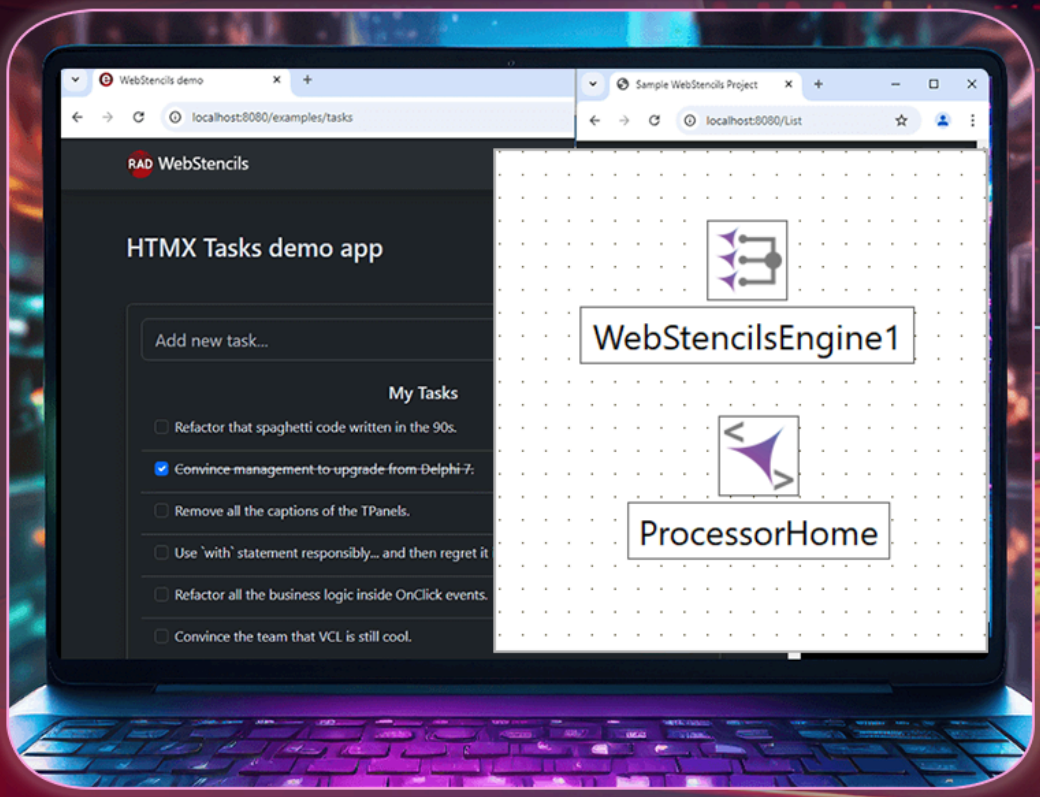
HTMX & WebStencils

Fast Web Development with RAD Studio



Author
Antonio Zapater

embarcadero

HTMX & WebStencils

Fast Web Development with RAD Studio

Preface **5**
01 - Introduction to HTMX **6**
What is HTMX? 6
Brief Overview 6
Comparison with Traditional JavaScript and AJAX 7
Core Concepts 8
hx-get: Fetching Content via GET Requests 8
hx-target: Specifying the Target Element for the Response 8
hx-post: Submitting Data via POST Requests 9
hx-put, hx-patch, hx-delete Requests 9
hx-swap: Controlling How Content is Swapped 9
Additional Core Concepts 10
02 - Introduction to WebBroker **11**
What is WebBroker? 11
Key Features of WebBroker 11
Core Concepts 12
Components and Architecture 12
Creating a WebBroker Application 12
Handling Requests and Responses 13
Deployment and Scalability 13
Session Management in WebBroker 14
Approaches to Session Management 14
Implementing In-Memory Session Management 14
Integrating Session Management into WebModule 17
Security Considerations 19
CSRF Protection 19
Data Validation 21
Cross-site Scripting (XSS) 21
Other Security Considerations 22
03 - Developing your first web app using WebBroker **23**
Introduction 23
Creating a "Hello World" Application 23
Basic To-Do App 24
04 - Advanced Attributes and Security Using HTMX **28**
Introduction 28
Advanced Attributes 28

hx-put and hx-delete: Submitting Requests via PUT and DELETE....28
hx-trigger: Customizing Event Triggers....29
hx-select: Selecting Parts of the Server Response....30
hx-include: Including Additional Data in Requests....30
hx-push-url: Updating the Browser's URL....31
05 - Introduction to WebStencils....32
What is WebStencils?....32
Core Concept....32
Integration with HTMX....32
CSS and JS Agnostic....33
WebStencils Syntax....33
The @ Symbol....33
Curly Braces for Blocks { }....33
Accessing Values with the Dot Notation....33
WebStencils Keywords and Examples....34
@page....34
@query....34
Comments (@* .. *@)....34
@if and @else....35
@if not....35
@ForEach....35
Conclusion....36
06 - Components and Layouts Options....37
Introduction....37
WebStencils Components....37
WebStencils Engine....37
WebStencils Processor....38
TWebStencilsEngine and WebBroker....38
Adding Data with AddVar....39
Layout and Content Placeholders....40
@RenderBody....40
@LayoutPage....41
@Import....42
@ExtraHeader and @RenderHeader....42
Template Patterns....43
Standard Layout....43
Header/Body/Footer....44
Reusable Components....44
Conclusion....44
07 - Migrating the To-Do app to WebStencils....46
Introduction....46

Converting HTML Constants to Templates ... 46
Main Layout Template ... 46
To-Do List Template ... 47
Updating the WebModule ... 48
Adding Extra Functionality ... 51
Task Categories ... 51
Task Filtering ... 53
Conclusion ... 54
08 - Advanced Options with WebStencils ... 56
Introduction ... 56
@query Keyword ... 56
@Scaffolding ... 57
@LoginRequired ... 57
OnValue Event Handler ... 58
Template Patterns ... 59
Standard Layout ... 59
Header/Body/Footer ... 59
Reusable Components ... 60
Conclusion ... 60
09 - Using RAD Server Integration with WebStencils ... 61
Introduction ... 61
Integrating WebStencils with RAD Server ... 61
Using WebStencils Processors ... 61
Using WebStencils Engine ... 62
Recreating the Tasks app to RAD Server ... 64
Database management ... 64
Controller arguments ... 65
From Actions to Endpoints ... 65
Processing data on the requests ... 65
Handling static JS, CSS and images ... 65
Frontend sources ... 65
10 - Resources and Further Learning ... 66
Documentation and links ... 66
Official HTMX documentation (HTMX.org) ... 66
Delphi and HTMX (Embarcadero Blog) ... 67
RAD Server Technical Guide ... 67
HTMX in an MVC Pattern (HTMX.org) ... 67
WebStencils (DocWiki) ... 67
UI/CSS libraries ... 67
Extending HTMX even further ... 68
AlpineJS ... 68
Hyperscript ... 68

Preface

This book focuses on a modern, streamlined approach to web development using HTMX and WebStencils.

HTMX is a lightweight JavaScript alternative for building dynamic web user interfaces and is becoming the goto solution for web developers as it helps them significantly reduce the amount of JavaScript they need to write, making the development process faster and more intuitive, simpler to read and debug and easier to maintain.

HTMX's simplicity aligns perfectly with RAD Studio's rapid application development ethos, allowing developers to focus more on application logic instead of struggling with complex front-end code.

The beauty of WebStencils lies in its template-driven architecture. Instead of reinventing the wheel, developers can expose existing business logic via reusable and customisable templates that seamlessly integrate with existing applications, reducing the friction of bringing older projects to the web. This not only accelerates development but also enhances collaboration between development teams, enabling them to work more closely with existing codebases.

By reading this book, you will learn how to leverage the power of HTMX and WebStencils to develop modern web applications with less effort and greater flexibility. Whether you are working on enhancing existing desktop applications for the web or building new, dynamic web projects, this book provides practical insights to help you get the most out of RAD Studio's evolving web development ecosystem.

You can learn more about RAD Studio or download a free trial to code along with the examples in this book from https://www.embarcadero.com/products/rad-studio

Let's dive into how these technologies can simplify your workflow and take your web development projects to the next level!

Note

WebStencils and HTMX Code Samples that follow a similar pattern as the examples discussed in this guide are available on this GitHub repo.
https://github.com/Embarcadero/WebStencilsDemos

01

Introduction to HTMX

What is HTMX?

Brief Overview

HTMX extends HTML as a hypertext medium, allowing you to make AJAX requests, trigger CSS transitions, create WebSockets, and utilize Server-Sent Events (SSE) directly from HTML elements. This approach reduces the need for custom JavaScript and allows for more declarative, HTML-centric development.

Key features of HTMX include:

- **Simplicity**: HTMX uses HTML attributes to describe behavior, making it easy to understand and maintain.
- **Power**: Despite its simplicity, HTMX allows for sophisticated interactions and real-time updates.
- **Flexibility**: It can be used with any back-end technology and integrates well with existing systems.
- **Performance**: HTMX is lightweight and fast, improving both load times and runtime performance.

Comparison with Traditional JavaScript and AJAX

Traditional web development often involves writing a significant amount of JavaScript to handle user interactions, make AJAX requests, and update the DOM. This approach can lead to complex, hard-to-maintain code, especially as applications grow in size and complexity.

HTMX takes a different approach:

- **Declarative vs Imperative**: Instead of writing JavaScript functions to describe how to fetch and update content, you declare them directly in HTML using attributes.
- **Reduced Boilerplate**: Common patterns like updating a div with the result of an AJAX call require minimal code with HTMX.
- **Improved Readability**: The behavior of an element is visible directly in the HTML, making it easier to understand at a glance.
- **Easier Maintenance**: With behavior tied directly to HTML elements, updating and maintaining HTMX-based code is often easier.

Here's a simple example to illustrate the difference:

Traditional JavaScript/AJAX:

```
<button id="loadButton">Load Content</button>
<div id="content"></div>

<script>
document.getElementById('loadButton').addEventListener('click', function() {
    fetch('/some-content')
        .then(response => response.text())
        .then(data => {
            document.getElementById('content').innerHTML = data;
        });
});
</script>
```

HTMX:

```
<button hx-get="/some-content" hx-target="#content">Load Content</button>
<div id="content"></div>
```

As you can see, the HTMX version is more concise, and the intent is clearer directly from the HTML.

Core Concepts

To effectively use HTMX, it's essential to understand its core concepts and how they work together to create dynamic web applications.

hx-get: Fetching Content via GET Requests

The `hx-get` attribute is used to make GET requests to a server and update the page with the response. When the user interacts with an element that has an `hx-get` attribute (by default, on click), HTMX will make a GET request to the specified URL and update the page with the response.

Example:

```
<button hx-get="/api/user" hx-target="#user-info">
    Load User Info
</button>
<div id="user-info"></div>
```

In this example, when the button is clicked, HTMX will make a GET request to `"/api/user"` and put the response into the div with the id `"user-info"`.

hx-target: Specifying the Target Element for the Response

As seen in the previous example, the `hx-target` attribute specifies which element should be updated with the response from the server. If not specified, the default is to update the element that triggered the request.

Example:

```
<button hx-get="/api/notification" hx-target="#notification-area">
    Check Notifications
</button>
<div id="notification-area"></div>
```

In this case, the response from `"/api/notification"` will be inserted into the div with id `"notification-area"`.

hx-post: Submitting Data via POST Requests

Similar to `hx-get`, the `hx-post` attribute is used to make POST requests. This is typically used for submitting form data or when you need to send data to the server that will cause a change in server state.

Example:

```
<form hx-post="/api/submit" hx-target="#response">
    <input type="text" name="username">
    <button type="submit">Submit</button>
</form>
<div id="response"></div>
```

When this form is submitted, HTMX will make a POST request to "`/api/submit`" with the form data and put the response in the #response div.

hx-put, hx-patch, hx-delete Requests

These other requests follow the same convention as `hx-get` and `hx-post`. It means that `hx-put`, `hx-patch` and `hx-delete` send a PUT, PATCH, and DELETE request respectively. They are used to submit modifications or deletions to the backend (we will see them in more detail later).

hx-swap: Controlling How Content is Swapped

The `hx-swap` attribute allows you to control how the new content is filled into the target element. The most common options are:

- `innerHTML` (default): Replace the inner HTML of the target element
- `outerHTML`: Replace the entire target element
- `beforebegin`: Insert before the target element
- `afterbegin`: Insert as first child of the target element
- `beforeend`: Insert as last child of the target element
- `afterend`: Insert after the target element

Example:

```
<div id="list">
    <button hx-get="/api/item" hx-target="#list" hx-swap="beforeend">
        Add Item
    </button>
</div>
```

This will append the new item to the end of the list, rather than replacing the entire list.

Additional Core Concepts

While the above four concepts form the foundation of HTMX, there are several other important features to be aware of:

1. `hx-trigger`: Allows you to specify what event triggers the AJAX request. By default, it is the Click event for most elements or the Submit event for forms.
2. `hx-params`: Controls what parameters are submitted with the request.
3. `hx-headers`: Allows you to add custom headers to the AJAX request.
4. `hx-vals`: Allows you to add additional values to the parameters submitted with a request.
5. `hx-boost`: A simple way to boost normal anchors and forms with AJAX.
6. `hx-push-url`: Pushes a new URL into the history stack, allowing for updated browser URLs without a full page load.

Understanding these core concepts provides a solid foundation for building dynamic, interactive web applications with HTMX. As you become more familiar with the basics, you'll be able to leverage more advanced HTMX features to create sophisticated, responsive user interfaces with minimal JavaScript.

For more in-depth documentation about HTMX, access their official documentation at this link: https://htmx.org/docs

02

Introduction to WebBroker

What is WebBroker?

WebBroker is a powerful framework within RAD Studio that enables developers to create robust web applications and web services. It provides a solid foundation for building server-side applications and supports the development of RESTful services, SOAP services, and more.

Key Features of WebBroker

1. **Versatility**: WebBroker supports various types of web services (stand-alone, Apache, ISAPI…), making it a versatile choice for different project requirements.
2. **Integration**: It integrates with RAD Studio, providing a familiar development environment for Delphi and C++Builder developers.
3. **Scalability**: WebBroker applications can be easily scaled to handle increasing loads and complex requirements.
4. **Performance**: The framework is designed for high performance, crucial for server-side applications.

Core Concepts

Understanding the core concepts of WebBroker is essential for effectively using WebStencils to build web applications and services.

Components and Architecture

WebBroker applications are built using a set of key components that work together to manage HTTP requests, responses, routing, and middleware. The main components include:

1. `TWebModule`: This is the central component of a WebBroker application. It acts as a container for other WebBroker components and handles the application's overall flow.
2. `TWebDispatcher`: This component is responsible for dispatching incoming HTTP requests to the appropriate action items.
3. `TWebActionItem`: This class defines how specific URL endpoints should be handled. Each action item is associated with a particular URL pattern and defines what should happen when that URL is requested.

The architecture of a WebBroker application typically follows this flow:

1. An HTTP request comes in
2. The `TWebDispatcher` routes the request to the appropriate `TWebActionItem`
3. The `TWebActionItem` executes its associated code
4. A response is generated and sent back to the client

Creating a WebBroker Application

To create a WebBroker application:

1. Select “File/New/Other…/Web/Web Server Application”.
2. You can choose Linux compatibility if you like.
3. Pick a type of application: standalone, Apache module etc.
4. Unless the port 8080 is being used locally on your computer, leave the defaults.

We could think about the WebModule actions as your router, where all the endpoints will be defined.

To configure your `TWebModule` and add `TWebActionItems` to handle different URL endpoints, click anywhere in the `TWebModule` and select Actions on the properties menu. In the menu that will appear, you’ll be able to create new endpoints and the methods associated with them. Like most RAD Studio components, these action items have multiple events available.

Here's a simple example of how you might set up a `TWebActionItem`:

```
procedure TWebModule1.WebModule1WebActionItem1Action(Sender: TObject;
  Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
begin
  Response.Content := '<html><body><h1>Hello, WebBroker!</h1></body></html>';
  Handled := True;
end;
```

This action item would generate a simple HTML response when its associated URL is requested.

Handling Requests and Responses

WebBroker provides a straightforward mechanism for handling HTTP requests and responses:

- `TWebRequest` provides access to all the information about the incoming HTTP request, including parameters, headers, and the request method.
- `TWebResponse` is used to set the response data, including the content, status code, and headers.

Example of accessing request data and setting response data:

```
procedure TWebModule1.HandleGetUser(Sender: TObject;
  Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
var
  UserId: string;
begin
  UserId := Request.QueryFields.Values['id'];
  // Fetch user data based on UserId
  Response.ContentType := 'application/json';
  Response.StatusCode := 200;
  Response.Content := '{"id": "' + UserId + '", "name": "John Doe"}';
  Handled := True;
end;
```

Deployment and Scalability

WebBroker applications offer flexible deployment options:

1. **Standalone Executables**: Can be run as independent web servers (Form-based or CLI).
2. **Apache/ISAPI Extensions**: Can be integrated with web servers like IIS or Apache.

This flexibility allows various deployment scenarios, ensuring scalability and performance for different types of web services. You can start with a standalone executable for development and testing, then deploy it as an ISAPI extension or Apache module for production use with a full-featured web server.

Session Management in WebBroker

While WebBroker doesn't provide built-in session management, it's possible to implement a robust session handling system in your WebBroker applications. Session management is crucial for maintaining user state across multiple requests and is often necessary for features such as user authentication, shopping carts or CSRF protection, to name a few.

Approaches to Session Management

There are several approaches to implementing session management:

1. **In-Memory Sessions**: Store session data in memory. This is fast but doesn't persist if the server restarts and doesn't scale well for multiple server instances.
2. **Database Sessions**: Store session data in a database. This allows for persistence and scalability but may introduce latency.
3. **File-Based Sessions**: Store session data in files on the server. This provides persistence but may have performance or locking issues with a large number of concurrent sessions.
4. **Distributed Cache Sessions**: Use a distributed cache like Redis for session storage. This offers a good balance of performance and scalability.

Implementing In-Memory Session Management

Let's look at a basic implementation of in-memory session management:

```
unit SessionManagerU;

interface

uses
  System.Generics.Collections, System.SysUtils, Web.HTTPApp;

type
  TSessionData = class
  private
```

```
    FValues: TDictionary<string, string>;
  public
    constructor Create;
    destructor Destroy; override;
    property Values: TDictionary<string, string> read FValues;
  end;

  TSessionManager = class
  private
    FSessions: TDictionary<string, TSessionData>;
    function GenerateSessionId: string;
  public
    constructor Create;
    destructor Destroy; override;
    function GetSession(const SessionId: string): TSessionData;
    function CreateSession: string;
    procedure RemoveSession(const SessionId: string);
  end;

implementation

uses
  System.Hash;

{ TSessionData }

constructor TSessionData.Create;
begin
  inherited;
  FValues := TDictionary<string, string>.Create;
end;

destructor TSessionData.Destroy;
begin
  FValues.Free;
  inherited;
end;

{ TSessionManager }

constructor TSessionManager.Create;
begin
  inherited;
  FSessions := TDictionary<string, TSessionData>.Create;
end;

destructor TSessionManager.Destroy;
```

```
begin
  for var Session in FSessions.Values do
    Session.Free;
  FSessions.Free;
  inherited;
end;

function TSessionManager.GenerateSessionId: string;
begin
  var GUID := TGuid.NewGuid.ToString;
  Result := THashSHA2.GetHashString(GUID);
end;

function TSessionManager.GetSession(const SessionId: string): TSessionData;
begin
  if not FSessions.TryGetValue(SessionId, Result) then
    Result := nil;
end;

function TSessionManager.CreateSession: string;
var
  SessionId: string;
  SessionData: TSessionData;
begin
  SessionId := GenerateSessionId;
  SessionData := TSessionData.Create;
  SessionData.Values.AddOrSetValue('SessionId', SessionId);
  FSessions.Add(SessionId, SessionData);
  Result := SessionId;
end;

procedure TSessionManager.RemoveSession(const SessionId: string);
var
  SessionData: TSessionData;
begin
  if FSessions.TryGetValue(SessionId, SessionData) then
  begin
    SessionData.Free;
    FSessions.Remove(SessionId);
  end;
end;

end.
```

Integrating Session Management into WebModule

To use session management in your WebBroker application, you can create a base WebModule class that includes session handling:

```
type
  TWebModuleWithSession = class(TWebModule)
    procedure WebModule1DefaultHandlerAction(Sender: TObject;
      Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
  private
    FSessionManager: TSessionManager;
    function GetSessionId(Request: TWebRequest; Response: TWebResponse): string;
    function GetSession(Request: TWebRequest; Response: TWebResponse): TSessionData;
  public
    constructor Create(AOwner: TComponent); override;
    destructor Destroy; override;
  end;

var
  WebModuleClass: TComponentClass = TWebModuleWithSession;

implementation

uses
  DateUtils;

{%CLASSGROUP 'System.Classes.TPersistent'}

{$R *.dfm}

constructor TWebModuleWithSession.Create(AOwner: TComponent);
begin
  inherited;
  FSessionManager := TSessionManager.Create;
end;

destructor TWebModuleWithSession.Destroy;
begin
  FSessionManager.Free;
  inherited;
end;

function TWebModuleWithSession.GetSessionId(Request: TWebRequest; Response:
TWebResponse): string;
const
  SessionCookieName = 'SessionId';
var
```

```
  LCookie: TCookie;
begin
  Result := Request.CookieFields.Values[SessionCookieName];
  if Result = '' then
  begin
    Result := FSessionManager.CreateSession;
    LCookie := Response.Cookies.Add;
    LCookie.Name := SessionCookieName;
    LCookie.Value := Result;
    LCookie.Path := '/';
    // For demo purposes, the Cookie is only valid for 1 minute
    LCookie.Expires := IncMinute(Now, 1);
  end;
end;

function TWebModuleWithSession.GetSession(Request: TWebRequest; Response: TWebResponse):
TSessionData;
var
  SessionId: string;
begin
  SessionId := GetSessionId(Request, Response);
  Result := FSessionManager.GetSession(SessionId);
end;
```

With this setup, you can easily access session data in your WebModule's action handlers:

```
procedure TWebModuleWithSession.WebModule1DefaultHandlerAction(Sender: TObject;
  Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
var
  LSession: TSessionData;
begin
  LSession := GetSession(Request, Response);
  // Use Session.Values to store or retrieve session data
  LSession.Values.AddOrSetValue('LastAccess', DateTimeToStr(Now));
  var HTMLResponse := '''
  <html>
  <head><title>Web Server Application</title></head>
  <body>Web Server Application</body>
  ''';
  HTMLResponse := HTMLResponse +
  '<p>Session ID: <strong>' + LSession.Values['SessionId'] + '</strong></p>' +
  '<p>Last Access: <strong>' + LSession.Values['LastAccess'] + '</strong></p>';
  HTMLResponse := HTMLResponse + '''
  </html>
  ''';
```

```
  Response.Content := HTMLResponse;
end;
```

Note

Keep in mind that this SessionManager implementation is just a proof of concept. To use it in production, further implementations would be required, such as Deleting expired sessions, implementing TSessionData more isolatedly or handling SessionManager as a singleton for multi-thread environments.

Security Considerations

CSRF Protection

Cross-Site Request Forgery (CSRF) is a type of attack where unauthorized commands are sent from a user that the web application trusts. To protect against CSRF:

1. Use request-verification tokens: Generate a unique token for each session and include it in forms.
2. Validate the token on the server for each non-GET request.

Example implementation in WebBroker:

```
unit CsrfProtection;

interface

uses
  System.SysUtils, Web.HTTPApp, SessionManager;

type
  TCsrfWebModule = class(TWebModule)
  private
    FSessionManager: TSessionManager;
    function GenerateCSRFToken: string;
  public
    procedure SendHTMLResponse(Sender: TObject;
      Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
    procedure ValidateCSRFToken(Request: TWebRequest);
    // The business logic related to session is the same as in the previous examples
```

```
    function GetSessionId(Request: TWebRequest; Response: TWebResponse): string;
    function GetSession(Request: TWebRequest; Response: TWebResponse): TSessionData;
  end;

implementation

uses
  System.Hash;

function TCsrfWebModule.GenerateCSRFToken: string;
begin
  var GUID := TGuid.NewGuid.ToString;
  Result := THashSHA2.GetHashString(GUID);
end;

procedure TCsrfWebModule.SendHTMLResponse(Sender: TObject;
  Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
var
  CSRFToken: string;
  Session: TSessionData;
begin
  Session := GetSession(Request, Response);
  CSRFToken := GenerateCSRFToken;
  LSession.Values.AddOrSetValue('CSRFToken', LCSRFToken);
  Response.Content :=
    '<form hx-post="/submit">' +
    '<input type="hidden" name="csrf_token" value="' + CSRFToken + '">' +
    // ... rest of your form ...
    '</form>';
  Handled := True;
end;

procedure TCsrfWebModule.ValidateCSRFToken(Request: TWebRequest);
var
  RequestToken, SessionToken: string;
  Session: TSessionData;
begin
  Session := GetSession(Request, Response);
  RequestToken := Request.ContentFields.Values['csrf_token'];
  SessionToken := Session.Values['CSRFToken'];

  if (RequestToken = '') or (SessionToken = '') or (RequestToken <> SessionToken) then
    raise Exception.Create('Invalid CSRF token');
end;

end.
```

In this implementation:

1. We generate a unique CSRF token for each form submission.
2. The token is stored in the session and included in the form as a hidden field.
3. When processing form submissions, we validate the token from the request against the one stored in the session.

Data Validation

Always validate and sanitize data on the server side, even if client-side validation is in place.

Example:

```
procedure TWebModule1.ValidateUserInput(const Username: string);
begin
  if Length(Username) < 3 then
    raise Exception.Create('Username must be at least 3 characters long');

  if not TRegEx.IsMatch(Username, '^[a-zA-Z0-9_]+$') then
    raise Exception.Create('Username can only contain letters, numbers, and
underscores');
end;

procedure TWebModule1.HandleUserRegistration(Sender: TObject;
  Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
var
  Username: string;
begin
  Username := Request.ContentFields.Values['username'];
  try
    ValidateUserInput(Username);
    // Process registration...
    Response.Content := 'Registration successful';
  except
    on E: Exception do
      Response.Content := 'Registration failed: ' + E.Message;
  end;
  Handled := True;
end;
```

Cross-site Scripting (XSS)

To prevent XSS attacks, always encode or escape user-generated content before including it in your HTML responses. The library NetEncoding offers multiple integrated ways to escape HTML strings.

Example:

```
function HTMLEncode(const S: string): string;
begin
  Result := TNetEncoding.HTML.Encode(S);
end;

procedure TWebModule1.DisplayUserComment(Sender: TObject;
  Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
begin
  var UserComment := Request.QueryFields.Values['comment'];
  Response.Content := '<div class="comment">' + HTMLEncode(UserComment) + '</div>';
   Handled := True;
end;
```

Other Security Considerations

1. **SQL Injection Prevention**: Use parameterized queries or prepared statements when interacting with databases.
2. **Secure Communication**: Use HTTPS to encrypt data in transit.
3. **Authentication and Authorization**: Implement robust user authentication and proper access controls.

Implementing these security measures can significantly enhance the security of your WebBroker applications. Remember that security is an ongoing process, and it's important to stay updated on the latest security best practices and vulnerabilities.

You can check more detailed info in the docwiki: Using WebBroker

03

Developing your first web app using WebBroker

Introduction

In this chapter, we'll walk through the process of creating your first web application using WebBroker. We'll start with a simple "Hello World" example and then move on to creating a basic To-Do app.

Creating a "Hello World" Application

Let's begin with a simple "Hello World" application that demonstrates the use of HTMX with WebBroker.

1. Start by creating a new WebBroker Application in RAD Studio.
2. In your WebModule, add a new WebActionItem and set its name to "ActionHello".
3. Double-click on the ActionHello item to create an event handler.
4. Implement the event handler as follows:

```
procedure TWebModule1.WebModule1ActionHelloAction(Sender: TObject;
  Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
begin
  Response.Content := '''
    <html>
      <head>
        <script src="https://unpkg.com/htmx.org@2.0.2"></script>
      </head>
      <body>
        <h1>Hello, WebBroker and HTMX!</h1>
        <button hx-get="/greet" hx-target="#greeting">Say Hello</button>
        <div id="greeting"></div>
      </body>
    </html>
    ''';
  Handled := True;
end;
```

5. Add another WebActionItem named "ActionGreet" and implement its event handler:

```
procedure TWebModule1.WebModule1ActionGreetAction(Sender: TObject;
  Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
begin
  Response.Content := '<p>Hello from the server!</p>';
  Handled := True;
end;
```

6. Run your application. When you open it in a web browser and click the "Say Hello" button, you should see the greeting appear without a full page reload.

This simple example demonstrates how WebBroker can serve HTML content and how HTMX can be used to make dynamic requests to the server.

Basic To-Do App

Now let's create a more complex application: a basic To-Do app that allows users to add and view tasks.

1. Create a new WebBroker Application or use the existing one.
2. Add a new unit to your project to store the To-Do list:

```
unit TodoList;

interface

uses
  System.Generics.Collections;

type
  TTodoItem = record
    Id: Integer;
    Text: string;
  end;

  TTodoList = class
  private
    FItems: TList<TTodoItem>;
    FNextId: Integer;
  public
    constructor Create;
    destructor Destroy; override;
    function AddItem(const Text: string): Integer;
    function GetItems: TArray<TTodoItem>;
  end;

implementation

{ TTodoList }

constructor TTodoList.Create;
begin
  FItems := TList<TTodoItem>.Create;
  FNextId := 1;
end;

destructor TTodoList.Destroy;
begin
  FItems.Free;
  inherited;
end;

function TTodoList.AddItem(const Text: string): Integer;
var
  Item: TTodoItem;
begin
  Item.Id := FNextId;
  Item.Text := Text;
  FItems.Add(Item);
```

```
  Result := FNextId;
  Inc(FNextId);
end;

function TTodoList.GetItems: TArray<TTodoItem>;
begin
  Result := FItems.ToArray;
end;

end.
```

3. In your WebModule, add a field for the TodoList:

```
FTodoList: TTodoList;
```

Initialize it in the WebModule's constructor and free it in the destructor.

4. Add WebActionItems for displaying the To-Dolist, adding new items, and updating the list. Implement their event handlers as follows:

```
procedure TWebModule1.WebModule1ActionTodoListAction(Sender: TObject;
  Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
var
  Html: string;
  Item: TTodoItem;
Begin
  Html := '''
    <html>
      <head>
        <script src="https://unpkg.com/htmx.org@2.0.2"></script>
      </head>
      <body>
        <h1>Todo List</h1>
        <form hx-post="/add-todo" hx-target="#todo-list">
          <input type="text" name="todo-text" placeholder="New todo item">
          <button type="submit">Add</button>
        </form>
        <div id="todo-list">
    ''';

  for Item in FTodoList.GetItems do
  begin
    Html := Html + Format('<p>%d: %s</p>', [Item.Id, Item.Text]);
```

```
  end;

  Html := Html + '''
        </div>
      </body>
    </html>
    ''';

  Response.Content := Html;
  Handled := True;
end;

procedure TWebModule1.WebModule1ActionAddTodoAction(Sender: TObject;
  Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
var
  TodoText: string;
  Html: string;
  Item: TTodoItem;
begin
  TodoText := Request.ContentFields.Values['todo-text'];
  FTodoList.AddItem(TodoText);

  Html := '';
  for Item in FTodoList.GetItems do
  begin
    Html := Html + Format('<p>%d: %s</p>', [Item.Id, Item.Text]);
  end;

  Response.Content := Html;
  Handled := True;
end;
```

5. Run your application. You should now be able to add new To-Doitems and see the list update dynamically without full page reloads.

This basic To-Do app demonstrates how to use WebBroker to handle different types of requests (GET for displaying the list, POST for adding items) and how to use HTMX to create a dynamic user interface that interacts with the server.

Remember, in a real-world application, you'd want to add error handling, input validation, and possibly persist the todo list to a database. But this example serves as a good starting point for understanding how WebBroker and HTMX can work together to create interactive web applications.

04

Advanced Attributes and Security Using HTMX

Introduction

In this chapter, we'll explore some of the more advanced attributes in HTMX and discuss important security considerations when building web applications with HTMX and WebBroker.

Advanced Attributes

HTMX provides a rich set of attributes that allow for fine-grained control over AJAX requests and DOM updates. Let's explore some of the more advanced attributes:

hx-put and hx-delete: Submitting Requests via PUT and DELETE

While `hx-get` and `hx-post` are commonly used, HTMX also supports other HTTP methods like PUT and DELETE.

Example using `hx-put`:

```
<button hx-put="/api/user/1" hx-target="#user-info">
    Update User
</button>
```

Example using `hx-delete`:

```
<button hx-delete="/api/user/1" hx-target="#user-list">
    Delete User
</button>
```

In your WebBroker application, you'll need to handle these requests appropriately:

```
procedure TWebModule1.HandlePutRequest(Sender: TObject;
  Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
begin
  if Request.MethodType = mtPUT then
  begin
    // Handle PUT request
    Response.Content := 'User updated';
    Handled := True;
  end;
end;

procedure TWebModule1.HandleDeleteRequest(Sender: TObject;
  Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
begin
  if Request.MethodType = mtDELETE then
  begin
    // Handle DELETE request
    Response.Content := 'User deleted';
    Handled := True;
  end;
end;
```

hx-trigger: Customizing Event Triggers

The `hx-trigger` attribute allows you to specify what event triggers the AJAX request. By default, it's 'click' for most elements and 'submit' for forms.

Example:

```
<input type="text"
       name="search"
       hx-get="/search"
       hx-trigger="keyup changed delay:500ms"
       hx-target="#search-results">
```

This will trigger a search request 500 milliseconds after the user stops typing.

hx-select: Selecting Parts of the Server Response

The `hx-select` attribute allows you to select a subset of the server response to update the target.

Example:

```
<button hx-get="/api/user"
        hx-target="#user-name"
        hx-select="#name">
    Load User Name
</button>
```

In this case, only the element with the id "name" from the server response will be used to update the target.

This approach allows your server to return full HTML pages (which could be useful for non-HTMX requests or for debugging), while still enabling HTMX to selectively update only parts of your page.

It's worth noting that while returning a full HTML document is possible and sometimes useful, in many cases when working with HTMX, you might choose to have your server return only the specific HTML fragment that's needed.

hx-include: Including Additional Data in Requests

`hx-include` allows you to include values from other elements in your request.

Example:

```
<form hx-post="/api/submit" hx-include="#extra-data">
    <input type="text" name="username">
    <button type="submit">Submit</button>
</form>
<input type="hidden" id="extra-data" name="extra" value="some-value">
```

This will include the value of the "extra-data" HTML input element in the form submission.

hx-push-url: Updating the Browser's URL

`hx-push-url` allows you to update the browser's URL without a full page load, useful for maintaining navigability in single-page applications.

Example:

```
<button hx-get="/new-page"
        hx-push-url="true">
    Go to New Page
</button>
```

This will update the browser's URL when the button is clicked, allowing for proper back/forward navigation.

05

Introduction to WebStencils

What is WebStencils?

WebStencils is a new server-side script-based integration and processing technology for HTML files, introduced in RAD Studio 12.2. It allows developers to create modern, professional-looking websites based on any JavaScript library, powered by the data extracted and processed by a RAD Studio server-side application.

Core Concept

WebStencils enables the development of navigational websites such as blogs, online catalogues, ordering systems, reference sites like dictionaries, and wikis. It provides a template engine that resembles ASP.NET Razor processing but is specifically designed for RAD Studio.

Integration with HTMX

WebStencils complements HTMX well. HTMX pages can benefit from server-side code generation and hook into REST servers for content updates. Meanwhile, Delphi web technologies can offer high-quality page generation and REST APIs.

CSS and JS Agnostic

One of WebStencils' key features is that it doesn't force you to use any specific JavaScript or CSS library. It's purely a template engine for server-side rendering, allowing you to use any front-end technologies you prefer.

WebStencils Syntax

WebStencils uses a simple syntax based on two main elements:

1. The @ symbol
2. Curly braces for blocks { }

The @ Symbol

The @ symbol is used as a special marker in WebStencils. It can be followed by:

- The name of an object or field
- A special processing keyword
- Another @

For example:

```
@object.value
```

This syntax accesses the 'value' property of 'object'. The object name is a symbolic local name that can match an actual server application object or be resolved in code while processing an `OnValue` event handler.

Curly Braces for Blocks { }

Curly braces are used to denote conditional or repeated blocks. They are processed only when used after a specific WebStencils conditional statement.

Accessing Values with the Dot Notation

Here's an example of how values are accessed in WebStencils:

```
<h2>User Profile</h2>
```

```
<p>Name: @user.name</p>
<p>Email: @user.email</p>
```

WebStencils Keywords and Examples

Let's explore the various WebStencils keywords with examples:

@page

The `@page` keyword allows accessing multiple properties from the page as well as access connection.

Example:

```
<p>Current page is: @page.pagename</p>
```

@query

The `@query` keyword is used to read HTTP query parameters.

Example:

```
<p>You searched for: @query.searchTerm</p>
```

In this example, searchTerm would be a parameter included in the URL: `yourdomain.com?searchTerm="mySearch"`

Comments (@* .. *@)

Comments in WebStencils are enclosed in `@*  *@` and are omitted from the resulting HTML.

Example:

```
@* This is a comment and will not appear in the output *@
<p>This will appear in the output</p>
```

@if and @else

Conditional execution is handled with `@if` and `@else`.

Example:

```
@if user.isLoggedIn {
    <p>Welcome, @user.name!</p>
}
@else {
    <p>Please log in to continue.</p>
}
```

@if not

For negative conditional execution, use `@if not`.

Example:

```
@if not cart.isEmpty {
    <p>You have @cart.itemCount items in your cart.</p>
}
@else {
    <p>Your cart is empty.</p>
}
```

@ForEach

The `@ForEach` keyword is used for iteration over elements in an enumerator.

Example:

```
<ul>
@ForEach (var product in productList) {
    <li>@product.name - @product.price</li>
}
</ul>
```

Conclusion

WebStencils provides a powerful way to generate dynamic web pages in RAD Studio applications. Its syntax allows for seamless integration of server-side logic into your HTML templates. By using the @ symbol, curly braces, and various keywords, you can create dynamic and interactive web pages with ease.

As you become more familiar with WebStencils, you'll be able to leverage its full potential to create complex and responsive web applications. In the following chapters, we'll explore more advanced features of WebStencils, including templates, layouts, and how to use WebStencils components effectively.

06

Components and Layouts Options

Introduction

In this chapter, we'll explore the core components of WebStencils, work with templates and layouts, and discuss common template patterns. Understanding these concepts will allow you to create more organized, maintainable, and reusable web applications with WebStencils.

WebStencils Components

WebStencils introduces two main components: the WebStencils Engine and the WebStencils Processor which work together to process your templates and generate the desired final HTML output.

WebStencils Engine

The WebStencils Engine is the central component that manages the overall processing of your templates. It can be used in two primary scenarios:

1. **Connected to WebStencilsProcessor components**: In this setup, the engine provides shared settings and behavior for multiple processors, reducing the need to customize each processor individually.

2. **Standalone usage**: The engine can create WebStencilsProcessor components as needed, allowing you to place only the engine component on your web modules.

Key properties and methods of `TWebStencilsEngine` include:

- `Dispatcher`: Specifies a file dispatcher (implements `IWebDispatch`) for post-processing of text files.
- `PathTemplates`: A collection of request path templates used for matching and processing requests.
- `RootDirectory`: Specifies the file system root path for relative file paths.
- `DefaultFileExt`: Sets the default file extension (default is '.html').
- `AddVar`: Adds objects to the list of script variables available to processors.
- `AddModule`: Scans an object for members marked with `[WebStencilsVar]` attributes and adds them as script variables.

WebStencils Processor

The WebStencils Processor is responsible for processing individual files (typically HTML) and their associated templates. It can be used standalone or created and managed by the WebStencils Engine.

Key properties and methods of `TWebStencilsProcessor` include:

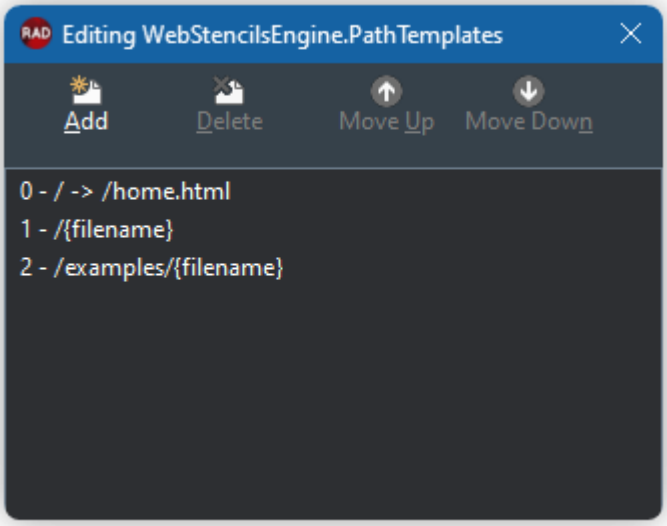
- `InputFilename`: Specifies the file to process.
- `InputLines`: Allows direct assignment of content to process.
- `Engine`: Specifies the engine to inherit data variables, event handlers, etc. (optional).
- `Content`: Produces the final processed content.
- `AddVar`: Adds objects to the list of script variables available to the processor.

TWebStencilsEngine and WebBroker

A WebStencils engine component can be easily linked to a `TWebFileDispatcher` to automatically deliver templates using WebBroker.

Once this is configured, wildcards can be used in the PathTemplates property to map automatically requests to file.

Example:



Let's analyze each path defined:

`0-/->/home.html`: Redirect accessing to the root of the website to the template `home.html`

`1-/{filename}`: The engine will try to map URIs to filenames. For example, accessing https://localhost:8080/basics will search for a template with the name `basics.html` in the predefined templates folder.

`2-/examples/{filename}`: Same behavior as the previous example, but in this case the templates will be searched in the path `/examples`

Adding Data with AddVar

The `AddVar` method is crucial for passing data from your Delphi code to your WebStencils templates. There are several ways to use AddVar:

1. **Direct object assignment**:

```
WebStencilsProcessor1.AddVar('user', UserObject);
```

2. **Using anonymous methods**:

```
WebStencilsProcessor1.AddVar('products',
```

```
  function: TObject
  begin
    Result := GetProductList;
  end);
```

3. **Using attributes**: You can mark fields, properties, or methods in your classes with the `[WebStencilsVar]` attribute, then use `AddModule` to add all marked members as script variables:

```
type
  TMyDataModule = class(TDataModule)
    [WebStencilsVar]
    FDMemTable1: TFDMemTable;
    [WebStencilsVar]
    function GetCurrentUser: TUser;
  end;

// In your WebModule:
WebStencilsProcessor1.AddModule(DataModule1);
```

Important: Keep in mind that only objects with the `GetEnumerator` method, where the enumerator returns an object value will work. Records can't be used from WebStencils.

Layout and Content Placeholders

WebStencils provides a powerful layout system similar to other template engines like Mustache, Blade, ERB or Razor among others. This system allows you to define a common structure for your pages and inject specific content into that structure.

@RenderBody

The `@RenderBody` directive is used in the layout template to indicate where the content from the specific page should be inserted. For example, let's imagine we have a common HTML structure like this, and we call it BaseTemplate.html:

```
<!-- This is the BaseTemplate.html -->
<!DOCTYPE html>
<html>
<head>
    <title>My Website</title>
</head>
<body>
    <header>
        <!-- Common header content -->
    </header>

    <main>
        @RenderBody
    </main>

    <footer>
        <!-- Common footer content -->
    </footer>
</body>
</html>
```

The `@Renderbody` keyword will be replaced with the content that will be included in other child templates.

@LayoutPage

The `@LayoutPage` directive is used in a content page to specify which layout template should be used as the structure for that page. It's typically placed at the top of the content file:

```
<!-- BaseTemplate.html will be used as a base and the rest of the content included where
the @RenderBody tag is located -->
@LayoutPage BaseTemplate
<h2>Welcome to My Page</h2>
<p>This is the content of my page.</p>
```

In this example, the BaseTemplate.html will be used as a base layout and the content under the keyword `@LayoutPage` will be rendered in the `@RenderBody` location we defined in the previous example.

@Import

The `@Import` directive allows you to merge an external file into a specific location in the current template. This is useful for creating reusable components.

This directive allows you to structure your templates in nested folders. You can also omit the file extension, as long as the default one has been defined in the Engine or Processor.

```
@Import Sidebar.html

@* Same Behavior *@
@Import Sidebar

@* Nested folder example *@
@Import folder/Sidebar
```

@ExtraHeader and @RenderHeader

The `@ExtraHeader` directive allows you to define additional content that should be placed in the section of your HTML document. This is particularly useful for including page-specific CSS or JavaScript files.

Here's how it works:

1. In your content page, you use `@ExtraHeader` to define additional header content.
2. In your layout template, you use `@RenderHeader` to specify where this extra header content should be inserted.

Let's look at an example:

Content Page (ProductPage.html):

```
@LayoutPage BaseTemplate
@ExtraHeader {
    <link rel="stylesheet" href="/css/product-page.css">
    <script src="/js/product-details.js"></script>
}

<h1>Product Details</h1>
<div id="product-info">
    <!-- Product information here -->
</div>
```

Layout Template (BaseTemplate.html):

```
<!DOCTYPE html>
<html>
<head>
    <title>My Online Store</title>
    <link rel="stylesheet" href="/css/main.css">
    @RenderHeader
</head>
<body>
    <main>
        @RenderBody
    </main>
</body>
</html>
```

In this example, the product page defines extra CSS and JavaScript files that are specific to the product details page. The `@RenderHeader` directive in the layout template ensures that these extra resources are included in the final HTML output.

This approach allows you to have a common base template while still allowing individual pages to add their own resources or meta tags as needed.

Remember that you can have multiple `@ExtraHeader` blocks in your content page if needed. They will all be rendered where `@RenderHeader` is placed in the layout template.

By utilizing `@LayoutPage`, `@RenderBody`, `@ExtraHeader`, and `@RenderHeader` together, you can create highly flexible and maintainable template structures.

Template Patterns

With WebStencils, you can apply several common template patterns to organize your application's views more easily.

Standard Layout

This pattern uses the built-in `@LayoutPage` and `@RenderBody` approach we've already discussed. It's ideal for maintaining a consistent structure across your site while allowing individual pages to provide their specific content.

Header/Body/Footer

In this pattern, each page is an individual template, but they all share common parts like headers and footers. Instead of using a layout page, you might structure your templates like this:

```
@Import Header.html

<main>
    <!-- Page-specific content here -->
</main>

@Import Footer.html
```

This approach offers more flexibility than the standard layout pattern but may require more management of common elements.

Reusable Components

Using the `@Import` directive, you can define individual sets of components that can be reused throughout your application. This directive also allows the passing of iterable objects and even defining aliases for a more agnostic component definition. For example:

```
<div class="product-list">
  @Import ProductList { @list = @ProductList }
</div>

<div class="tasks">
  @ForEach (var Task in Tasks.AllTasks) {
            @Import partials/tasks/item { @Task }
      }
</div>
```

This pattern allows you to break your UI into smaller, manageable pieces that can be easily maintained and reused across different pages.

Conclusion

WebStencils provides a flexible and powerful system for creating templates and layouts in your web applications. By leveraging the WebStencils Engine and Processor components, using `AddVar` to pass data to your templates, and utilizing layout directives like `@LayoutPage`, `@RenderBody`,

and `@Import`, you can create well-structured, maintainable, and reusable views for your web applications.

The template patterns we've discussed - standard layout, header/body/footer, and reusable components - offer different approaches to organizing your views. Choose the pattern (or combination of patterns) that best fits your application's needs and your team's workflow.

In the next chapter, we'll explore WebStencils' more advanced features, including working with forms, handling user input, and implementing dynamic content updates.

07

Migrating the To-Do app to WebStencils

Introduction

In this chapter, we'll take the To-Do app we built earlier using plain HTML in constants and migrate it to WebStencils templates. This migration will demonstrate how WebStencils can make our code more maintainable and scalable. We'll also add a few extra functionalities to showcase the flexibility of our new approach.

Check out the GitHub Repo with the WebStencils demos to see a project that includes a functional To-Do app that follows conceptually the same idea as the code shown in this guide.

Converting HTML Constants to Templates

Let's start by converting our HTML constants into WebStencils templates. We'll create separate templates for different parts of our application.

Main Layout Template

First, let's create a main layout template that will serve as the base for our application.

Create a file named `layout.html`:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <link href="https://cdn.jsdelivr.net/npm/bootstrap@@5.1.3/dist/css/bootstrap.min.css"
rel="stylesheet">
    <script src="https://unpkg.com/htmx.org@@1.9.2"></script>
    @RenderHeader
</head>
<body>
    <div class="container">
        @RenderBody
    </div>
    <script
src="https://cdn.jsdelivr.net/npm/bootstrap@@5.1.3/dist/js/bootstrap.bundle.min.js"></scr
ipt>
</body>
</html>
```

In the previous example we have included different libraries directly from CDNs. In case any of those links have specific versions, it's important to escape the `@` symbol with `@@` so it's not parsed by WebStencils.

To-Do List Template

Now, let's create a template for the To-Do list. Create a file named `todo-list.html`:

```
@LayoutPage layout.html

<div id="todo-list">
    @ForEach (var todo in Todos) {
        <div class="card mb-2">
            <div class="card-body d-flex justify-content-between align-items-center">
                <span>@todo.Description</span>
                <div>
                    @if not todo.Completed {
```

```
                        <button class="btn btn-sm btn-success"
hx-post="/complete/@todo.Id" hx-target="#todo-list">Complete</button>
                    }
                    <button class="btn btn-sm btn-danger" hx-delete="/delete/@todo.Id"
hx-target="#todo-list">Delete</button>
                </div>
            </div>
        </div>
    }
</div>

<form hx-post="/add" hx-target="#todo-list" class="mt-4">
    <div class="input-group">
        <input type="text" name="description" class="form-control" placeholder="New todo
item" required>
        <button type="submit" class="btn btn-primary">Add</button>
    </div>
</form>
```

Updating the WebModule

Now, let's update our WebModule to use these new templates. Keep in mind that in the code below, the `TTodoList` class has been extended with extra methods like `Delete`, `Complete`, or `GetAllItems`. This code is not included in this guide because it is pure Delphi code for simplicity. Check the GitHub demo project to see a similar code approach.

```
unit TodoWebModule;

interface

uses
  System.SysUtils, System.Classes, Web.HTTPApp, TodoList, WebStencils;

type
  TTodoWebModule = class(TWebModule)
    procedure WebModuleCreate(Sender: TObject);
    procedure WebModuleDestroy(Sender: TObject);
    procedure WebModuleDefault(Sender: TObject; Request: TWebRequest;
      Response: TWebResponse; var Handled: Boolean);
  private
    FTodoList: TTodoList;
    FWebStencilsProcessor: TWebStencilsProcessor;
```

```
    procedure HandleGetTodoList(Response: TWebResponse);
    procedure HandleAddTodo(Request: TWebRequest; Response: TWebResponse);
    procedure HandleCompleteTodo(Request: TWebRequest; Response: TWebResponse);
    procedure HandleDeleteTodo(Request: TWebRequest; Response: TWebResponse);
  public
    { Public declarations }
  end;

var
  TodoWebModule: TTodoWebModule;

implementation

{%CLASSGROUP 'System.Classes.TPersistent'}

{$R *.dfm}

procedure TTodoWebModule.WebModuleCreate(Sender: TObject);
begin
  FTodoList:= TTodoList.Create;
  FWebStencilsProcessor := TWebStencilsProcessor.Create(Self);
  FWebStencilsProcessor.TemplateFolder := ExtractFilePath(ParamStr(0)) + 'templates\';
end;

procedure TTodoWebModule.WebModuleDestroy(Sender: TObject);
begin
  FTodoList.Free;
  FWebStencilsProcessor.Free;
end;

procedure TTodoWebModule.WebModuleDefault(Sender: TObject; Request: TWebRequest;
  Response: TWebResponse; var Handled: Boolean);
Begin
// This is a WebBroker action that handles all the requests. For better maintainability,
it should be split into different actions.
  if Request.PathInfo = '' then
    HandleGetTodoList(Response)
  else if (Request.PathInfo = '/add') and (Request.MethodType = mtPost) then
    HandleAddTodo(Request, Response)
  else if Request.PathInfo.StartsWith('/complete/') and (Request.MethodType = mtPost)
then
    HandleCompleteTodo(Request, Response)
  else if Request.PathInfo.StartsWith('/delete/') and (Request.MethodType = mtDelete)
then
    HandleDeleteTodo(Request, Response)
  else
  begin
```

```
    Response.Content := 'Not Found';
    Response.StatusCode := 404;
  end;

  Handled := True;
end;

procedure TTodoWebModule.HandleGetTodoList(Response: TWebResponse);
begin
  FWebStencilsProcessor.AddVar('Todos', FTodoList.GetAllItems);
  FWebStencilsProcessor.InputFileName := 'todo-list.html';
  Response.Content := FWebStencilsProcessor.Content;
end;

procedure TTodoWebModule.HandleAddTodo(Request: TWebRequest; Response: TWebResponse);
var
  Description: string;
begin
  Description := Request.ContentFields.Values['description'];
  FTodoList.AddItem(Description);
  HandleGetTodoList(Response);
end;

procedure TTodoWebModule.HandleCompleteTodo(Request: TWebRequest; Response:
TWebResponse);
var
  ItemId: Integer;
Begin
  // The ItemId is extracted from the PathInfo of the Request and converted to int
  ItemId := StrToIntDef(Request.PathInfo.Substring('/complete/'.Length), -1);
  if ItemId <> -1 then
    FTodoList.CompleteItem(ItemId);
  HandleGetTodoList(Response);
end;

procedure TTodoWebModule.HandleDeleteTodo(Request: TWebRequest; Response: TWebResponse);
var
  ItemId: Integer;
Begin
  ItemId := StrToIntDef(Request.PathInfo.Substring('/delete/'.Length), -1);
  if ItemId <> -1 then
    FTodoList.DeleteItem(ItemId);
  HandleGetTodoList(Response);
end;

end.
```

Note

This code above is oversimplified. For better visual understanding, it doesn't use Actions for each endpoint. The demo available on GitHub follows a more maintainable MVC approach and better abstraction of the logic.

Adding Extra Functionality

Now that we've migrated our app to use WebStencils, let's add some extra functionality to demonstrate the scalability and maintainability of our new structure.

Task Categories

Let's add the ability to categorize tasks. First, update the `TTodoItem` class in your `TTodoList` unit:

```
TTodoItem = class
public
  Id: Integer;
  Description: string;
  Completed: Boolean;
  Category: string;
  constructor Create(AId: Integer; const ADescription, ACategory: string);
end;

constructor TTodoItem.Create(AId: Integer; const ADescription, ACategory: string);
begin
  Id := AId;
  Description := ADescription;
  Completed := False;
  Category := ACategory;
end;
```

Now, update the `todo-list.html` template to include categories:

Update the `HandleAddTodo` method in the WebModule:

```
@LayoutPage layout.html

<div id="todo-list">
    @ForEach (var todo in Todos) {
        <div class="card mb-2">
            <div class="card-body d-flex justify-content-between align-items-center">
                <div>
                    <span>@todo.Description</span>
                    <small class="text-muted ms-2">[@todo.Category]</small>
                </div>
                <div>
                    @if not todo.Completed {
                        <button class="btn btn-sm btn-success"
                                hx-post="/complete/@todo.Id"
                                hx-target="#todo-list">Complete</button>
                    }
                    <button class="btn btn-sm btn-danger"
                            hx-delete="/delete/@todo.Id"
                            hx-target="#todo-list">Delete</button>
                </div>
            </div>
        </div>
    }
</div>

<form hx-post="/add" hx-target="#todo-list" class="mt-4">
    <div class="input-group">
        <input type="text"
               name="description"
               class="form-control"
               placeholder="New todo item" required>
        <input type="text" name="category" class="form-control" placeholder="Category">
        <button type="submit" class="btn btn-primary">Add</button>
    </div>
</form>
```

```
procedure TTodoWebModule.HandleAddTodo(Request: TWebRequest; Response: TWebResponse);
var
  Description, Category: string;
begin
  Description := Request.ContentFields.Values['description'];
  Category := Request.ContentFields.Values['category'];
  FTodoList.AddItem(Description, Category);
  HandleGetTodoList(Response);
```

```
end;
```

Task Filtering

Let's add the ability to filter tasks by category. Create a new template file named `category-filter.html`:

```
<div class="mb-4">
    <h5>Filter by Category</h5>
    <div class="btn-group" role="group">
        <button class="btn btn-outline-primary"
                hx-get="/" hx-target="#todo-list">All</button>
        @ForEach (var category in Categories) {
            <button class="btn btn-outline-primary"
                    hx-get="/filter/@category" hx-target="#todo-list">@category</button>
        }
    </div>
</div>
```

Update the `todo-list.html` template to include the category filter:

```
@LayoutPage layout.html

@Import category-filter.html

<div id="todo-list">
    <!-- ... existing todo list content ... -->
</div>

<!-- ... existing form ... -->
```

Add a new method to handle filtering in the WebModule:

```
procedure TTodoWebModule.HandleFilterTodos(Request: TWebRequest; Response: TWebResponse);
var
  Category: string;
  FilteredTodos: TArray<TTodoItem>;
begin
  Category := Request.PathInfo.Substring('/filter/'.Length);
```

```
  FilteredTodos := FTodoList.GetItemsByCategory(Category);
  FWebStencilsProcessor.AddVar('Todos', FilteredTodos);
  FWebStencilsProcessor.AddVar('Categories', FTodoList.GetAllCategories);
  FWebStencilsProcessor.InputFileName := 'todo-list.html';
  Response.Content := FWebStencilsProcessor.Content;
end;
```

Update the `WebModuleDefault` method to handle the new filter route:

```
procedure TTodoWebModule.WebModuleDefault(Sender: TObject; Request: TWebRequest;
  Response: TWebResponse; var Handled: Boolean);
begin
  if Request.PathInfo = '' then
    HandleGetTodoList(Response)
  else if Request.PathInfo.StartsWith('/filter/') then
    HandleFilterTodos(Request, Response)
  // ... existing routes ...
end;
```

This guide's main focus is WebStencils and how to use it with RAD Studio. In the previous code, some of the extra required logic for category filtering, etc., that would be just Delphi code not related to WebStencils hasn't been included to simplify the understanding of the code snippets.

Conclusion

In this chapter, we've successfully migrated our To-Do app from using HTML constants to WebStencils templates. This migration has made our code more maintainable and easier to read. We've separated our concerns by moving our HTML into separate template files, which can be easily modified without touching our Delphi code.

We've also demonstrated the scalability of our new approach by adding new features like task categories and filtering. These additions were straightforward to implement thanks to the flexibility of WebStencils templates.

By using WebStencils, we've gained several advantages:

1. **Separation of concerns**: Our HTML is now separate from our Delphi code.

2. **Reusability**: We can easily reuse components like the category filter across different pages.
3. **Maintainability**: Changes to the UI can be made without modifying the Delphi code.
4. **Scalability**: Adding new features is simpler while avoiding code duplication as much as possible.

In the next chapter, we'll explore more advanced WebStencils features and how to integrate them into our application architecture.

08

Advanced Options with WebStencils

Introduction

In this chapter, we'll see more advanced features and options available in WebStencils. These tools will allow you to create more dynamic, efficient, and sophisticated web applications.

@query Keyword

The `@query` keyword allows you to read HTTP query parameters directly in your templates. This is useful for creating dynamic content based on URL parameters.

```
<!--
Example url: https://example.com?searchTerm="searchingString"&page=9&totalPages=34
-->

<h1>Search Results for: @query.searchTerm</h1>
<p>Page @query.page of @query.totalPages</p>
```

@Scaffolding

The `@Scaffolding` keyword provides a powerful way to generate HTML dynamically based on your application's data structures. This can be particularly useful for creating forms or displaying data tables.

```
<form>
  @Scaffolding User
</form>
```

To implement scaffolding, you need to handle the `OnScaffolding` event of your WebStencilsProcessor:

```
procedure TMyWebModule.ProcessorScaffolding(Sender: TObject;
  const AQualifClassName: string; var AReplaceText: string);
begin
  if SameText(AQualifClassName, 'User') then
  begin
    AReplaceText := '';
    // Generate form fields based on User properties
    AReplaceText := AReplaceText + '<input type="text" name="Username"
placeholder="Username">';
    AReplaceText := AReplaceText + '<input type="email" name="Email"
placeholder="Email">';
    // ... add more fields as needed
  end;
end;
```

The above code replaces the Scaffolding keyword with the AReplaceText value.

In this `@Scaffolding` code example, the HTML tags are hardcoded for better understanding, but this keyword should be leveraged using code stored in other classes, records, constants, etc., to improve the structuring and reusability of HTML snippets.

@LoginRequired

The `@LoginRequired` keyword can be used to restrict access to certain parts of your templates based on user authentication status. When this keyword is encountered, it triggers a specific event

that you can handle to check if the user is logged in. The goal is to have pages requiring authorization.

To define if a user is loggedIn or not, WebStencilsProcessor provides a boolean property `UserLoggedIn` that allows storing if the user is successfully logged in or not.

```
@LoginRequired
<div class="protected-content">
  This content is only visible to logged-in users.
</div>
```

You can also specify roles for more granular access control:

```
@LoginRequired(admin)
<div class="admin-panel">
  Admin-only content goes here.
</div>
```

To specify the roles of the current user, the property `UserRoles` can be found in WebStencilsProcessor. In case a user has different roles, define these roles separated by a coma. Example: `sales,management`

An internal event in the Processor will evaluate the roles of the user and raise an `EWebStencilsLoginRequired` exception in case the user doesn't have the specific role if they are not logged in.

Keep in mind that WebStencils doesn't provide a built-in authentication system, and these properties and keywords are meant to be used only if the user request is properly authenticated with an external service.

OnValue Event Handler

The `OnValue` event handler provides a way to supply data to your templates dynamically. This is particularly useful when you need to compute values on the fly or when you want to keep your data access logic separate from your templates.

```
procedure TMyWebModule.ProcessorOnValue(Sender: TObject;
  const ObjectName, FieldName: string; var ReplaceText: string;
  var Handled: Boolean);
begin
  if SameText(ObjectName, 'CurrentTime') then
  begin
    ReplaceText := FormatDateTime('yyyy-mm-dd hh:nn:ss', Now);
    Handled := True;
  end;
end;
```

In your template, you can then use:

```
<p>Current time: @CurrentTime</p>
```

Template Patterns

WebStencils supports several common template patterns that can help you organize your application's views effectively.

Standard Layout

This pattern uses the built-in `@LayoutPage` and `@RenderBody` approach. It's ideal for maintaining a consistent structure across your site while allowing individual pages to provide their specific content. The demos shown so far have followed this pattern.

Header/Body/Footer

In this pattern, each page is an individual template, but they all share common parts like headers and footers. Instead of using a layout page, you might structure your templates like this:

```
@Import Header.html

<main>
    <!-- Page-specific content here -->
</main>

@Import Footer.html
```

This approach offers more flexibility than the standard layout pattern but may require more management of common elements.

Reusable Components

Using the `@Import` directive, you can define individual sets of components that can be reused throughout your application. This directive also allows the passing of iterable objects and even defining aliases for a more agnostic component definition. For example:

```
<div class="product-list">
    @Import ProductList { @list = @ProductList }
</div>

<div class="tasks">
  @ForEach (var Task in Tasks.AllTasks) {
        @Import partials/tasks/item { @Task }
    }
</div>
```

This pattern allows you to break your UI into smaller, manageable pieces that can be easily maintained and reused across different pages. You can find some examples of this pattern in the code of this demo.

Conclusion

These advanced features of WebStencils provide powerful tools for creating dynamic, efficient, and sophisticated web applications. By leveraging these capabilities, you can create more maintainable and flexible code, separate your concerns more effectively, and build robust web applications that can grow and adapt to changing requirements.

In the next chapter, we'll explore how to integrate WebStencils with RAD Server, opening up even more possibilities for building scalable web applications.

09

Using RAD Server Integration with WebStencils

Introduction

RAD Server has also benefited from the new library WebStencils. A specific integration has been developed to allow RAD Server to leverage all the potential of web development. When it comes to syntax, components etc, all we have learnt so far is still applicable in RAD Server.

Integrating WebStencils with RAD Server

As with the previous examples we've seen, there are multiple approaches when it comes to using RAD Server and WebStencils. Let's see two of the most common patterns:

Using WebStencils Processors

This is a simple approach where HTML is generated on demand for each request using processors (created at design time or programmatically) and returned directly in the RAD Server response. The template to be used needs to be read from a file, stored in constants, variables, etc., and then processed.

```
type
  [ResourceName('testfile')]
  TTestResource = class(TDataModule)
    [ResourceSuffix('get', './')]
    [EndpointProduce('get', 'text/html')]
    procedure Get(const AContext: TEndpointContext; const ARequest: TEndpointRequest;
const AResponse: TEndpointResponse);

...

procedure TTestResource.Get(const AContext: TEndpointContext; const ARequest:
TEndpointRequest; const AResponse: TEndpointResponse);
var
  LTemplateFile, LHTMLContent: string;
Begin
  // replace this variable with the real path to the template
  LTemplateFile := 'C:\path\to\your\file.html';
  WebStencilsProcessor.InputFileName := LTemplateFile;
  LHTMLContent := WebStencilsProcessor.Content;
  AResponse.Body.SetString(LHTMLContent);
end;
```

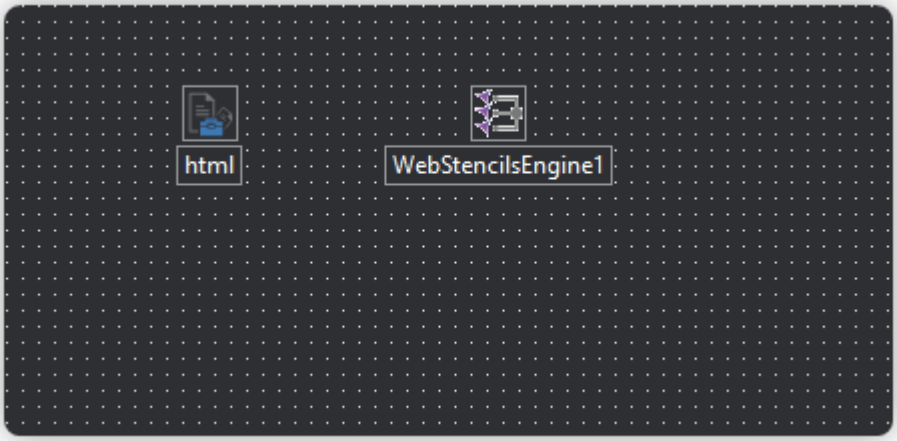
Executing this RAD Server project and opening a browser, you should be able to access the URL http://localhost:8080/testfile and see the rendered content of the template defined in the variable `LFilePath`.

Note

For RAD Server, template paths must be absolute. It's not possible to use relative ones due to the way that RAD Server is being deployed in production using Apache or IIS.

Using WebStencils Engine

For this option, we recommend combining the existing `TEMSFileResource` component with a `TWebStencilsEngine` component. The first does the mapping to the file system, while the second manages the HTTP mapping and the template processing. The components are connected using the Dispatcher property of the WebStencils Engine.



To configure the `TEMSFileResource` correctly, we need to specify in this one where the templates are using the property `PathTemplate`.

`C:\path\to\your\templates\{filename}`

Pay special attention to the `{filename}` wildcard, as it will reference each page.

Once the components are configured, `PathsTemplates` can be defined in the Engine the same way we've seen previously using the wildcard `{fileName}` or the names of the template files.

To map the files correctly, we need to define some attributes on the FileResource as we did in the previous example:

```
type
  [ResourceName('testfile')]
  TTestfileResource1 = class(TDataModule)
    [ResourceSuffix('./')]
    [ResourceSuffix('get', './{filename}')]
    [EndpointProduce('get', 'text/html')]
    html: TEMSFileResource;
```

Given this example, we can now access to the endpoint http://localhost:8080/testfile/{filename} where `{filename}` is any of the templates stored in the path we defined in the `PathTemplate` property.

If the same `TWebStencilsEngine` needs more than one `TEMSFileResource`, the global method `AddProcessor` can assign extra ones.
Example:

```
AddProcessor(FileResourceResource, WebStencilsEngine1);
```

Recreating the Tasks app to RAD Server

RAD Server works a bit differently compared to WebBroker. Being a REST-compliant application, it doesn't have a memory state, so a few things need to be taken into account.

If you have checked the WebBroker demo included in the GitHub repository you'll see that it follows a MVC approach. In the same repository you can find the same demo fully migrated to RAD Server and offering the same functionality. Let's enumerate what was needed to be refactored:

Database management

In the WebBroker demo, because the tasks are stored in memory, a singleton pattern was used to avoid issues with multi-threading. In RAD Server, the tasks can't be stored in memory (at least not easily), so we will use an InterBase database instead.

To access the database from the model instead of defining it directly in it, the `TFDConnection` component is assigned in the `TTasks` model constructor. For demo simplification, the `TFDConnection` component was created in the same RAD server DataModule. In a production app, using the `TFDManager` component would be recommended.

Controller arguments

WebBroker and RAD Server requests and responses are slightly different, so some refactoring in the arguments passed to the controller's methods was needed.

From Actions to Endpoints

WebBroker uses Actions to define endpoints and the business logic associated with them. With RAD Server, methods inside a resource must be defined with the specific URI associated with the method using attributes.

Processing data on the requests

RAD Server works with JSON. Sending data from our To-Do app to our backend, as well as processing that data, required some changes.

1. **Use of the HTMX extension JSON-enc**: This extension encodes the requests sent to the backend in JSON format instead of form-data (default behavior). To use this extension, a

simple `<script>` tag has to be added, and the attribute `hx-ext="json-enc"` must be specified on the same tag where we define the `hx-post` or `hx-put` request.

2. **Modification of data processing**: Because RAD Server processes request data as JSON, some modifications were needed to get the values from the request. Using the standard JSON library is sufficient for the purpose of this demo.

Handling static JS, CSS and images

The `TEMSFileResource` component needs to be used to deliver static files. Independent components were created to map each folder where those files (JS, CSS, and images) are.

Frontend sources

RAD Server relies on resources, which means that our main URL will have the name of the resource added to the end. e.g. - https://localhost:8080/web.

The endpoints created by the WebBroker website have been migrated to RAD Server, but this required slight modifications to the WebStencils templates to point to the newly created resource.

10

Resources and Further Learning

Here are a few useful resources for extending the potential of the topics described in this book even further:

Documentation and links

Official HTMX documentation (HTMX.org)

Even though this book addresses the most common keywords, HTMX can add much more to your projects. The documentation the HTMX team offers is extensive but approachable, and not overwhelming.

As well as the official documentation, the htmx.org website offers essays and examples.

- Official Docs
- Examples
- Essays

Delphi and HTMX (Embarcadero Blog)

In this series of posts, a first approach to HTMX and WebBroker was introduced. Although most of the concepts have been discussed in this book, more code examples and other integrations are also shown: e.g. WordPress integration with HTMX.

- Harnessing the power of the Web with Delphi & HTMX
- Harnessing the power of the Web with Delphi & HTMX – Part 2
- Running Delphi & HTMX in WordPress – HTMX series part 3

RAD Server Technical Guide

If you are not familiar with RAD Server and want to explore all its possibilities, a book is available. It will guide you through the main functionalities as well as more specific and advanced ones.

Explore it here

HTMX in an MVC Pattern (HTMX.org)

This page gives a concise explanation of how HTMX fits into an MVC-style web application. It provides examples of thin controllers and how to structure the MVC flow for web development using HTMX. While not Delphi-specific, the concepts are broadly applicable to MVC design patterns in various languages.

Explore it here

WebStencils (DocWiki)

Check out our official WebStencils documentation in the DocWiki.

Explore it here

UI/CSS libraries

This space is massive and there are multiple libraries out there. Given the UI-agnostic nature of WebStencils and HTMX, virtually any library can be used. Here are a few that work nicely and have great documentation.

- Bootstrap
- BeerCSS
- PicoCSS
- Bulma
- DaisyUI
- TailwindCSS

TailwindCSS is a very heavy library size-wise, and including it through CDN adds a very heavy initial payload. Their CLI tool to remove all unused classes should be used in production environments.

Extending HTMX even further

Even though the declarative nature of HTMX makes the projects much less JS-dependent, there are scenarios where JS will be required for pure client-side interactivity. For example, consider changing a website from light to dark mode. We could manage this in the backend and send back the new HTML in dark mode, but thanks to modern CSS libraries, that can be easily done on the client side.

When we require extra functionality and interaction on the client side, there are different micro JS libraries that integrate very nicely with HTMX and WebStencils.

AlpineJS

AlpineJS is a lightweight JavaScript framework designed to add simple interactions to HTML. It provides a declarative way to manipulate DOM elements without writing extensive JavaScript code. It's often compared to frameworks like Vue or React, but it's much smaller and easier to integrate into existing HTML. AlpineJS is ideal for adding JavaScript behavior to web pages without the complexity of a larger framework.

- Use cases: Modals, dropdowns, toggles, form validation, and other UI interactivity.
- Key features: Declarative syntax, reactive data, directives for DOM manipulation.

Documentation: AlpineJS Docs

Hyperscript

Hyperscript is a scripting language designed to simplify event handling and logic in HTML. Unlike JavaScript, which requires more verbose syntax, Hyperscript is designed to be embedded directly into HTML attributes. It focuses on making event-driven programming more intuitive by using natural language-like syntax, allowing developers to create dynamic behavior without complex JavaScript.

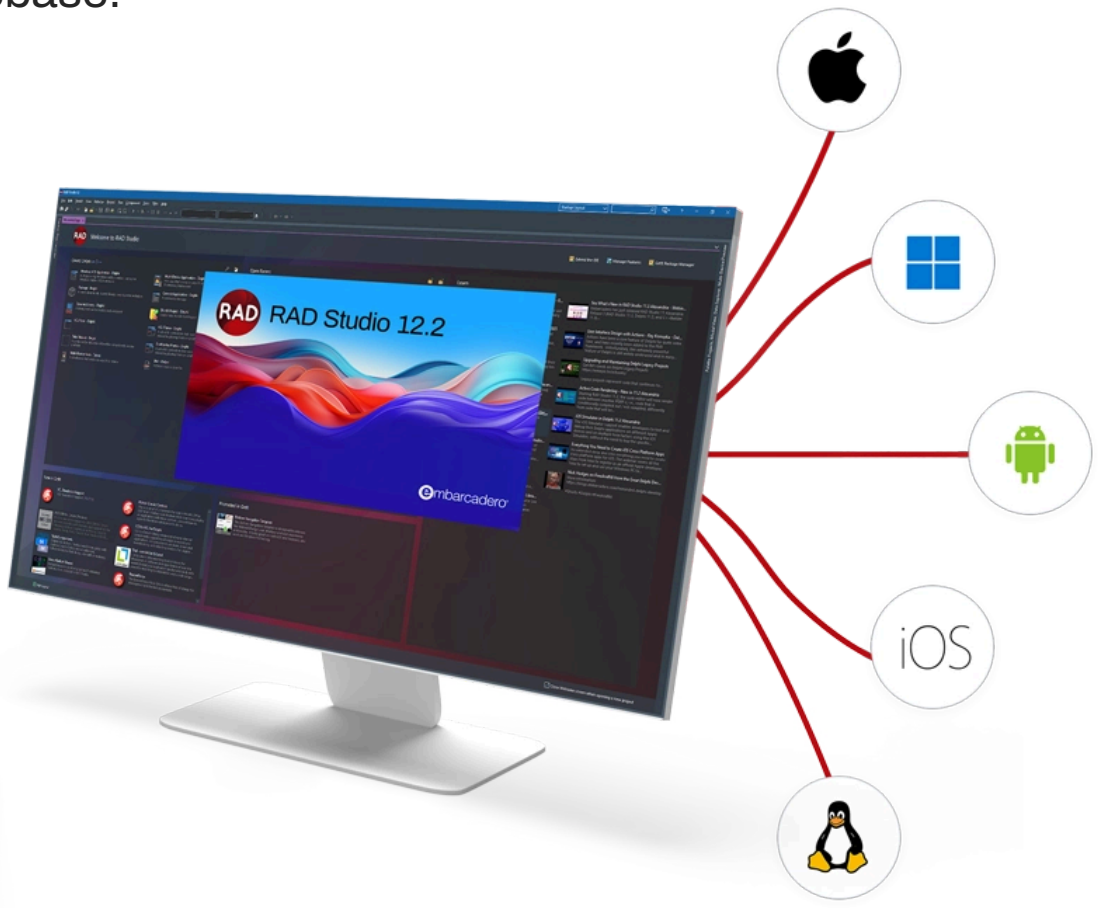
- Use cases: Interactive buttons, form submission handling, element visibility toggles.
- Key features: Natural language syntax, declarative events, simplified handling of common interactions.

Documentation: Hyperscript Docs

Try RAD Studio Now!

Discover how easy it is to target multiple platforms with native apps using just one codebase!

www.embarcadero.com



embarcadero