure MyMethod(Value: String);
```

Cause 3: Wrong quote escaping in the onclick attribute.

```
// WRONG -- mismatched quotes
'<span onclick="App:MyMethod("test")">Click</span>'

// RIGHT -- use single quotes inside double, or escape
'<span onclick="App:MyMethod(''test'')">Click</span>'
```

Cause 4: The element is overlapped by another element. The click hits the wrong element.

Fix: Add a distinct `style="cursor: pointer; z-index: 10;"` and ensure nothing overlaps.

Images Not Loading

Problem: `<img src="https://...">` shows a broken image or empty space.

Cause 1: Cache directory not set or not writable.

```
Renderer.CacheEnabled := True;
Renderer.CacheDir := TPath.Combine(TPath.GetDocumentsPath, 'MyAppCache');
ForceDirectories(Renderer.CacheDir);  // Create if it does not exist
```

Cause 2: HTTPS image URL but SSL DLLs not installed.

Fix: Install SSL DLLs as described in Section 1.

Cause 3: The image URL returns a redirect that the renderer does not follow.

Fix: Use the final URL directly, or download the image separately and use a `data:` URI:

```
var B64 := FileToBase64('downloaded_image.jpg');
Renderer.HTML.Text := '<img src="' + B64 + '">';
```

---

6. Page Navigation Issues

Default Page Not Showing

Problem: The app starts with a blank renderer. No page is displayed.

Cause: No page has `IsDefault := True`.

Fix:

```
var LoginPage := Pages.Pages.Add;
LoginPage.PageName := 'login';
LoginPage.IsDefault := True;  // This page shows on startup
LoginPage.HTMLContent.Text := '<h1>Login</h1>';
```

Links Not Navigating

Problem: Clicking `<a href="#dashboard">` does not change the page.

Cause 1: The `href` value does not match any `PageName`.

```
// Link href:     #dashboard
// Page.PageName: Dashboard   <-- case mismatch

// Fix: make them match exactly
Page.PageName := 'dashboard';  // lowercase
```

```
// HTML: <a href="#dashboard">  // matches
```

Cause 2: The renderer is not linked to the Pages component.

```
Pages.Renderer := Renderer;  // Must be set
```

OnBeforeNavigate Not Cancelling

Problem: Setting `Allow := False` in `OnBeforeNavigate` does not prevent navigation.

Cause: The `Allow` parameter is a `var` parameter. Make sure your event handler signature matches:

```
// WRONG -- parameter not passed by reference
procedure TFormMain.BeforeNav(Sender: TObject;
  const FromPage, ToPage: string; Allow: Boolean);  // missing 'var'

// RIGHT
procedure TFormMain.BeforeNav(Sender: TObject;
  const FromPage, ToPage: string; var Allow: Boolean);
begin
  if (ToPage <> 'login') and (not IsAuthenticated) then
    Allow := False;  // Prevents navigation
end;
```

---

7. Twig Template Issues

Template Not Rendering

Problem: Setting `Twig.Text` produces empty output or the raw template text.

Cause 1: Wrong template path for file-based templates.

```
// Check the path exists
ShowMessage(Pages.TwigTemplatePath);
ShowMessage(BoolToStr(DirectoryExists(Pages.TwigTemplatePath), True));
```

Cause 2: Template syntax error. A missing `{% endif %}` or unmatched braces.

Fix: Test with a minimal template:

```
Renderer.Twig.Text := '<p>{{ name }}</p>';
// If this works, the issue is in your complex template
```

Variables Empty

Problem: Template renders but variables show as empty.

Cause: Variables were not set before the template was assigned.

```
// WRONG -- template renders before variables are set
Renderer.Twig.Text := '<h1>{{ title }}</h1>';
Renderer.SetTwigVariable('title', 'Hello');

// RIGHT -- set variables first
Renderer.SetTwigVariable('title', 'Hello');
Renderer.Twig.Text := '<h1>{{ title }}</h1>';
```

Includes Failing

Problem: `{% include 'header.html' %}` does not work. The template renders without the included content.

Cause: `TwigTemplatePath` is not set or points to the wrong directory.

Fix:

```
// Set the base path for includes
Pages.TwigTemplatePath := 'C:\MyApp\templates';
// or
Renderer.TwigTemplatePath := 'C:\MyApp\templates';

// The included file must exist at:
// C:\MyApp\templates\header.html
```

---

8. WebSocket Issues

Connection Refused

Problem: WebSocket connection fails immediately.

Cause 1: Wrong URL scheme. Use `wss://` for secure or `ws://` for insecure.

```
// WRONG
FWebSocket.URL := 'https://api.example.com/ws';

// RIGHT
FWebSocket.URL := 'wss://api.example.com/ws';
```

Cause 2: Server not running or firewall blocking the port.

Fix: Test the WebSocket URL with a browser-based tool (like websocat or a browser extension) to verify the server is reachable.

Messages Not Receiving

Problem: WebSocket connects successfully but `OnMessage` never fires.

Cause 1: `OnMessage` event not wired up.

```
// WRONG -- event handler not assigned
FWebSocket.Connect;

// RIGHT
FWebSocket.OnMessage := procedure(Sender: TObject; const Msg: string)
begin
  TThread.Synchronize(nil, procedure
  begin
    ProcessMessage(Msg);
  end);
end;
FWebSocket.Connect;
```

Cause 2: The server sends binary messages, not text. Check server documentation.

Cause 3: The server requires a subscription message after connecting.

```
FWebSocket.OnConnect := procedure(Sender: TObject)
begin
  { Some servers require subscribing to channels }
  FWebSocket.Send('{"action": "subscribe", "channel": "notifications"}');
end;
```

Auto-Reconnect Not Working

Problem: After a disconnect, the WebSocket does not reconnect.

Cause: Auto-reconnect properties not configured.

Fix:

```
FWebSocket.AutoReconnect := True;
FWebSocket.ReconnectInterval := 5000;  // 5 seconds between attempts

FWebSocket.OnDisconnect := procedure(Sender: TObject)
begin
  TThread.Synchronize(nil, procedure
  begin
    ShowNotification('Connection lost. Reconnecting...');
  end);
end;

FWebSocket.OnReconnect := procedure(Sender: TObject)
begin
  TThread.Synchronize(nil, procedure
  begin
    ShowNotification('Reconnected.');
    { Re-authenticate if needed }
    FWebSocket.Send('{"action": "auth", "token": "' + FBearerToken + '"}');
  end);
end;
```

---

9. Diagnostic Checklists

TTina4REST Checklist

When REST calls are not working:

- Is `BaseUrl` set and correct (no trailing slash)?
- Is the endpoint path correct (starts with `/`)?
- Are SSL DLLs installed for the correct bitness?
- Is authentication set (`SetBearer` or `Username`/`Password`)?
- What status code comes back? (0 = network error, 401 = auth, 404 = wrong path)
- Can you reach the URL from a browser or curl?

TTina4HTMLRender Checklist

When the renderer is not displaying correctly:

- Does the renderer have non-zero width and height?
- Is `RegisterObject` called for RTTI onclick?
- Are the methods public (not private)?
- Is the HTML valid (all tags closed)?
- Are images using HTTPS (SSL DLLs needed)?
- Is `CacheDir` writable?

TTina4HTMLPages Checklist

When pages are not navigating:

- Is `Renderer` linked to a TTina4HTMLRender?
- Does at least one page have `IsDefault := True`?
- Do `href` values match `PageName` values (case-sensitive)?
- Is `OnBeforeNavigate` blocking with `Allow := False`?
- Is the page collection empty?

TTina4RESTRequest Checklist

When data is not loading into MemTable:

- Is `Tina4REST` linked to a TTina4REST component?
- Is `EndPoint` set correctly?
- Is `DataKey` set to the correct JSON key?
- Is `MemTable` linked to a TFDMemTable?
- Does the API response contain the expected `DataKey`?
- Is `SyncMode` set correctly? (Sync needs `IndexFieldNames`)

---

10. Quick Reference: Error to Fix

Symptom	Likely Cause	Section
"Could not load SSL library"	Wrong bitness SSL DLLs	1
Status code 0	Network error or timeout	2
Status code 401	Token expired or wrong auth	2
Status code 404	Wrong endpoint or double path	2
Access violation on JSON parse	nil result not checked	3
"Field not found" on MemTable	snake_case conversion	4
Blank renderer	Zero width/height or empty HTML	5
onclick does nothing	RegisterObject missing	5
No default page shown	IsDefault not set	6
Twig variables empty	Set after template assigned	7
WebSocket not connecting	Wrong URL scheme (https vs wss)	8
UI freezes on REST call	Synchronous call, use async	Ch. 13
Memory leak	TJSONObject not freed	Ch. 13

Summary

Most Tina4 Delphi issues fall into five categories:

- **SSL configuration** -- wrong DLL bitness, missing DLLs, certificate issues
- **REST communication** -- wrong endpoints, auth problems, missing error handling
- **JSON structure** -- nil checks, field name mismatches, nested object handling
- **HTML rendering** -- registration, event wiring, CSS support limits
- **Threading** -- synchronous calls blocking UI, missing TThread.Synchronize

When something does not work, start with the diagnostic checklist for the component involved. Check the status code. Log the response. Verify the field names. Nine times out of ten, the fix is in this chapter.