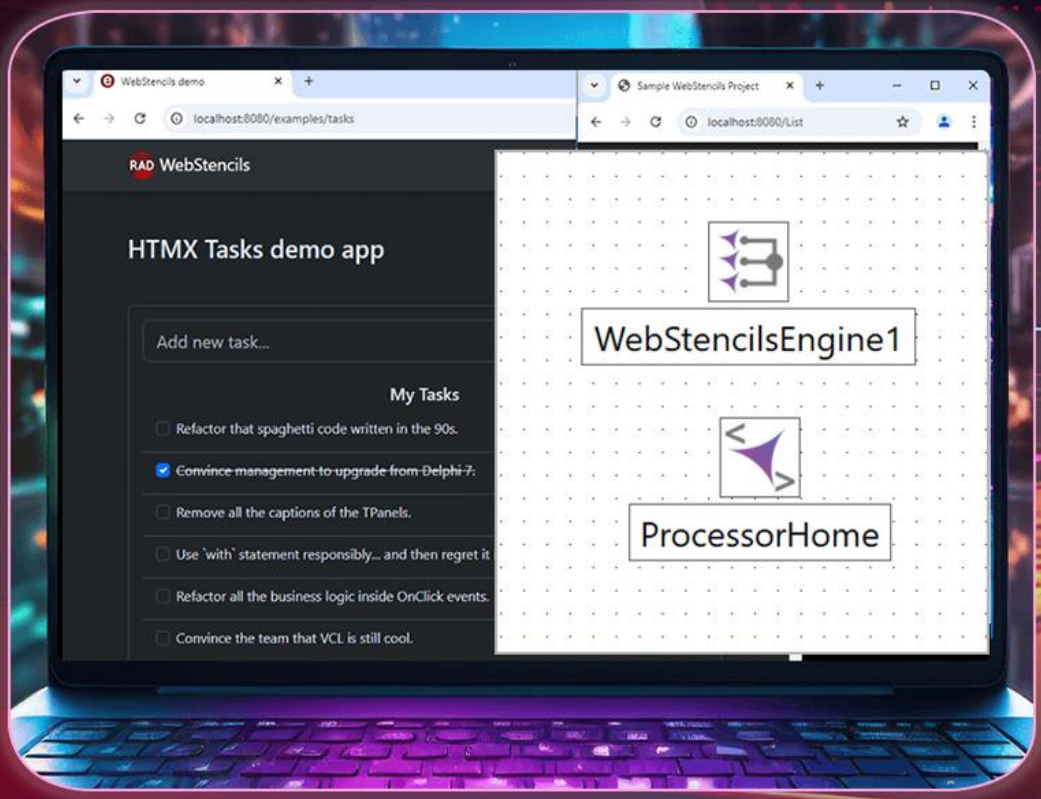
HTMX & WebStencils

使用 RAD Studio 进行快速 Web 开发



Author
Antonio Zapater

embarcadero

HTMX 和 WebStencils

使用 RAD Studio 进行快速 Web 开发

内容

前言

本书重点介绍如何使用 HTMX 和 WebStencils 进行现代化、简易化方法的 Web 开发.

HTMX 是用于建立动态 Web 用户接口的轻量级 JavaScript 替代方案，并且正在成为 Web 开发人员的首选解决方案，因为它可以帮助他们显着减少需要编写的 JavaScript 程序代码数量，使开发过程更快、更直观、更易于阅读和除错并且更容易维护.

HTMX 的简单性与 RAD Studio 的快速应用程序开发精神完美契合，使开发人员能够更专注于应用程序逻辑，而不是为复杂的前端程序代码而苦苦挣扎.

WebStencils 的美妙之处在于其模板驱动的架构。开发人员无需重新发明轮子，而是可以透过可重复使用且可自定义的模板公开现有业务逻辑，这些模板与现有应用程序可无缝集成，从而减少将旧项目引入网路的摩擦. 这不仅加速了开发，还增强了开发团队之间的协作，使他们能够与现有程序代码库更紧密地合作.

透过阅读本书，您将了解如何利用 HTMX 和 WebStencils 的强大功能，以更少的精力和更大的灵活性开发现代化 Web 应用程序. 无论您是致力于增强现有的 Web 桌面应用程序还是建立新的动态 Web 项目，本书都提供了实用的见解，帮助您充分利用 RAD Studio 不断发展的 Web 开发生态系统.

您可以了解有关 RAD Studio 的更多信息，或者下载免费试用版以与本书中的范例一起进行编码 https://www.embarcadero.com/products/rad-studio

让我们深入了解这些技术如何简化您的工作流程并将您的 Web 开发项目提升到新的水平!


备注

遵循与本指南中讨论的范例类似的模式的 WebStencils 和 HTMX 程序代码范例可在此 GitHub 储存库中找到.
https://github.com/Embarcadero/WebStencilsDemos

01

HTMX 简介

什么是 HTMX?

简要概述

HTMX 将 HTML 扩展为超文本介质，能让您发出 AJAX 请求、触发 CSS 转换、建立 WebSocket 并直接从 HTML 元素利用服务器发送事件 (SSE). 这种方法减少了对自定义 JavaScript 的需求，并允许进行更具声明性、以 HTML 为中心的开发.

HTMX 的主要特点包括:

- **简单**: HTMX 使用 HTML 属性来描述行为，使其易于理解和维护.
- **强大**: 尽管 HTMX 很简单，但它允许复杂的互动和实时更新.
- **弹性**: 它可以与任何后端技术一起使用，并与现有系统很好地集成.
- **效率**: HTMX 轻量且快速，可提高加载时间和运行时间效能.

与传统 JavaScript 和 AJAX 的比较

传统的 Web 开发通常需要编写大量 JavaScript 来处理使用者互动、发出 AJAX 请求以及更新 DOM. 这种方法可能会导致程序代码复杂且难以维护，尤其是当应用程序规模和复杂性不断增加时.

HTMX 采用了不同的方法:

- **声明式与命令式**: 您无需编写 JavaScript 函式来描述如何取得和更新内容，而是使用属性直接在 HTML 中声明它们.
- **减少样板文件**: 使用 AJAX 呼叫的结果更新 div 等常见模式只需要使用 HTMX 编写最少的程序代码.
- **提高可读性**: 元素的行为在 HTML 中直接可见，一目了然.
- **易于维护**: 由于行为直接与 HTML 元素相关，更新和维护基于 HTMX 的程序代码通常更容易.

这是一个简单的例子来说明差异:

传统 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>
```

正如你所看到的，HTMX 版本更加简洁，直接从 HTML 中意图更加清晰.

核心概念

为了有效地使用 HTMX，您必须了解其核心概念以及它们如何协同工作来创建动态 Web 应用程序.

hx-get: 透过 GET 请求获取内容

hx-get 属性用于向服务器发出 GET 请求并使用响应更新页面. 当用户与具有 hx-get 属性的元素互动时（在默认情况下，即点击一下时），HTMX 将向指定的 URL 发出 GET 请求，并使用响应更新页面.

范例:

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

在此范例中，当点击一下按钮时，HTMX 将向"/api/user"发出 GET 请求，并将回应放入 id 为"user-info"的 div 中.

hx-target: 指定响应的目标元素

如前面的范例所示, hx-target 属性指定应使用服务器的响应更新哪个元素。如果不指定，默认更新触发请求的元素.

范例:

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

在这个范例中，来自"/api/notification"的回应将插入到 id 为"notification-area"的 div 中.

hx-post: 透过 POST 请求提交数据

与 hx-get 类似, hx-post 属性用于发出 POST 请求。这通常用于提交窗体数据或当您需要向服务器发送数据从而导致服务器状态变更时.

范例:

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

当提交此窗体时，HTMX 将使用窗体数据向"/api/submit"发出 POST 请求，并将回应放在 #response div 中.

hx-put, hx-patch, hx-delete 请求

这些其他请求遵循与 hx-get 和 hx-post 相同的约定. 表示 hx-put, hx-patch 和 hx-delete 分别发送 PUT、PATCH、DELETE 请求. 它们用于向后端提交修改或删除（我们稍后会更详细地看到它们）.

hx-swap: 控制内容的交换方式

hx-swap 属性可让您控制如何将新内容填入目标元素中。最常见的选项是:

- innerHTML (内定): 替换目标元素的内部 HTML
- outerHTML: 替换整个目标元素
- beforebegin: 插入到目标元素之前
- afterbegin: 作为目标元素的第一个子元素插入
- beforeend: 作为目标元素的最后一个子元素插入
- afterend: 插入到目标元素之后

范例:

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

这会将新项目附加到列表末尾，而不是替换整个列表.

其他核心概念

虽然上述四个概念构成了 HTMX 的基础，但还有其他几个重要功能要注意:

1. hx-trigger: 允许您指定触发 AJAX 请求的事件。默认情况下，它是大多数元素的 Click 事件或窗体的 Submit 事件.
2. hx-params: 控制随请求提交的参数.
3. hx-headers: 允许您向 AJAX 请求新增自定义标头.
4. hx-vals: 允许您向随请求提交的参数中添加附加数值.
5. hx-boost: 使用 AJAX 增强普通锚点和窗体的简单方法.
6. hx-push-url: 将新的 URL 推送到历史堆栈中，允许更新浏览器 URL，而无需加载整个页面.

了解这些核心概念为使用 HTMX 建立动态、交互式 Web 应用程序奠定了坚实的基础. 随着您对基础知识越来越熟悉，您将能够利用更高级的 HTMX 功能，以最少的 JavaScript 创建复杂的响应式用户接口.

有关 HTMX 的更深入文档，请访问此连结的官方文档: https://htmx.org/docs

02

简介 WebBroker

什么是 WebBroker?

WebBroker 是 RAD Studio 中的一个强大的框架，使开发人员能够创建强大的 Web 应用程序和 Web 服务。它为建立服务器端应用程序提供了坚实的基础，并支持 RESTful 服务、SOAP 服务等的开发.

WebBroker 的主要特点

1. **多功能性**: WebBroker 支持各种类型的 Web 服务（独立、Apache、ISAPI…），使其成为满足不同项目需求的多功能选择.
2. **集成性**: 它与 RAD Studio 集成，为 Delphi 和 C++Builder 开发人员提供熟悉的开发环境.
3. **延展性**: WebBroker 应用程序可以轻松扩展，以处理不断增加的负载和复杂的要求.
4. **效率**: 该框架专为高效能而设计，这对于服务器端应用程序至关重要.

核心概念

了解 WebBroker 的核心概念对于有效使用 WebStencils 建立 Web 应用程序和服务至关重要.

组件和架构

WebBroker 应用程序是使用一组关键组件建构的，这些组件协同工作来管理 HTTP 请求、响应、路由和中间件。主要组成部分包括:

1. TWebModule: 这是 WebBroker 应用程序的核心组件。它可作为其他 WebBroker 组件的容器并处理应用程序的整体流程.
2. TWebDispatcher: 此组件负责将传入的 HTTP 请求分派到适当的操作项.
3. TWebActionItem: 此类别定义应如何处理特定 URL 端点。每个操作项都与特定的 URL 模式相关联，并定义请求该 URL 时应该执行的相应程序代码.

WebBroker 应用程序的架构通常遵循此流程:

1. HTTP 请求进来
2. TWebDispatcher 将请求路由到适当的 TWebActionItem
3. TWebActionItem 执行其关联程序代码
4. 产生响应并发回客户端

创建 WebBroker 应用程序

要创建 WebBroker 应用程序:

1. 选择"File/New/Other…/Web/Web Server Application".
2. 如果您愿意，可以选择 Linux 兼容性.
3. 选择应用程序类型：独立、Apache 模块等.
4. 除非端口 8080 在您的计算机上本机使用，否则请使用此默认值.

我们可以将 WebModule 操作视为您的路由器，其中将定义所有端点.

若要设定 TWebModule 并新增 TWebActionItems 来处理不同的 URL 端点，请单击 TWebModule 中的任何位置，然后在属性选单上选择 Actions. 在出现的选单中，您将能够建立新的端点以及与其关联的方法。与大多数 RAD Studio 组件一样，这些操作项有多个可用事件.

以下是如何设定 TWebActionItem 的简单范例:

```
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;
```

当请求其关联的 URL 时，此操作项目将产生一个简单的 HTML 回应.

处理请求和响应

WebBroker 提供了一个简单的机制来处理 HTTP 请求和响应:

- TWebRequest 提供对传入 HTTP 请求的所有信息的访问，包括参数、标头和请求方法.
- TWebResponse 用于设定响应数据，包括内容、状态代码和标头.

存取请求数据和设定响应数据的范例:

```
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;
```

部署和可扩充性

WebBroker 应用程序提供灵活的部署选项:

1. **独立的可执行文件案**: 可以作为独立的 Web 服务器运作（基于窗体或 CLI）.
2. **Apache/ISAPI 扩充模块**: 可以与 IIS 或 Apache 等 Web 服务器集成.

这种灵活性允许各种部署方案，确保不同类型 Web 服务的可扩充性和效能. 您可以从独立的可执行文件开始进行开发和测试，然后将其部署为 ISAPI 扩充功能或 Apache 模块，以便与全功能 Web 服务器一起用于生产环境.

WebBroker 服务中的会话管理

虽然 WebBroker 服务不提供内建会话管理，但可以在您的 WebBroker 服务应用程序中实现强大的会话处理系统. 会话管理对于跨多个请求维护使用者状态至关重要，并且通常是使用者身份验证、购物车或 CSRF 保护等功能所必需的.

会话管理方法

有多种方法可以实现会话管理:

1. **在内存中实作会话管理**: 将会话数据储存在内存中。这很快速，但如果服务器重新启动则不会持续存在并且不能很好地扩展到多个服务器实例.
2. **在数据库中实作会话管理**: 将会话数据储存在数据库中。这允许持久性和可扩展性，但可能会引入延迟.
3. **使用档案文件实作会话管理**: 将会话数据储存在服务器上的档案中。这提供了持久性，但可能会出现大量并发会话的效能或锁定问题.
4. **使用分布式快取实作会话管理**: 使用 Redis 等分布式快取进行会话储存。这提供了性能和可扩展性的良好平衡.

实作内存中会话管理

L 让我们看看内存中会话管理的基本实现:

```
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.
```

将会话管理集成到 WebModule 中

若要在 WebBroker 应用程序中使用会话管理，您可以建立一个包含会话处理的 WebModule 基类别:

```
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;
```

透过此设置，您可以轻松存取 WebModule 的操作处理程序中的会话数据:

```
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

请记住，这个 SessionManager 实作只是一个概念证明. 要在生产环境中使用它，需要进一步的实作，例如删除过期的会话、更独立地实现 TSessionData 或将 SessionManager 作为多线程环境的单例(singleton)处理.

安全考虑

CSRF 保护

跨站点请求伪造 (CSRF) 是一种攻击，其中 Web 应用程序信任的用户发送未经授权的命令。要防范 CSRF:

1. 使用请求验证令牌: 为每个会话产生唯一的令牌并将其包含在窗体中.
2. 验证服务器上每个非 GET 请求的令牌.

WebBroker 服务中的实施范例:

```
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.
```

在这个实作中:

1. 我们为每个窗体提交产生一个唯一的 CSRF 令牌.
2. 令牌储存在会话中并作为隐藏域包含在窗体中.
3. 处理窗体提交时，我们根据会话中储存的令牌验证请求中的令牌.

数据验证

即使客户端验证已到位，也始终在服务器端验证和清理数据.

范例:

```
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;
```

跨站脚本 (XSS)

为了防止 XSS 攻击，请务必先对用户产生的内容进行编码或转义，然后再将其包含在 HTML 响应中. NetEncoding 函式库提供了多种转义 HTML 字符串的集成方法.

范例:

```
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;
```

其他安全考虑因素

1. **SQL 注入预防**: 与数据库互动时使用参数化查询或准备好的语句.
2. **安全通讯**: 使用 HTTPS 对传输中的数据进行加密.
3. **认证与授权**: 实施强大的使用者身份验证和适当的访问控制.

实施这些安全措施可以显着增强您的 WebBroker 应用程序的安全性. 请记住，安全是一个持续的过程，及时了解最新的安全最佳实践和漏洞是非常重要的.

您可以在 docwiki 中查看更详细的信息: 使用 WebBroker

03

使用 WebBroker 开发您的第一个 Web 应用程序

简介

在本章中，我们将逐步介绍使用 WebBroker 建立第一个 Web 应用程序的过程。我们将从一个简单的“Hello World”范例开始，然后继续创建一个基本的待办事项应用程序.

创建“Hello World”应用程序

让我们从一个简单的「Hello World」应用程序开始，该应用程序示范了 HTMX 与 WebBroker 的使用.

1. 首先在 RAD Studio 中建立一个新的 WebBroker 应用程序.
2. 在您的 WebModule 中，新增一个新的 WebActionItem 并将其名称设为“ActionHello”.
3. 双击 ActionHello 项目建立事件处理程序.
4. 实作事件处理程序如下:

```
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. 新增另一个名为"ActionGreet"的 WebActionItem 并实作其事件处理程序:

```
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. 运行您的应用程序。当您在网页浏览器中开启它并点击“Say Hello”按钮时，您应该会看到问候语出现，而无需重载整个页面.

这个简单的范例示范了 WebBroker 如何提供 HTML 内容以及如何使用 HTMX 向服务器发出动态请求.

基本待办事项应用程序

现在让我们创建一个更复杂的应用程序：一个基本的待办事项应用程序，允许用户添加和查看任务.

1. 建立一个新的 WebBroker 应用程序或使用现有的 WebBroker 应用程序.
2. 在您的项目中新增一个单元来储存待办事项列表:

```
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. 在您的 WebModule 中，为 TodoList 新增一个字段:

```
FTodoList: TTodoList;
```

在 WebModule 的建构函式中初始化它并在解构函数中释放它.

4. 新增 WebActionItems 用于显示待办事项列表、新增项目和更新列表。依下列方式实现其事件处理程序:

```
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. 运行您的应用程序。现在您应该能够添加新的待办事项并查看列表动态更新，而无需重新加载整页.

这个基本的待办事项应用程序示范如何使用 WebBroker 处理不同类型的请求（GET 用于显示列表，POST 用于新增项目）以及如何使用 HTMX 建立与服务器互动的动态用户接口.

请记住，在现实应用程序中，您需要新增错误处理、输入验证，并可能将待办事项列表储存到数据库中. 但此范例是了解 WebBroker 和 HTMX 如何协同建立交互式 Web 应用程序的良好起点.

04

使用 HTMX 的高阶属性和安全性

简介

在本章中，我们将探讨 HTMX 中的一些更进阶的属性，并讨论使用 HTMX 和 WebBroker 建置 Web 应用程序时的重要安全注意事项.

高阶属性

HTMX 提供了一组丰富的属性，允许对 AJAX 请求和 DOM 更新进行细微控制。让我们来探索一些更高级的属性:

hx-put 和 hx-delete: 透过 PUT 和 DELETE 提交请求

虽然 hx-get 和 hx-post 很常用, 但 HTMX 也支持其他 HTTP 方法，例如 PUT 和 DELETE.

使用 hx-put 范例:

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

使用 hx-delete:范例:

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

在您的 WebBroker 应用程序中，您需要适当地处理这些请求:

```
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: 自定义事件触发器

hx-trigger 属性可让您指定触发 AJAX 请求的事件。默认情况下，大多数元素为'Click'，窗体为'Submit'.

范例:

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

这将在用户停止输入后 500 毫秒触发搜寻请求.

hx-select: 选择服务器响应的部分内容

hx-范例属性可让您选择服务器响应的子集来更新目标.

Example:

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

在这种情况下，只有服务器回应中 id 为"name"的元素将用于更新目标.

这种方法允许您的服务器返回完整的 HTML 页面（这对于非 HTMX 请求或调试可能有用），同时仍然使 HTMX 能够选择性地仅更新页面的部分内容.

值得注意的是，虽然传回完整的 HTML 文件是可能的且有时很有用，但在许多情况下使用 HTMX 时，您可能会选择让服务器只传回所需的特定 HTML 片段内容.

hx-include: 在请求中包含附加数据

hx-include 允许您在请求中包含其他元素的值.

范例:

```
<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">
```

这将在窗体提交中将"extra-dat"HTML 输入元素的值包含在其中.

hx-push-url: 更新浏览器的 URL

hx-push-url 允许您在不加载整个页面的情况下更新浏览器的 URL，这对于维护单页应用程序中的可导航性很有用.

范例:

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

单击按钮时，这将更新浏览器的 URL，从而实现正确的后退/前进导航.

05

简介 WebStencils

什么是 WebStencils?

WebStencils 是一种新的基于服务器端脚本的 HTML 文件集成和处理技术, 在 RAD Studio 12.2 中引入. 它允许开发人员基于任何 JavaScript 库创建现代、专业的网站，并由 RAD Studio 服务器端应用程序提取和处理的数据提供支持.

核心理念

WebStencils 支持开发导航网站，例如部落格、在线目录、订购系统、参考网站,例如字典和 wikis. 它提供了一个类似于 ASP.NET Razor 处理但专为 RAD Studio 设计的模板引擎.

与 HTMX 集成

WebStencils 很好地补充了 HTMX。 HTMX 页面可以受益于服务器端程序代码产生并连接到 REST 服务器以进行内容更新。同时，Delphi Web 技术可以提供高质量的页面产生和 REST API.

CSS 和 JS 中立性

WebStencils 的主要功能之一是它不会强迫您使用任何特定的 JavaScript 或 CSS 链接库. 它纯粹是一个用于服务器端渲染的模板引擎，允许您使用任何您喜欢的前端技术.

WebStencils 语法

WebStencils 使用基于两个主要元素的简单语法:

1. @ 符号
2. 区块的大括号 {}

@符号

@符号在 WebStencils 中用作特殊标记。接在其后的可以是:

- 对象或字段的名称
- 特殊处理关键词
- 另外一个 @

例如:

```
@object.value
```

此语法存取“object”的“value”属性值. 对象名称是一个本地符号名称，它可以匹配实际的服务器应用程序对象或在处理 OnValue 事件处理程序时在程序代码中解析.

区块的大括号{}

大括号用于表示条件区块或重复区块。它们仅在特定 WebStencils 条件语句之后使用时才会被处理.

使用点符号存取数值

以下是如何在 WebStencils 中存取数值的范例:

```
<h2>User Profile</h2>
<p>Name: @user.name</p>
<p>Email: @user.email</p>
```

WebStencils 关键词和范例

让我们透过范例探索各种 WebStencils 关键词:

@page

@page k 关键词允许从页面存取多个属性以及存取连接.

范例:

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

@query

@query 关键词用于读取 HTTP 查询参数.

范例:

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

在此范例中，searchTerm 将是 URL 中包含的参数: yourdomain.com?searchTerm=”mySearch”

批注 (@* .. *@)

WebStencil 中的批注包含在@* *@中，并且在产生的 HTML 中被省略.

范例:

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

@if 和@else

条件执行的处理方式是使用@if 和 @else.

范例:

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

@if not

对于否定条件执行，请使用@if not.

范例:

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

@ForEach

@ForEach 关键词用于迭代列举器中的元素.

范例:

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

结论

WebStencils 提供了一种在 RAD Studio 应用程序中产生动态网页的强大方法. 其语法允许将服务器端逻辑无缝集成到 HTML 模板中. 藉由使用 @ 符号, 大括号和各种关键词，您可以轻松建立动态和交互式网页.

随着您越来越熟悉 WebStencils，您将能够充分利用其潜力来创建复杂且响应迅速的 Web 应用程序. 在接下来的章节中，我们将探索 WebStencils 的更多进阶功能，包括模板、布局以及如何有效使用 WebStencils 组件.

06

组件和布局选项

简介

在本章中，我们将探索 WebStencils 的核心组件，使用模板和布局，并讨论常见的模板模式. 了解这些概念将使您能够使用 WebStencils 创建更有组织、可维护和可重复使用的 Web 应用程序.

WebStencils 组件

WebStencils 引入了两个主要组件：WebStencils 引擎和 WebStencils 处理器，它们协同工作来处理您的模板并产生所需的最终 HTML 输出.

WebStencils 引擎

WebStencils 引擎是管理模板整体处理的核心组件。它可以用于两个主要场景:

1. **连接到 WebStencilsProcessor 组件**: 在此设定中，引擎为多个处理器提供共享设定和行为，从而减少了单独自定义每个处理器的需要.
2. **独立使用**: 引擎可根据需要建立 WebStencilsProcessor 组件，让您仅将引擎组件放置在 Web 模块上.

TWebStencilsEngine 的关键属性和方法包含了:

- Dispatcher: 指定档案调度程序(实作 IWebDispatch) 用于文本文件的后处理.
- PathTemplates: 用于匹配和处理请求的请求路径模板集合.
- RootDirectory: 指定相对档案路径的文件系统根路径.
- DefaultFileExt: 设定预设档案扩展名(预设为“.html”).
- AddVar: 将对象新增至处理器可用的脚本变量列表中.
- AddModule: 扫描对象中标记有[WebStencilsVar] 属性的成员并将它们新增为脚本变数.

WebStencils 处理器

WebStencils 处理器负责处理单一档案(通常是 HTML)及其关联的模板. 它可以独立使用，也可以由 WebStencils 引擎建立和管理.

TWebStencilsProcessor 的关键属性和方法包括:

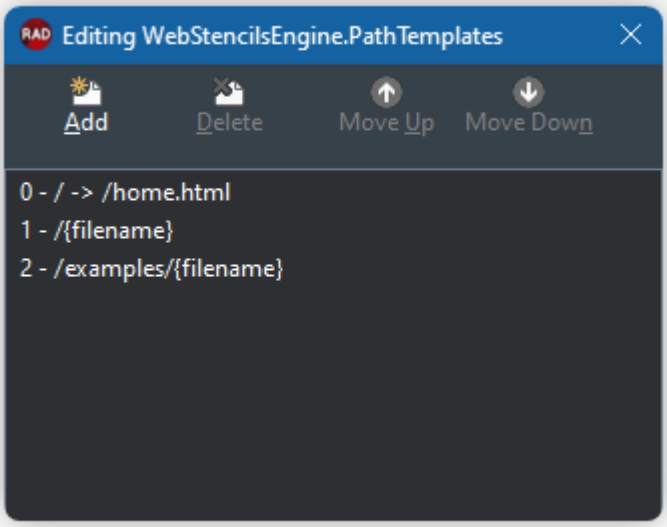
- InputFilename: 指定要处理的文件档案.
- InputLines: 允许直接分配指定要处理的内容.
- Engine: 指定引擎继承数据变量、事件处理程序等(可选).
- Content: 产生最终的处理内容.
- AddVar: 将对象新增至处理器可用的脚本变量列表中.

TWebStencilsEngine 和 WebBroker

WebStencils 引擎组件可以轻松链接到 TWebFileDispatcher，以使用 WebBroker 自动交付模板.

配置完成后，可以在 PathTemplates 属性中使用通配符将请求自动对应到文件.

范例:



让我们来分析一下定义的每条路径:

0-/->/home.html: 将网站根目录的存取权重新导向到模板 home.html

1-/{filename}:引擎将尝试将 URI 映像到文件名. 例如，访问 https://localhost:8080/basics 将在预定义模板文件夹中搜寻名为 basics.html 的模板.

2-/examples/{filename}:与前面的范例相同的行为，但在这种情况下，将在路径/example 中搜寻模板

使用 AddVar 新增数据

AddVar 方法对于将数据从 Delphi 程序代码传递到 WebStencils 模板至关重要。有多种使用 AddVar 的方法:

1. **直接对象赋值**:

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

2. **使用匿名方法**:

```
WebStencilsProcessor1.AddVar('products',
  function: TObject
  begin
    Result := GetProductList;
```

```
  end);
```

3. **使用属性**: 您可以使用[WebStencilsVar] 属性标记类别中的字段、属性或方法, 然后使用 AddModule 将所有标记的成员新增为脚本变量:

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

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

重要: 请记住，只有具有 GetEnumerator 方法(其中列举器传回对象值)的对象才有效。无法从 WebStencils 使用记录(Record).

版面和内容占位符号

WebStencils 提供了一个强大的布局系统，类似于 Mustache、Blade、ERB 或 Razor 等其他模板引擎. 该系统允许您为页面定义通用结构并将特定内容注入该结构.

@RenderBody

@RenderBody d 指令在布局模板中用于指示特定页面的内容应插入到何处。例如，假设我们有一个像这样的通用 HTML 结构，我们称之为 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>
```

@Renderbody 关键词将被替换为将包含在其他子模板中的内容.

@LayoutPage

@LayoutPage 指令在内容页面中使用来指定应使用哪个布局模板作为该页面的结构。它通常放置在内容文件的顶部:

```
<!-- 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>
```

在这个范例中，BaseTemplate.html 将用作基本布局，关键词 @LayoutPage 下的内容将渲染在我们在上一个范例中定义的 @RenderBody 位置中.

@Import

@Import 指令允许您将外部文件合并到目前模板中的特定位置。这对于创建可重复使用的组件很有用.

该指令允许您在嵌套文件夹中建立模板。您也可以省略档案扩展名，只要引擎或处理器中已定义预设扩展名即可.

```
@Import Sidebar.html
```

```
@* Same Behavior *@
@Import Sidebar

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

@ExtraHeader 和@RenderHeader

@ExtraHeader 指令可让您定义应放置在 HTML 文件部分中的附加内容。这对于包含特定于页面的 CSS 或 JavaScript 档案特别有用.

这是它的工作原理:

1. 在你的内容页面中，你使用@ExtraHeader 来定义额外的头部内容.
2. 在您的版面配置模板中，使用 @RenderHeader 来指定应插入此额外标题内容的位置.

让我们来看一个例子:

内容页(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>
```

布局模板(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>
```

在此范例中，产品页面定义了特定于产品详细信息页面的额外 CSS 和 JavaScript 文件. @RenderHeader 布局模板中的指令确保这些额外资源包含在最终的 HTML 输出中.

这种方法允许您拥有一个通用的基本模板，同时仍允许各个页面根据需要添加自己的资源或元标记.

请记住，如果需要，您可以在内容页面中拥有多个 @ExtraHeader 区块. 它们都将在布局模板中放置 @RenderHeader 的位置进行渲染.

透过一起使用@LayoutPage、@RenderBody、@ExtraHeader 和@RenderHeader，您可以创建高度灵活且可维护的模板结构.

模板模式

使用 WebStencils，您可以应用几种常见的模板模式来更轻松地组织应用程序的视图.

标准布局

此模式使用我们已经讨论过的内建 @LayoutPage 和 @RenderBody 方法. 它非常适合在整个网站上保持一致的结构，同时允许各个页面提供其特定内容.

Header/Body/Footer

在此模式中，每个页面都是一个单独的模板，但它们都共享公共部分，例如页首和页尾。您可以像这样建立模板，而不是使用布局页面:

```
@Import Header.html

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

```
@Import Footer.html
```

这种方法比标准布局模式提供了更大的灵活性，但可能需要对公共元素进行更多管理.

可重复使用的组件

使用 @Import 指令，您可以定义可在整个应用程序中重复使用的单独组件集. 该指令还允许传递可迭代对象，甚至可以为更不可知的组件定义定义别名。例如:

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

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

这种模式可让您将 UI 分成更小的、可管理的部分，这些部分可以轻松地在不同页面上维护和重复使用.

结论

WebStencils 提供了一个灵活且强大的系统，用于在 Web 应用程序中建立模板和布局. 透过利用 WebStencils 引擎和处理器组件，使用 AddVar 将数据传递到模板，并利用 @LayoutPage、@RenderBody 和 @Import 等布局指令，您可以为 Web 应用程序建立结构良好、可维护且可重复使用的视图.

我们讨论过的模板模式 - 标准版面、页首/正文/页尾和可重复使用组件-提供不同的方法来组织您的观点. 选择最适合您的应用程序需求和团队工作流程的模式（或模式组合）.

在下一章中，我们将探索 WebStencils 的更进阶功能，包括使用窗体、处理用户输入以及实作动态内容更新.

07

将待办事项应用程序迁移到 WebStencils

简介

在本章中，我们将采用先前在常数中使用纯 HTML 建立的待办事项应用程序，并将其迁移到 WebStencils 模板。此迁移将示范 WebStencils 如何让我们的程序代码更具可维护性和可扩充性。我们还将添加一些额外的功能来展示我们新方法的灵活性.

查看带有 WebStencils 演示的 GitHub 存储库，查看一个包含功能性待办事项应用程序的项目，该应用程序在概念上遵循与本指南中显示的程序代码相同的想法.

将 HTML 常数转换为模板

让我们先将 HTML 常数转换为 WebStencils 模板。我们将为应用程序的不同部分建立单独的模板.

主布局模板

先，让我们建立一个主布局模板，作为我们应用程序的基础.

建立一个名为 layout.html 的档案:

```
<!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>
```

在前面的范例中，我们直接包含来自 CDN 的不同函式库。如果这些连结中的任何一个具有特定版本，请务必使用 @@ 转义 @ 符号，这样 WebStencils 就不会对其进行解析.

待办事项列表模板

现在，让我们为待办事项列表建立一个模板。建立一个名为 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>
```

更新 WebModule

现在，让我们更新 WebModule 以使用这些新模板。请记住，在下面的程序代码中，TTodoList 类别已对 Delete、Complete 或 GetAllItems 等方法进行了额外的扩展. 该程序代码未包含在本指南中，因为为了简单起见，它是纯 Delphi 程序代码。请查看 GitHub 演示项目可以看到类似的程序代码方法.

```
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.
```

备注

上面的程序代码过于简化。为了更好的视觉理解，它没有为每个端点使用 Actions. GitHub 上提供的演示遵循更易于维护的 MVC 方法和更好的逻辑抽象.

新增额外功能

现在我们已经将应用程序迁移为使用 WebStencils，让我们添加一些额外的功能来演示新结构的可扩展性和可维护性.

任务类别

让我们加入对任务进行分类的功能。首先，更新 TTodoList 单元中的 TTodoItem 类:

```
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;
```

现在，更新 todo-list.html 范本以包含类别:

更新 WebModule 中的 HandleAddTodo 方法:

```
@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;
```

任务过滤

让我们新增按类别过滤任务的功能。建立一个新的模板文件，名为 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>
```

更新 todo-list.html 模板以包含类别过滤器:

```
@LayoutPage layout.html

@Import category-filter.html

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

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

新增方法来处理 WebModule 中的过滤:

```
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;
```

更新 WebModuleDefault 方法以处理新的过滤器路由:

```
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;
```


警告

本指南的主要重点是 WebStencils 以及如何将其与 RAD Studio 一起使用。在前面的程序代码中，为了简化对代码段的理解，没有包含一些额外的类别过滤等所需逻辑，这些只是与 WebStencils 无关的 Delphi 程序代码.

结论

在本章中，我们成功地将待办事项应用程序从使用 HTML 常数迁移到 WebStencils 模板. 这次迁移使我们的程序代码更易于维护和阅读. 我们透过将 HTML 移到单独的模板档案中来分离我们的关注点，这些模板档案可以轻松修改，而无需触及 Delphi 程序代码.

我们还透过新增任务类别和筛选等新功能来展示新方法的可扩展性. 由于 WebStencils 模板的灵活性，这些附加功能的实作非常简单.

透过使用 WebStencils，我们获得了几个优势:

1. **关注点分离**: 我们的 HTML 现在与 Delphi 程序代码分开了.
2. **可重复使用性**: 我们可以轻松地在不同页面上重复使用类别过滤器等组件.
3. **可维护性**: 无需修改 Delphi 程序代码即可更改 UI.
4. **可扩展性**: 新增功能更简单，同时尽可能避免程序代码重复.

在下一章中，我们将探索更高级的 WebStencils 功能以及如何将它们集成到我们的应用程序架构中.

08

WebStencils 的高阶选项

简介

在本章中，我们将看到 WebStencils 中提供的更多进阶功能和选项。这些工具将允许您创建更动态、高效和复杂的 Web 应用程序.

@query 关键词

@query 关键词可让您直接在范本中读取 HTTP 查询参数。这对于建立基于 URL 参数的动态内容非常有用.

```
<!--
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

@Scaffolding 关键词提供了一种根据应用程序的数据结构动态产生 HTML 的强大方法。这对于建立窗体或显示数据表特别有用.

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

要实作 scaffolding，您需要处理 WebStencilsProcessor 的 OnScaffolding 事件:

```
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;
```

上面的程序代码将 Scaffolding 关键词替换为 AReplaceText 值.


窍门

在此@Scaffolding 程序代码范例中，为了更好地理解，HTML 标记被硬编码，但应使用储存在其他类别、记录、常数等中的程序代码来利用此关键词，以提高 HTML 片段的结构化和可重复使用性.

@LoginRequired

@LoginRequired 关键词可用于根据用户验证状态限制对范本的某些部分的存取。当遇到此关键词时，它会触发一个特定事件，您可以处理该事件来检查使用者是否登入.

为了定义使用者是否登录，WebStencilsProcessor 提供了一个布尔属性 UserLoggedIn，允许储存用户是否成功登入.

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

您还可以指定角色以进行更精细的访问控制:

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

若要指定目前使用者的角色，可以在 WebStencilsProcessor 中找到 UserRoles 属性。如果使用者有不同的角色，请定义这些角色，并以逗号分隔。范例: sales,management

处理器中的内部事件将评估使用者的角色，并引发 EWebStencilsLoginRequired 异常，以防使用者在未登入的情况下不具有特定角色.

> ⚠ 警告
>
> *请记住，WebStencils 不提供内建的身份验证系统，并且仅在用户要求透过外部服务正确身份验证时才使用这些属性和关键词.*

OnValue 事件处理函式

OnValue 事件处理函式提供了一种动态向模板提供数据的方法. 当您需要动态计算值或希望将数据存取逻辑与模板分开时，这特别有用.

```
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;
```

在您的模板中，您可以使用:

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

模板模式

WebStencils 支持几种常见的模板模式，可以帮助您有效地组织应用程序的视图.

标准布局

模式使用内建的@LayoutPage 和@RenderBody 方法。它非常适合在整个网站上保持一致的结构，同时允许各个页面提供其特定内容。到目前为止所展示的演示都遵循这种模式.

Header/Body/Footer

在此模式中，每个页面都是一个单独的模板，但它们都共享公共部分，例如页首和页尾。您可以像这样建立模板，而不是使用布局页面:

```
@Import Header.html

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

@Import Footer.html
```

这种方法比标准布局模式提供了更大的灵活性，但可能需要对公共元素进行更多管理.

可重复使用的组件

使用 @Import 指令，您可以定义可在整个应用程序中重复使用的单独组件集. 该指令还允许传递可迭代对象，甚至可以为更不可知的组件定义定义别名。例如:

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

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

此模式可让您将 UI 分解为更小的、可管理的部分，这些部分可以轻松维护并在不同页面上重复使用。您可以在程序代码中找到此模式的一些范例此范例.

结论

WebStencils 的这些进阶功能为创建动态、高效和复杂的 Web 应用程序提供了强大的工具. 透过利用这些功能，您可以创建更易于维护和灵活的程序代码，更有效地分离您的关注点，并建立可以扩展并适应不断变化的需求的强大 Web 应用程序.

在下一章中，我们将探讨如何将 WebStencils 与 RAD Server 集成，为建立可扩展的 Web 应用程序提供更多可能性.

09

使用 RAD 服务器与 WebStencils 集成

简介

RAD Server 也受益于新的 WebStencils 函式库. 已开发出特定的集成，以允许 RAD Server 充分利用 Web 开发的所有潜力。当涉及到语法、组件等时，我们迄今为止所学到的一切仍然适用于 RAD Server.

将 WebStencils 与 RAD 服务器集成

与我们之前看到的范例一样，使用 RAD Server 和 WebStencils 有多种方法。让我们来看看两种最常见的模式:

使用 WebStencils 处理器

这是一种简单的方法，使用处理器(在设计时或以程序设计方式建立)根据每个请求产生 HTML，并直接在 RAD 服务器响应中传回。要使用的模板需要从档案中读取，储存在常数、变量等中，然后进行处理.

```
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;
```

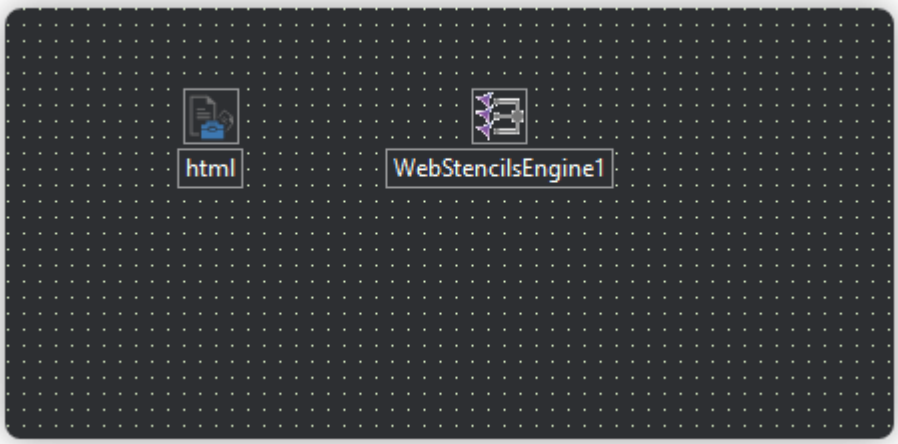
执行此 RAD Server 项目并开启浏览器，您应该能够存取 URL http://localhost:8080/testfile 并查看在变量 LTemplateFile 中定义的模板的渲染内容.


备注

对于 RAD 服务器，模板路径必须是绝对路径。由于使用 Apache 或 IIS 在生产中部署 RAD 服务器的方式，因此无法使用相对的服务器路径.

使用 WebStencils 引擎

对于此选项，我们建议将现有的 TEMSFileResource 组件与 TWebStencilsEngine 组件结合起来. 第一个执行到文件系统的映射，而第二个管理 HTTP 映射和模板处理. 这些组件使用 WebStencils Engine 的 Dispatcher 属性进行连接.



要正确配置 TEMSFileResource，我们需要使用 PathTemplate 属性指定模板的位置.

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

请特别注意 {filename} 通配符号，因为它将引用每个页面.

配置组件后，可以使用通配符号 {fileName} 或模板档案的名称在引擎中定义 PathsTemplates.

为了正确地映像文件，我们需要在 FileResource 上定义一些属性，就像我们在前面的范例中所做的那样:

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

在此范例中，我们现在可以存取端点 http://localhost:8080/testfile/{filename} 而{filename}是储存在我们在 PathTemplate 属性中定义的路径中的任何模板.

如果同一个 TWebStencilsEngine 需要多个 TEMSFileResource，那么全局方法 AddProcessor 可以指定额外的 TEMSFileResource，.
范例:
AddProcessor(FileResourceResource, WebStencilsEngine1);

重新建立 RAD 服务器的任务应用程序

RAD 服务器的工作方式与 WebBroker 略有不同。作为一个与 REST 兼容的应用程序，它没有内存状态，因此需要考虑一些事情.

如果您检查了 GitHub 储存库中包含的 WebBroker 范例，您会发现它遵循 MVC 方法。在同一个储存库中，您可以找到完全迁移到 RAD 服务器并提供相同功能的相同范例。让我们列举一下需要重构的内容:

数据库管理

在 WebBroker 范例中，由于任务储存在内存中，因此使用单例模式来避免多线程问题。在 RAD Server 中，任务无法储存在内存中(至少不容易)，因此我们将使用 InterBase 数据库代替.

若要从模型存取数据库而不是直接在模型中定义数据库，请在 TTasks 模型建构元中指派 TFDConnection 组件. 为了简化演示，在同一台 RAD 服务器 DataModule 中建立了 TFDConnection 组件。在生产应用程序中，建议使用 TFDManager 组件.

控制器参数

WebBroker 和 RAD 服务器请求和响应略有不同，因此需要对传递给控制器方法的参数进行一些重构.

从行动(Actions)到端点(EndPoint)

WebBroker 使用 Actions 来定义端点以及与其关联的业务逻辑。对于 RAD 服务器，必须使用属性的方法关联的特定 URI 来定义资源内的方法.

处理请求的数据

RAD 服务器使用 JSON。将数据从我们的待办事项应用程序传送到后端并处理该数据需要进行一些更改.

1. **HTMX 扩充的使用 JSON-enc**: 此扩充功能以 JSON 格式而不是窗体数据(默认行为)对传送到后端的请求进行编码. 要使用此扩展，必须新增一个简单的 <script> 卷标, 且属性 hx-ext="json-enc" 必须在我们定义 hx-post 或 hx-put 请求的相同标签上指定.
2. **数据处理的修改**: 由于 RAD 服务器将请求数据处理为 JSON，因此需要进行一些修改才能从请求中取得值。使用标准 JSON 函式库足以满足本示范的目的.

处理静态 JS、CSS 和影像

需要使用 TEMSFileResource 组件来传递静态档案。请建立独立的组件来对应这些档案(JS、CSS 和映像)所在的每个文件夹.

前端资源

RAD Server 依赖资源，这表示我们的主 URL 将在末尾新增资源的名称。例如 - https://localhost:8080/web.

WebBroker 网站建立的端点已迁移到 RAD 服务器，但这需要对 WebStencils 模板进行轻微修改以指向新建立的资源

10

资源和进一步学习

以下是一些有用的资源，可进一步扩展本书所描述的主题的潜力:

文件和连结

官方 HTMX 文件 (HTMX.org)

尽管本书讨论了最常见的关键词，但 HTMX 可以为您的项目添加更多内容。 HTMX 团队提供的文件内容丰富但平易近人.

除了官方文件外，htmx.org 网站还提供论文和范例.

- 官方文件
- 范例
- 论文

Delphi 和 HTMX(Embarcadero 部落格)

在本系列文章中，首先介绍了 HTMX 和 WebBroker 。尽管本书已经讨论了大多数概念，但也显示了更多程序代码范例和其他集成：例如 WordPress 与 HTMX 集成.

- 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 服务器技术指南

如果您不熟悉 RAD 服务器并想探索其所有可能性，您可以阅读一本现在就提供的书籍。它将引导您完成主要功能以及更具体和高级的功能.

在这里探索它

MVC 模式中的 HTMX (HTMX.org)

本页简要说明了 HTMX 如何融入 MVC 风格的 Web 应用程序. 它提供了精简控制器的范例以及如何使用 HTMX 建置用于 Web 开发的 MVC 流程. 虽然不是特定于 Delphi，但这些概念广泛适用于各种语言的 MVC 设计模式.

在这里探索它

WebStencils (DocWiki)

查看 DocWiki 中我们提供的官方 WebStencils 文檔.

在这里探索它

UI/CSS 函式库

这个范围很大并且有多个函式库. 鉴于 WebStencils 和 HTMX 与 UI 无关的性质，因此几乎可以使用任何函式库。这里有一些工作得很好并且有很好文档的相关函式库.

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

TailwindCSS 是一个非常巨大的函式库，透过 CDN 包含它会增加非常重的初始负载. 他们的 CLI tool 删除所有未使用的类别因此您应该在生产环境中使用.

进一步扩展 HTMX

尽管 HTMX 的声明性质使项目对 JS 的依赖性大大降低，但在某些情况下，纯客户端互动仍然需要 JS. 例如，考虑将网站从浅色模式变更为深色模式。我们可以在后端进行管理，并在深色模式下发回新的 HTML，但藉助现代 CSS 函式库，这可以在客户端轻松完成.

当我们在客户端需要额外的功能和互动时，有不同的微型 JS 函式库可以与 HTMX 和 WebStencils 很好地集成.

AlpineJS

AlpineJS 是一个轻量级 JavaScript 框架，旨在为 HTML 添加简单的互动. 它提供了一种操作 DOM 元素的声明性方式，而无需编写大量 JavaScript 程序代码. 人们经常将它与 Vue 或 React 等框架进行比较，但它更小且更容易集成到现有 HTML 中. AlpineJS 非常适合在网页上添加 JavaScript 行为，而无需大型框架的复杂性.

- 使用案例: 模态框、下拉式选单、切换、窗体验证和其他 UI 互动.
- 主要特点: 声明式语法、反应式数据、DOM 操作指令.

文件: AlpineJS 文件

Hyperscript

Hyperscript 是一种脚本语言，旨在简化 HTML 中的事件处理和逻辑. 与需要更详细语法的 JavaScript 不同，Hyperscript 旨在直接嵌入到 HTML 属性中. 它专注于透过使用类似自然语言的语法使事件驱动程序设计更加直观，允许开发人员无需复杂的 JavaScript 即可创建动态行为.

- 使用案例: 交互式按钮、窗体提交处理、元素可见性切换.
- 主要特点: 自然语言语法、声明性事件、常见互动的简化处理.

文件: Hyperscript 文件

现在就试用 RAD Studio!

了解如何仅使用一个程序代码库即可轻松地将原生应用程序部署到多个平台!

www.embarcadero.com