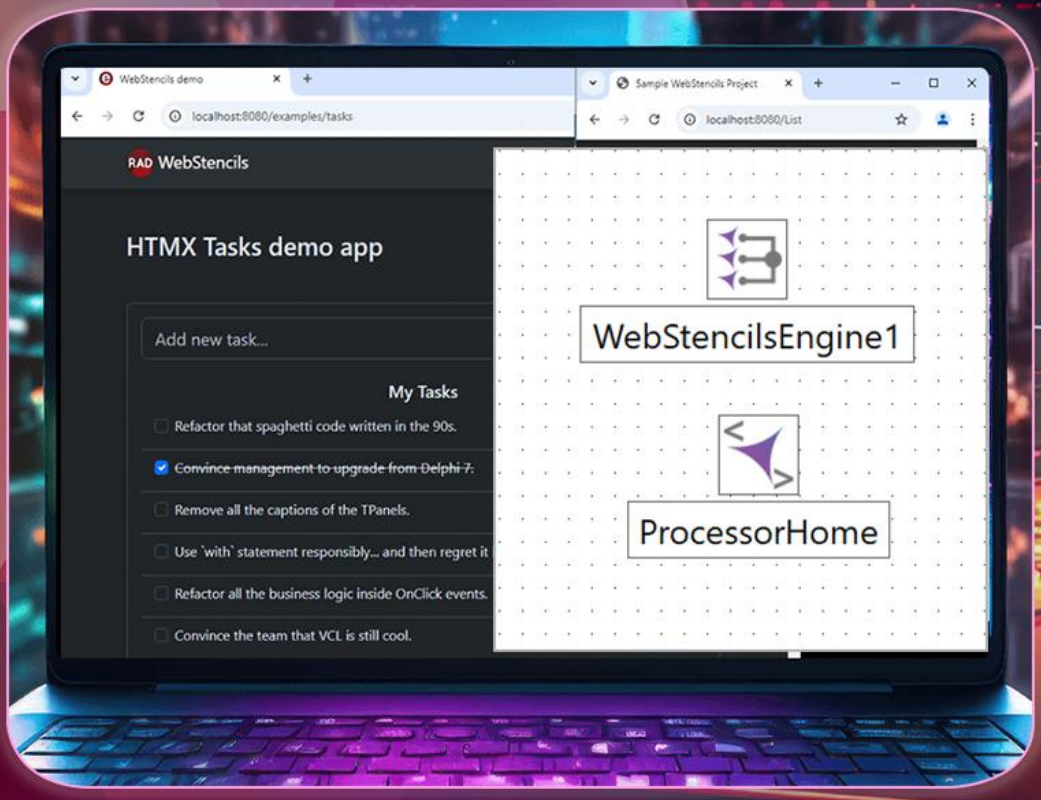
HTMX & WebStencils

使用 RAD Studio 快速进行网页开发 (更新到 13.0)



Author
Antonio Zapater

embarcadero

目录

目录

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整合，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

在线范例: *https://wsdemo.embarcadero.com*

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整合.
- **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. **整合性**: 它与 RAD Studio 整合，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 服务器整合.

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.
```

将会话管理整合到 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 字符串的整合方法.

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 文件整合和处理技术, 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 整合

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整合到 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:

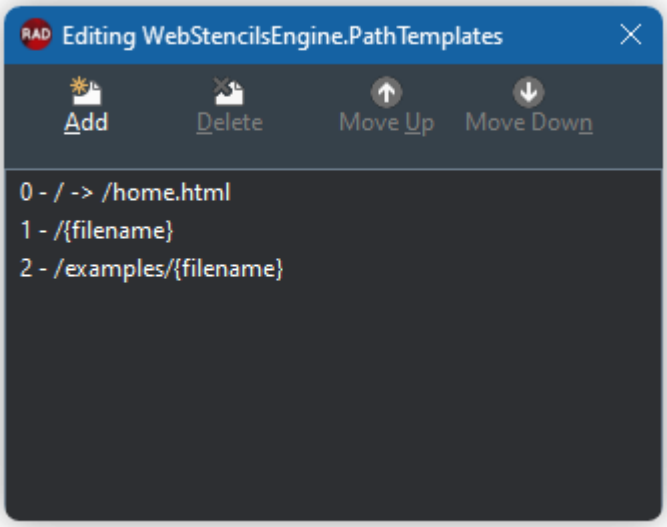
- `InputFilename`: 指定要处理的文件档案.
- `InputLines`: 允许直接分配指定要处理的内容.
- `Engine`: 指定引擎继承数据变量、事件处理程序等(可选).
- `Content`: 产生最终的处理内容.
- `AddVar`: 将对象新增至处理器可用的脚本变量列表中.

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>
```


窍门

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.

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. 虽然并非每个项目都需要它们，但了解它们的存在可以在遇到合适的场景时节省大量时间.

我们将透过表达式求值、使用鹰架(scaffolding)产生动态 HTML 以及使用 OnValue 事件进行自定义值解析来了解 @ 关键词的更高级用法.

使用 @() 进行表达式求值

我们已经了解了使用 @object.property 语法存取基本属性的方法，这种方法在大多数情况下都很好用，但有时你需要更复杂的计算、方法呼叫、数组索引或组合多个值。这时@()文法就显得至关重要了.

@() 语法使用 Delphi 的 LiveBindings 表达式求值器，它支持远超简单属性存取的复杂表达式.

@() 能做什么

方法呼叫：

可以使用标准的 RTL 函数，例如格式化日期、字符串操作或数学函数，以及在您自己的类别中定义的自定义公共函数.

```
<p>Format date: @(FormatDateTime('yyyy-mm-dd', order.Date))</p>
<p>UpperCase: @(UpperCase(customer.FirstName))</p>
<p>Round: @(Round(product.SalePrice))</p>

<!-- GetFullName() is a public method within customer object -->
<p>Public method class: @(customer.GetFullNume)</p>
```

算术和计算：

```
<p>Price with VAT: $@(product.Price * product.VatPercentage)</p>
<p>Discount: $@(product.basePrice - product.salePrice)</p>
<p>Previous page: @(pagination.PageNumber - 1)</p>
```

数组和列表索引：

```
<!-- TStringList -->
<p>First category: @(categories.Strings[0])</p>
<p>Selected color: @(colors.Strings[context.userChoice])</p>

<!-- TList<T> -->
<p>Product name: @(products.Items[2].Name)</p>

<!-- Classic static arrays (requires indexed property with getter) -->
<p>First item: @(data.ClassicArray[0])</p>
```

重要：

- 对于 `TStringList`, 使用`.Strings[index]` 语法
- 对于 `TList<T>`,使用`.Items[index]` 语法
- 对于经典静态数组经典静态数组 (`array[0..N] of Type`), 你需要在 Delphi 类别中建立一个带有 getter 函数的索引属性

备注

与简单的 @object.property 存取相比，@() 语法会略微增加效能负荷，因为它在运行时间使用了 Delphi 的表达式求值器。但是，对于大多数应用程序来说，这种负荷可以忽略不计.

@Scaffolding

@Scaffolding 关键词提供了一种根据应用程序数据结构动态产生 HTML 的方法。当需要重复产生类似的 HTML 模式，并且产生这些 HTML 涉及更复杂的业务逻辑（难以整合到模板中）时，这对于建立窗体或显示数据表尤其有用.

然而，对于独特的、手工设计的用户接口，使用 @Import 关键词的常规模板语法通常更清晰、更易于维护.

以下是基本语法:

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

若要实现鹰架功能，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;
```

上述程序代码将 Scaffolding 关键词替换为 AReplaceText 值.

实用鹰架范例

在上面的程序代码范例中，为了简单起见，HTML 卷标是硬编码的，但在实际应用中，您需要根据类别结构以程序设计方式产生这些 HTML。以下是一种更复杂的方法:

```
procedure TMyWebModule.ProcessorScaffolding(Sender: TObject;
  const AQualifClassName: string; var AReplaceText: string);
var
  LContext: TRttiContext;
  LType: TRttiType;
  LProp: TRttiProperty;
  LFieldHtml: string;
begin
  if SameText(AQualifClassName, 'User') then
  begin
    AReplaceText := '<div class="form-group">';

    LContext := TRttiContext.Create;
    try
      LType := LContext.FindType('TUser');
      if LType <> nil then
      begin
        for LProp in LType.GetProperties do
        begin
          // Skip internal properties
          if not (LProp.Visibility in [mvPublic, mvPublished]) then
            Continue;

          // Generate appropriate input based on property type
          case LProp.PropertyType.TypeKind of
            tkInteger:
              LFieldHtml := Format('<input type="number" name="%s" class="form-
control">', [LProp.Name]);
            tkString, tkUString, tkLString, tkWString:
              LFieldHtml := Format('<input type="text" name="%s" class="form-
control">', [LProp.Name]);
            tkEnumeration:
              if LProp.PropertyType.Handle = TypeInfo(Boolean) then
                LFieldHtml := Format('<input type="checkbox" name="%s" class="form-
check-input">', [LProp.Name])
              else
                Continue;
          else
            Continue;
          end;
```

```
          AReplaceText := AReplaceText + Format(
            '<div class="mb-3">' +
            '<label class="form-label">%s</label>' +
            '%s' +
            '</div>', [LProp.Name, LFieldHtml]);
        end;
      end;
    finally
      LContext.Free;
    end;

    AReplaceText := AReplaceText + '</div>';
  end;
end;
```

这种方法利用实时互动技术 (RTTI) 来检查您的类别并自动产生相应的表单域。当您的用户类别发生变更时，窗体会自动调整，无需手动更新.

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或者，当您想拦截特性存取以进行特殊处理时，这特别有用.

当 WebStencils 遇到无法透过常规 RTTI 寻找解析的属性参考时，就会触发此事件。这使您有机会自行提供值并建立自定义对象和属性层次结构.

基本 OnValue 范例

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

然后，您可以在模板中使用:

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

进阶 OnValue 场景

OnValue 事件比乍看强大得多。以下是一些实际应用场景:

1. Computed 特性

```
procedure TMyWebModule.ProcessorOnValue(Sender: TObject;
  const ObjectName, FieldName: string; var ReplaceText: string;
  var Handled: Boolean);
begin
  if SameText(ObjectName, 'stats') then
  begin
    Handled := True;
    if SameText(FieldName, 'TotalRevenue') then
      ReplaceText := FormatFloat('$#,##0.00', CalculateTotalRevenue)
    else if SameText(FieldName, 'ActiveUsers') then
      ReplaceText := IntToStr(GetActiveUserCount)
    else if SameText(FieldName, 'ConversionRate') then
      ReplaceText := FormatFloat('0.00%', CalculateConversionRate * 100)
    else
      Handled := False;
  end;
end;
```

模板使用:

```
<div class="dashboard">
  <div class="stat">Total Revenue: @stats.TotalRevenue</div>
  <div class="stat">Active Users: @stats.ActiveUsers</div>
  <div class="stat">Conversion Rate: @stats.ConversionRate</div>
</div>
```

2. 在地化/翻译

```
procedure TMyWebModule.ProcessorOnValue(Sender: TObject;
  const ObjectName, FieldName: string; var ReplaceText: string;
  var Handled: Boolean);
```

```
begin
  if SameText(ObjectName, 'i18n') then
  begin
    ReplaceText := GetTranslation(FieldName, CurrentUserLanguage);
    Handled := True;
  end;
end;
```

模板使用:

```
<h1>@i18n.WelcomeMessage</h1>
<p>@i18n.IntroductionText</p>
<button>@i18n.GetStartedButton</button>
```

3. 基于复杂逻辑的条件内容

```
procedure TMyWebModule.ProcessorOnValue(Sender: TObject;
  const ObjectName, FieldName: string; var ReplaceText: string;
  var Handled: Boolean);
begin
  if SameText(ObjectName, 'feature') then
  begin
    Handled := True;
    if SameText(FieldName, 'enabled') then
      ReplaceText := BoolToStr(IsFeatureEnabled(CurrentUser, FieldName), True)
    else if SameText(FieldName, 'available') then
      ReplaceText := BoolToStr(IsFeatureAvailable(CurrentSubscription, FieldName),
True)
    else
      Handled := False;
  end;
end;
```

模板使用:

```
@if (feature.enabled) {
  <div class="premium-feature">
    <h2>Premium Feature</h2>
    <p>This feature is available with your subscription.</p>
```

```
    </div>
}
```

OnValue 与 AddVar

您可能想知道何时使用 OnValue，何时使用 AddVar。当您有实际对象且其属性在模板处理之前就已存在时，请使用 AddVar。这样可以为您的网域对象启用正常的 RTTI 属性存取.

当您需要透过程序代码计算值、数据不是以对象形式存在，或想要拦截和自定义属性存取以实现虚拟属性或伪对象时，请使用 OnValue.

`OnValue` 本质上是一种回退机制。 `WebStencils` 首先尝试使用 `RTTI`，如果找不到该属性，则呼叫 `OnValue`.

身份验证和授权

关于 @LoginRequired 的说明: 在早期版本的 WebStencils 中，建议使用 @LoginRequired 关键词并手动进行用户验证检查。但是，自 RAD Studio 13.0 起，WebStencils 内建了全面的会话管理和身份验证系统，使得手动身份验证处理基本上过时.

我们将在第 9 章详细介绍这些特性。如果您正在建立新的应用程序，请使用内建的身份验证系统，而不是自行实现。内建系统更安全、更易于使用，并且能够处理自定义实作中容易忽略的特殊情况.

对于使用 @LoginRequired 批注的旧程序代码，我们建议您在条件允许的情况下迁移到新的身份验证系统。迁移过程非常简单，在程序代码量减少、安全性提升和粒度控制方面优势显着.

结论

WebStencils 的进阶功能为您提供了处理复杂场景的额外工具. `Scaffolding` 有助于使用程序代码重复生成 HTML，OnValue 允许您提供计算值并拦截属性访问，而 @() 语法允许将基本逻辑嵌入到模板本身中.

然而，最好的程序代码往往是最简单的程序代码。只有当这些特性确实能让你的应用程序更易于维护时才使用它们，但不必在所有地方都使用它们. 有时候，使用简单的模板，结合常规的 @if 语句和 @ForEach 循环，才是正确的答案.

09

会话管理和身份验证

简介

会话管理历来是使用 WebBroker 建立 Web 应用程序时最棘手的部分之一. 你需要手动追踪使用者在不同请求中的活动，将会话数据储存在某个地方，处理会话过期，实现身份验证逻辑，以及管理授权. 这虽然可行，但意味着在专注于实际应用程序之前，需要编写和维护大量的基础设施程序代码.

RAD Studio 13.0 引入了一个全面的会话管理和身份验证系统，可以自动处理所有这些操作。只需在 Web 模块上新增三个组件，处理一个事件，即可拥有一个完整的、可用于生产环境的身份验证系统，并具备基于角色的访问控制功能.

这并非一个简单的概念验证。内建系统包含了成熟身分验证框架应有的所有功能：自动会话生命周期管理、可设定的储存选项、基于角色的授权、未认证使用者的自动复位向以及加密签署的会话 ID.

本章我们将探讨如何使用这些新组件，以最少的程序代码为您的 WebBroker 应用程序新增身分验证功能.

3 个组件

会话管理系统由三个组件组成，它们协同工作以处理身份验证和授权:

TWebSessionManager

这是管理会话生命周期的基础组件。它负责会话的创建、储存、过期和清理。当收到请求时，会话管理器会撷取现有会话或建立一个新会话.

会话管理器具有高度可配置性。您可以选择会话 ID 的储存位置（cookie、标头或查询参数）、会话的作用域（按请求、按使用者或按使用者+IP），以及会话的过期时间.

TWebFormsAuthenticator

此组件处理基于 HTML 窗体的身份验证。它管理整个验证流程：将未认证用户重新导向到登入页面、处理登入窗体提交、透过您的程序代码验证凭证，以及在登入尝试成功或失败后重新导向用户.

身份验证器的工作原理是拦截对受保护页面的请求。如果用户未通过身份验证，系统会自动将其重新导向至登入 URL。使用者提交凭证后，身份验证器会触发一个事件，用于验证用户名称和密码。根据您的响应，用户要么登入并复位向到目标页面，要么被复位向到失败页面.

TWebAuthorizer

身份验证器负责处理“你的身份”，而授权器负责处理“你可以做什么”。它透过授权区域提供基于角色的访问控制。您可以定义网站的受保护区域，并指定存取这些区域所需的角色.

例如，您可以设定一个只有具有"admin"角色的使用者才能存取的/admin 区域，而所有具有"user"角色的使用者都可以存取/user 区域。授权器会自动强制执行这些规则.

设定身份验证

让我们逐步了解如何建构一个完整的身份验证系统。好消息是，它只需要很少的程序代码.

组件配置

首先，将这三个组件拖曳到您的 WebModule 中:

1. `TWebSessionManager`
2. `TWebFormsAuthenticator`
3. `TWebAuthorizer`

现在配置身份验证器的属性:

```
// In the Object Inspector or code:
```

```
// Where to send unauthenticated users
WebFormsAuthenticator.LoginURL := '/login';

// Where to redirect after successful login
WebFormsAuthenticator.HomeURL := '/';

// Where to redirect after failed login
WebFormsAuthenticator.FailedURL := '/login?error=1';
```

这就是基本设定。各个组件透过 WebModule 自动相互链接.

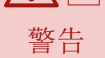
实作凭证验证

您唯一需要编写的程序代码是在验证器的 OnAuthenticate 事件中进行凭证验证:

```
procedure TWebModule1.WebFormsAuthenticatorAuthenticate(
  Sender: TCustomWebAuthenticator;
  Request: TWebRequest;
  const UserName, Password: string; var Roles: string; var Success: Boolean);
begin
  // Validate credentials against your user database
  Success := False;
  Roles := '';

  // Example: Check against database
  if ValidateUserCredentials(UserName, Password) then
  begin
    Success := True;
    Roles := GetUserRoles(UserName);  // e.g., 'user,admin'
  end;
end;
```

Roles 参数是一个以逗号分隔的角色名称字符串。例如，使用者可能拥有'user,moderator'角色，而管理员可能拥有'user,admin'角色.


警告

切勿以明文形式储存密码！请使用正确的密码哈希算法。以上范例假设 ValidateUserCredentials 函数内部会处理哈希值比较。为了演示，您可以使用硬编码的凭证，但生产应用程序应始终使用包含哈希密码的正确用户数据库进行验证.

建立登入窗体

建立一个简单的登入窗体模板(login.html):

```
@LayoutPage layouts/baseLayout

<div class="login-container">
  <h1>Sign In</h1>

  @if query.error {
    <div class="alert alert-danger">
      Invalid username or password. Please try again.
    </div>
  }

  <form method="post" action="/login">
    <div class="form-group">
      <label for="username">Username</label>
      <input type="text" id="username" name="username"
             class="form-control" required autofocus>
    </div>

    <div class="form-group">
      <label for="password">Password</label>
      <input type="password" id="password" name="password"
             class="form-control" required>
    </div>

    <button type="submit" class="btn btn-primary">Sign In</button>
  </form>
</div>
```

窗体会提交至 /login 页面，该页面由验证器自动处理。您无需为登入端点编写任何操作处理程序代码.

建立注销处理程序

注销操作由身份验证器自动处理。只需向配置的注销 `LogoutURL` 发送 POST 请求即可:

```
// In the Object Inspector or code:
WebFormsAuthenticator.LogoutURL := '/logout';
```

就是这样。当使用者向 /logout 发送 POST 请求时, 他们会自动注销并重新导向到主页 URL `HomeURL`. 也不需要任何操作处理程序代码.

您的注销窗体非常简单:

```
<form method="post" action="/logout">
  <button type="submit" class="btn btn-secondary">Sign Out</button>
</form>
```

身份验证器处理整个注销流程：终止会话、清除身份验证状态并将用户重新导向至主页.

@session 物件

身份验证设定完成后，每个模板都会自动取得 @session 对象的访问权限。该对象提供有关当前会话和用户的信息.

Session 特性

`@session` 对象公开了几个有用的属性:

认证状态:

- `@session.Authenticated` -布尔值，指示使用者是否已登入
- `@session.UserName` -经过验证的用户名
- `@session.UserRoles` -以逗号分隔的用户角色字符串

会话信息:

- `@session.SessionID` -唯一的会话标识符
- `@session.Timeout` -会话超时时间(分钟)
- `@session.CreatedTime` -会话创建时间
- `@session.AccessedTime` -最后访问时间戳
- `@session.AccessedCount` -本会话中的请求次数

请求信息:

- `@session.LastURL` -最后请求的 URL
- `@session.LastIP` -客户端 IP 地址
- `@session.LastReferer` -参考页面
- `@session.LastUserAgent` -浏览器用户代理字符串

在模板中使用 @session

以下是如何在模板中使用会话对象的典型方法:

```
<!-- Navbar showing different content based on authentication -->
<nav class="navbar">
  <a href="/" class="nav-brand">My App</a>

  <div class="nav-menu">
    <a href="/home" class="nav-link">Home</a>
    <a href="/about" class="nav-link">About</a>

    @if session.Authenticated {
      <a href="/dashboard" class="nav-link">Dashboard</a>

      @if session.UserHasRole('admin') {
        <a href="/admin" class="nav-link">Admin</a>
      }

      @if session.UserHasRole('moderator') {
        <a href="/moderate" class="nav-link">Moderate</a>
      }

      <a href="/logout" class="nav-link">Sign Out (@session.UserName)</a>
    } @else {
      <a href="/login" class="nav-link">Sign In</a>
      <a href="/register" class="nav-link">Register</a>
    }
  </div>
</nav>
```

UserHasRole() 方法特别有用。它会检查使用者是否具有特定角色，并在内部处理以逗号分隔的角色字符串.

授权区

授权区域可让您根据用户角色保护网站的整个部分。这比在每个模板中检查角色要方便得多.

配置保护区

您可以透过程序代码或组件的 Zones 属性配置授权区域。以下是程序代码配置方法:

```
procedure TWebModule1.WebModuleCreate(Sender: TObject);
var
  Zone: TWebAuthorizationZone;
begin
  // Protect all /admin/* URLs - require 'admin' role
  Zone := WebAuthorizer.Zones.Add;
  Zone.PathInfo := '/admin*';
  Zone.Kind := zkProtected;
  Zone.Roles := 'admin';

  // Allow public access to documentation - no authentication required
  Zone := WebAuthorizer.Zones.Add;
  Zone.PathInfo := '/docs*';
  Zone.Kind := zkFree;

  // Health check endpoint - bypass all authentication and session management
  Zone := WebAuthorizer.Zones.Add;
  Zone.PathInfo := '/health*';
  Zone.Kind := zkIgnore;

  // Protected user dashboard - require any authenticated user
  Zone := WebAuthorizer.Zones.Add;
  Zone.PathInfo := '/dashboard*';
  Zone.Kind := zkProtected;
  Zone.Roles := 'user,admin';  // Either role works
end;
```

星号 (*) 是通配符，符合路径之后的所有内容。因此，/admin* 可以匹配 /admin、/admin/users、/admin/settings 等.

区域类型

区域类型有三种:

- `zkProtected` -需要身份验证和指定角色(预设)
- `zkFree` -允许无需身份验证即可访问，但仍会进行会话管理
- `zkIgnore` -完全绕过身份验证和会话管理

对于监控系统频繁存取的健康检查端点或其他完全不需要会话管理的端点，请使用 zkIgnore。这样可以防止每次请求都建立会话，从而减少内存使用量和数据库负载 (如果您使用的是持久性会话储存).

当未经身份验证的使用者尝试存取 zkProtected 保护的区域时，系统会自动将其重新导向至登入页面。登入成功后，系统会将其重新导向回最初请求的页面.

如果使用者已登入但尝试存取与其角色不符的受保护区域，则会被重新导向至属性中配置的端点 `UnauthorizedURL`.

多重角色

您可以指定多个角色，角色之间以逗号分隔。使用者只需拥有所列角色之一即可获得访问权限:

```
Zone.Roles := 'admin,moderator,support';  // Any of these roles grants access
```

如果您需要使用者拥有所有角色(交集而非并集)，则需要在程序代码或范本中实现该检查，**因为内建系统使用的是 OR 逻辑**.

会话配置选项

`TWebSessionManager` 组件提供了多个设定选项，用于自定义会话行为.

会话 ID 存储

您可以使用 IdLocation 属性控制会话 ID 的储存位置:

```
// Store in cookies (default and recommended)
WebSessionManager.IdLocation := ilCookie;

// Store in HTTP headers (useful for APIs)
WebSessionManager.IdLocation := ilHeader;

// Store in query parameters (least secure, use only if necessary)
WebSessionManager.IdLocation := ilQuery;
```

Cookie 储存是预设设置，适用于大多数 Web 应用程序。对于不适合使用 Cookie 的 API 端点，可以使用 Header 储存。出于安全考虑（例如，会话 ID 会显示在 URL 中，并记录在服务器日志中等），应避免在生产环境中使用查询参数储存.

会话范围

会话范围决定了会话如何透过 Scope 属性与用户关联:

```
// New session for each request (no specific user binding)
WebSessionManager.Scope := ssUnlimited;

// Session tied to authenticated user (identified by username and roles)
WebSessionManager.Scope := ssUser;

// Session tied to user AND IP address (more secure but may break if IP changes)
WebSessionManager.Scope := ssUserAndIP;
```

默认值为 ssUnlimited，它会为任何没有会话 ID 的请求建立一个新会话。这适用于匿名用户和已认证用户.

使用 ssUser，会话 ID 是透过 HMAC-SHA2 算法结合 SharedSecret，根据用户名称和角色确定性地产生的。这意味着同一用户始终获得相同的会话 ID，从而防止会话固定攻击，并确保同一用户在不同装置或浏览器上的一致性.

`ssUserAndIP` 会将客户端的 IP 地址新增至 HMAC 运算中，透过将会话绑定到用户及其 IP 地址来提供额外的安全性。但是，对于使用行动网络或负载平衡器的用户来说，这可能会带来问题，因为这些情况下 IP 地址在会话期间可能会发生变化.

会话逾时

使用 Timeout 属性(以秒为单位)配置会话持续时间:

```
// Set timeout to 30 minutes (1800 seconds)
WebSessionManager.Timeout := 1800;

// Set timeout to 2 hours (7200 seconds)
WebSessionManager.Timeout := 7200;
```

预设超时时间为 3600 秒(1 小时)。逾时时间会在每次要求时自动重置，因此只要使用者仍在造访您的网站，就不会被强制注销。 AccessedTime 属性会在请求开始和结束时更新，确保每次使用者互动后逾时倒数都会重新开始.

共享密钥

对于使用者范围的会话（ssUser 或 ssUserAndIP），您必须配置共享密钥

```
WebSessionManager.SharedSecret := 'your-secret-key-here';
```

共享密钥在产生会话 ID 时用作 HMAC-SHA2 哈希的密钥.

> ⚠ 警告
>
> *在生产环境中，使用一个足够长的随机字符串作为共享密钥（至少 32 个字符）。将其安全地储存在配置文件或环境变量中，并且永远不要将其提交到版本控制系统中.*

结论

RAD Studio 内建的会话管理和验证系统省去了传统 Web 验证所需的大部分样板程序代码。该系统仅需三个组件和一个事件处理程序，即可建立一个生产就绪的系统，支持加密签章会话、基于窗体的身份验证和基于角色的授权.

该系统足够灵活，可以处理大多数常见场景：不同的会话范围、多个储存位置和细粒度的授权区域。@session 对象可让您的模板轻松存取身份验证状态，从而可以轻松建立适应用户权力的 UI.

10

数据库驱动的用户接口生成

简介

窗体是 Web 开发中最繁琐的部分之一。每次数据库架构变更时，都需要更新窗体，在 IDE 中重新产生数据集字段，然后手动更新所有引用这些字段的 HTML 窗体。要新增列吗?那将需要更新五个不同的窗体。将字段从可选改为必填?那需要在模板中寻找所有相关实例.

RAD Studio 13.0 引进了数据库驱动的 UI 产生功能，彻底改变了这个模式。您无需手动建立窗体来匹配数据库，而是建立能够自动适应数据库的窗体。您的 FireDAC 查询成为唯一的数据源，而您的范本则直接读取字段元数据.

警告

在模板中存取字段元数据之前，必须先配置白名单。出于安全考虑，TField 属性默认不会公开。我们稍后会详细介绍，但请记住，数据库驱动的窗体需要在 WebModule 初始化中进行明确配置.

它如何工作

ebStencils 现在可以透过模板表达式存取 FireDAC 属性。例如，当您引用数据集的字段集合时，您可以存取每个字段定义，包括您在 FireDAC 查询中配置的所有元数据.

这里有一个简单的例子:

```
@forEach(var f in customers.fields) {
  <div>
    <label>@f.DisplayLabel</label>
    <input type="text" name="@f.FieldName" value="@f.Value">
  </div>
}
```

此循环遍历 customers 数据集中的每个字段，并对每个字段读取以下内容:

- `DisplayLabel` -您在 FireDAC 中设定的易于理解的标签
- `FieldName` -实际的数据库域名
- `Value` -该字段的当前值

安全：白名单系统

数据库驱动的用户接口功能强大，但也带来了安全隐患。您肯定不希望模板意外暴露连接字符串或内部数据集状态等敏感属性.

WebStencils 包含一个白名单系统，用于控制哪些属性可以透过模板存取。默认情况下，只有最少的属性会被列入白名单:

TDataSet (预设已加入白名单):

- Active, FieldByName, First, Last, Next, Prior
- Bof, Eof, FieldCount, Fields, Found, RecordCount, RecNo

TStrings (预设已加入白名单):

- Contains, ContainsName, IndexOf, IndexOfName
- CommaText, Count, IsEmpty, DelimitedText
- Names, KeyNames, Values, ValueFromIndex, Strings, Text

TField (没有预设已加入白名单):

这是有意为之。 TField 属性必须根据应用程序的需求明确地列入白名单。如果没有将 TField 列入白名单，您将无法在范本中存取 DisplayLabel、FieldName 或 Value 等属性，数据库驱动的窗体也将无法正常运作.

您必须在 WebModule 初始化中设定 TField 白名单:

```
procedure TWebModule1.WebModuleCreate(Sender: TObject);
begin
  // Configure TField whitelist - REQUIRED for database-driven forms
  // Without this, templates cannot access field properties
  TWebStencilsProcessor.Whitelist.Configure(
    TField,
    ['DisplayText', 'Value', 'DisplayLabel', 'FieldName',
     'Required', 'LookupDataSet', 'LookupKeyFields',
     'Visible', 'DataType', 'Size', 'IsNull'],
    nil,   // No properties blocked
    False  // Don't inherit from parent class restrictions
  );
end;
```

如果没有此配置，任何尝试存取范本中的 `@field.DisplayLabel` 或类似属性的操作都将被静默忽略。这是出于安全考虑的设计概念，因此您需要明确选择要公开的内容.


警告

谨慎选择白名单上的项目。如果您不确定，请不要将其新增至白名单，以后需要时可以随时新增.

建立动态窗体

最常见的用例是建立能够自动适应数据集结构的窗体。以下是基本模式:

```
<form method="post">
  @forEach(var field in customers.fields) {
    @if(field.Visible) {
      <div class="form-group">
        <label for="@field.FieldName">
          @field.DisplayLabel
          @if(field.Required) {
            <span class="text-danger">*</span>
```

```
          }
        </label>

        <input
          type="text"
          name="@field.FieldName"
          value="@field.Value"
          @if(field.Required) { required }
          @if(field.Size > 0) { maxlength="@field.Size" }>
      </div>
    }
  }

  <button type="submit">Save</button>
</form>
```

此模板会产生一个完整的窗体:

- 来自 DisplayLabel 的标签
- 必填字段指标
- HTML5 必需属性
- 最大长度验证
- 显示当前值

但要注意的是：每个字段都会有一个文字输入框，无论它是数字、日期、布尔值或备注栏位。我们来解决这个问题.

@switch 运算符和字段类型

@switch 运算符(RAD Studio 13.0 中加入)非常适合处理不同的字段类型。您可以检查 field.DataType 并产生对应的 HTML 输入:

```
@forEach(var field in customers.fields) {
  @if(field.Visible) {
    <div class="form-group">
      <label>@field.DisplayLabel</label>

      @switch(field.DataType) {
        @case "ftString" {
          <input type="text"
                 name="@field.FieldName"
```

```
                value="@field.Value"
                @if(field.Size > 0) { maxlength="@field.Size" }>
        }
        @case "ftInteger" {
          <input type="number"
                name="@field.FieldName"
                value="@field.Value"
                step="1">
        }
        @case "ftFloat" {
          <input type="number"
                name="@field.FieldName"
                value="@field.Value"
                step="0.01">
        }
        @case "ftDate" {
          <input type="date"
                name="@field.FieldName"
                value="@field.DisplayText">
        }
        @case "ftDateTime" {
          <input type="datetime-local"
                name="@field.FieldName"
                value="@(FormatDateTime('yyyy-mm-dd''T''hh:nn', field.Value))">
        }
        @case "ftBoolean" {
          <input type="hidden" name="@field.FieldName" value="False">
          <input type="checkbox"
                name="@field.FieldName"
                value="True"
                @if(field.Value) { checked }>
        }
        @case "ftMemo" {
          <textarea name="@field.FieldName" rows="3">@field.Value</textarea>
        }
        @default {
          <input type="text" name="@field.FieldName" value="@field.Value">
        }
      }
    </div>
  }
}
```

这会根据数据库字段类型产生对应的 HTML5 输入框。数字使用 <input type="number">，日期使用 <input type="date">，依此类推.


窍门

复选框前的隐藏输入框是标准的 HTML 模式。如果窗体提交时未提交未选取的复选框（浏览器不会提交未选取的复选框），则隐藏域会确保传回「False」而不是空值。这简化了服务器端的处理.

可重复使用字段组件

动态窗体虽然好用，但当需要一致的样式、验证显示和特殊处理时，它们可能会变得冗长。解决方案是将字段渲染提取到可重复使用的局部零件/组件中.

建立 dynamicInput.html 模板:

```
@if(field.Visible) {
<div class="form-group">
  <label for="@field.FieldName">
    @field.DisplayLabel
    @if(field.Required) { <span class="text-danger">*</span> }
  </label>

  @switch(field.DataType) {
    @case "ftInteger" {
      <input type="number"
             class="form-control"
             name="@field.FieldName"
             value="@field.Value"
             @if(field.Required) { required }>
    }
    @case "ftDate" {
      <input type="date"
             class="form-control"
             name="@field.FieldName"
             value="@field.DisplayText"
             @if(field.Required) { required }>
    }
    @case "ftBoolean" {
      <div class="form-check">
        <input type="hidden" name="@field.FieldName" value="False">
        <input type="checkbox"
               class="form-check-input"
               name="@field.FieldName"
               value="True"
               @if(field.Value) { checked }>
        <label class="form-check-label">@field.DisplayLabel</label>
```

```
      </div>
    }
    @default {
      <input type="text"
             class="form-control"
             name="@field.FieldName"
             value="@field.Value"
             @if(field.Required) { required }
             @if(field.Size > 0) { maxlength="@field.Size" }>
    }
  }
</div>
} @else {
  <input type="hidden" name="@field.FieldName" value="@field.Value">
}
```

然后在你的主窗体中使用它:

```
<form method="post">
  @forEach(var f in customers.fields) {
    @import partials/dynamicInput { @field = @f }
  }
  <button type="submit">Save</button>
</form>
```

这种模式可以确保所有窗体的一致性，同时将字段特定的逻辑集中在一个地方。需要更改日期的显示方式？只需更新一个文件即可.

混合方法(通常是最佳选择)

虽然看起来所有窗体都可以自动生成，但在实际应用中，窗体有时比这更复杂，需要特定的格式、组织方式等等.

但这并不意味着我们无法从中受益。一个技巧是使用字段的 Tag 属性定义哪些字段应该自动处理，哪些应该手动处理。例如，将手动处理的字段的 Tag 值定义为 1，或者如果您希望将它们呈现在不同的版块中，甚至可以使用不同的 Tag 将它们分组.

我们来看一个例子:

```
<form method="post">
  <!-- Custom header section -->
  <div class="form-header">
    <h2>Customer Information</h2>
    <p>Please provide accurate contact details.</p>
  </div>

  <!-- Dynamic generation for fields with tags -->
  <div class="group-1">
    @forEach(var f in customers.fields) {
      @if(f.Visible and (f.Tag = 1)) {
        @import partials/dynamicInput { @field = @f }
      }
    }
  </div>
  <div class="group-2">
    @forEach(var f in customers.fields) {
      @if(f.Visible and (f.Tag = 2)) {
        @import partials/dynamicInput { @field = @f }
      }
    }
  </div>

  <!-- Custom section for special field handling -->
</form>
```

这结合了动态窗体的可维护性和自定义布局的灵活性。您可以兼得两者之长：FireDAC 控制字段定义，而您则掌控使用者体验.

关键在于，**无论是否自动产生窗体，字段元数据都很有用**. 即使在完全自定义或只读布局中，从 FireDAC 读取 DisplayLabel、Required、DataType、Value 等字段也意味着您拥有权威的字段定义来源，而无需在模板中重复编写这些信息.

结论

数据库驱动的 UI 产生简化了数据库和 Web 应用程序之间的关系。无需手动维护模式和模板之间的同步，您只需在 FireDAC 中定义一次元数据，WebStencils 即可在运行时间读取它.

关键在于，字段元数据存取的价值体现在两个方面:

1. **全动态生成**-让模板循环遍历字段并自动产生内容。非常适合管理接口、快速原型设计以及需要处理大量类似 CRUD 窗体的场景.
2. **选择性元数据访问**- 在自定义布局中使用字段属性，可以集中管理卷标、验证规则和类型信息。即使采用完全自定义的设计，@customers.fields.EMAIL.DisplayLabel 也能确保只有一个数据源.

白名单系统确保安全性，而 @switch 运算符则让您可以精确控制不同字段类型的渲染方式。结合可重复使用的局部视图，您既能获得灵活性，又能兼顾可维护性.

11

使用现代 CSS 框架

简介

WebStencils 最实用的特性之一正是它不做的事情：它不会对你的 CSS 或 JavaScript 程序代码强加任何预设。你无需学习任何自定义组件库，没有专有的样式系统，也没有所谓的”WebStencils 式”接口建构方式。它只是一个带有动态内容的 HTML 模板.

这种 CSS 和 JavaScript 的独立性意味着你可以使用任何框架，或完全不使用任何框架。想要 Bootstrap？没问题。喜欢 Tailwind？完全可以。 Bulma？当然可以。 BeerCSS？尽管用。自己写 CSS？当然支持.

我们一直在使用的示范项目都采用了 Bootstrap 5，但这只是众多选择之一。本章将探讨 WebStencils 如何与不同的 CSS 框架协同工作，特别着重于 Tailwind CSS，由于其建置流程的要求，Tailwind CSS 带来了一些有趣的挑战.

CSS 独立优势

许多服务器端框架通常会捆绑自己的 UI 组件和样式系统。这乍看之下似乎很有帮助（预先建置的组件可以加快初始开发），但随着时间的推移，它会变得非常局限。你会受到框架提供的功能的限制，而自定义往往意味着要与框架的预设设定作斗争.

WebStencils 采用不同的方法。它纯粹是一个模板渲染引擎。它处理你的 HTML，插入动态数据，处理条件循环等等，并输出结果。至于 HTML 的具体内容，则完全由你决定.

这具有若干实际好处:

- **你选择你的审美观.** 企业应用需要一种设计，消费者应用需要另一种设计，而内部工具则需要完全不同的设计.
- **你要跟上生态系的发展步伐.** 当新的 CSS 框架发布或现有框架更新时，您无需等待 WebStencils 新增支持。只需更新您的 CDN 连结或本机档案即可.
- **您可以依需求进行混搭.** 或许你可以使用 Bootstrap 进行快速原型设计，然后逐步用自定义 CSS 取代其中的各个部分。或使用 Tailwind 来建立公用类别，同时保留一些 Bootstrap 组件.
- **避免锁定效应.** 如果你以后决定更换框架，只需要修改 CSS，无需完全重写程序代码。你的服务器端逻辑和模板结构将保持不变.

框架选项(几个范例)

CSS 框架种类繁多，但以下是一些与 WebStencils 兼容性良好的常用选项:

Bootstrap

Bootstrap 是最广泛使用的 CSS 框架之一。它提供全面的组件、响应式网格系统和丰富的文件。它特别适用于企业级应用和快速原型开发，在这些应用中，一致且专业的外观比独特的设计更为重要.

Bootstrap 是基于组件的：您可以使用预先定义的类别来定义按钮、窗体、卡片、模态方块和其他 UI 元素。它还包含用于交互式组件（例如下拉式选单和工具提示）的 JavaScript.

Bulma

Bulma 是一个纯 CSS 的现代 CSS 框架，不依赖任何 JavaScript。它使用 Flexbox 进行布局，具有简洁现代的美学风格。与 Bootstrap 相比，Bulma 更容易定制，学习曲线也更平缓.

Bulma 采用模块化设计，因此您只需包含所需功能即可。它拥有完善的文档和不断增长的扩展生态系统.

查看更多信息

PicoCSS

PicoCSS 是一个专注于语意化 HTML 的极简 CSS 框架。它无需编写数十个实用类，即可为原生 HTML 元素赋予美观的样式。只需编写 `<button>` 即可，无需添加 “class="btn btn-primary"。

这种方法对于内容型网站来说非常实用，并且可以大幅减少 HTML 程序代码的冗余.

查看更多信息

BeerCSS

BeerCSS 是一个以简洁为核心的素材设计框架。它以轻量级的方式提供 Google 的 素材设计组件和样式。虽然它优先考虑行动端，但在桌面端也表现出色.

查看更多信息

DaisyUI

DaisyUI 是一个基于 Tailwind CSS 建立的组件库。它提供预先样式化的组件，同时保留了 Tailwind 的实用类别。它兼具两者的优点：组件的便利性和实用类的灵活性.

查看更多信息

使用这些框架

对于大多数框架来说，整合非常简单。只需在布局模板中添加 CDN 连结即可:

```
<!-- baseLayout.html -->
<!DOCTYPE html>
<html>
<head>
  <title>@page.title</title>

  <!-- Bootstrap -->
  <link rel="stylesheet"
        href="https://cdn.jsdelivr.net/npm/bootstrap@@5.3.8/dist/css/bootstrap.min.css">

  <!-- Or Bulma -->
  <link rel="stylesheet"
```

```
        href="https://cdn.jsdelivr.net/npm/bulma@@1.0.4/css/bulma.min.css">

  <!-- Or PicoCSS -->
  <link rel="stylesheet"
        href="https://cdn.jsdelivr.net/npm/@@picocss/pico@@2/css/pico.min.css">

  @RenderHeader
</head>
<body>
  @RenderBody
</body>
</html>
```

备注

请注意 @@ 语法。 WebStencils 要求在 CDN URL 中出现 @ 符号时将其加倍，以区别于模板语法.

就是这样。现在你的模板可以使用框架提供的所有类别和组件了。更改 CDN 链接，更改框架。WebStencils 根本不会知道或在乎你使用的 CSS.

Tailwind 特例

对于小型项目或快速实验，Tailwind CSS 的工作方式与其他框架类似，只需添加 CDN 连结即可。但由于其独特的运作机制，Tailwind 为生产应用带来了有趣的挑战.

了解 Tailwind 的使用方法

Tailwind 是一个实用性优先的 CSS 框架。它不使用预先建构的组件，而是使用小型、单一用途的实用类别来组合接口:

```
<!-- Traditional CSS approach -->
<button class="btn btn-primary">Click me</button>

<!-- Tailwind utility approach -->
<button class="bg-blue-500 text-white px-4 py-2 rounded hover:bg-blue-600">
  Click me
</button>
```

这种方法赋予你极大的灵活性，因为你无需编写自定义 CSS 即可建立任何设计。但要注意的是：Tailwind 会产生数千个实用类别。如果全部包含，你的 CSS 档案将达到 3-4MB。对于生产环境网站来说，这是一个巨大的初始负载.

解决方案在于建置过程。 Tailwind 会扫描你的 HTML 文件，辨识你实际使用的类，并产生一个只包含这些类的 CSS 文件。这个过程(称为「摇树优化」或「清理」)会将 CSS 档案的大小减少到 5-50KB 左右.

传统的 Tailwind 工作流程需要:

- 安装 Node.js
- npm (Node Package Manager)
- 类似 Vite 或 Webpack 的建置工具
- 包含依赖项的 package.json 文件
- 执行 npm install 和 npm run build 指令

对于只想为 Web 应用程序添加样式的 RAD Studio 开发人员来说，这会造成很大的额外成本。你实际上需要建立一个 JavaScript 开发环境，只是为了产生 CSS.

RAD 方法：独立 CLI

Tailwind 提供了一个独立的 CLI 二进制文件，无需安装 Node.js。它是一个单独的可执行文件，可以实现 Node.js 版本的所有功能，而无需 npm 生态系统.

这大大简化了在 RAD Studio 中使用 Tailwind 的过程。您无需再使用 Node.js、npm、package.json 和建置工具，只需…:

1. 下载 独立 CLI 二进制文件 (单一档案)
2. 在 Delphi 中配置生成后事件
3. 让建置系统自动处理所有事情

更理想的做法是，您可以采用混合方法，在开发过程中消除建置流程，同时保持生产环境的优化.

混合开发工作流程

巧妙的解决方案是根据建立配置切换 CSS 来源:

除错版本:

- 使用 Tailwind 的 CDN
- 所有实用类别均可立即使用
- 无需建置步骤

- 实时范本更改

发布版本:

- 透过建构后事件产生优化的 CSS
- 仅包含您正在使用的类别
- 档案体积小(通常为 5-50KB)
- 由 RAD Studio 自动处理

您的应用程序会侦测其运行模式并加载相应的 CSS。此模板使用 WebStencils 的条件语法:

```
<!-- baseLayout.html -->
<head>
  <title>@page.title</title>

  @if (env.debug) {
    <!-- Development: Full Tailwind via CDN -->
    <script src="https://cdn.jsdelivr.net/npm/@@tailwindcss/browser@@4"></script>
  } @else {
    <!-- Production: Optimized generated CSS -->
    <link rel="stylesheet" href="/static/css/output.css">
  }

  @RenderHeader
</head>
```

env.debug 变量是根据建置配置在您的 Delphi 程序代码中设定的:

```
// In WebModule initialization
procedure TWebModule1.SetupEnvironmentVariables;
begin
  // env.debug is automatically available in templates
  WebStencilsEngine.AddVar('env', nil,
    function(AVar: TWebStencilsDataVar; const APropName: string;
      var AValue: string): Boolean
    begin
      if SameText(APropName, 'debug') then
      begin
        AValue := {$IFDEF DEBUG}'True'{$ELSE}'False'{$ENDIF};
        Result := True;
      end;
    end);
end;
```

备注

上面的程序代码展示了定义环境变量值的一种方法。这里使用了匿名方法，但也可以使用 OnValue 事件或建立自定义对象来处理环境变量.

配置建置过程

实际的 CSS 产生是透过 Delphi 的建构后事件自动完成的。在项目选项中，设定 Release 版本以执行 Tailwind CLI:

建置后事件命令:

```
tools\tailwindcss.exe -i src\input.css -o templates\static\css\output.css --minify
--content "templates\**\*.html"
```

这个命令:

- 将 src/input.css 作为输入(该档案仅汇入 Tailwind)
- 扫描 templates/ 目录中的所有 HTML 文件
- 产生优化、精简的 CSS 文件
- 输出到 `templates/static/css/output.css`

仅对发布版本进行此配置。侦错版本将完全跳过此步骤，直接使用 CDN.

支持文件

两个小的配置文件就能实现这个功能:

src/input.css (输入 Tailwind):

```
@import "tailwindcss";
```

tools/tailwind.config.js (告诉 Tailwind 在哪里找到类别):

```
module.exports = {
```

```
    content: ["templates/**/*.html"],
    safelist: [
      // Add any dynamic classes generated in Delphi code
      'bg-red-500',
      'text-green-600'
    ]
}
```

safelist 用于存放您可能在 Delphi 程序代码中动态产生的类别（例如基于数据的条件颜色）。 Tailwind 无法透过扫描 HTML 侦测到这些类，因此您需要明确列出它们.

这个 GitHub 储存库包含一个完整的运行范例，所有档案均已配置.

关于框架选择的说明

CSS 框架领域竞争激烈，且存在一定的部落主义倾向。开发者对于实用性优先还是组件化方法、Bootstrap 的普及性和 Tailwind 的灵活性、框架究竟是有益还是有害等问题，都持有强烈的观点.

对于 WebStencils 使用者来说，这些争论大多无关紧要。框架的选择与服务器端架构是两个独立的决定。选择最适合你的项目、团队技能和设计需求的框架即可。 WebStencils 可以完美地满足所有这些需求.

你选择的框架会影响你的 HTML 程序代码，但不会影响你的 Delphi 程序代码或 WebStencils 架构。这才是正确的做法.

结论

ebStencils 的 CSS 无关性是一项特性，而非限制。它完全不参与样式设置，让您可以自由选择适合项目的工具，并在以后更改这些选择而无需重写应用程序.

Tailwind 整合证明，即使是具有复杂建置要求的框架也能在 WebStencils 生态系统中完美运作。这种混合 CDN/CLI 方法消除了开发摩擦，同时优化了生产输出，所有这些都透过标准的 Delphi 建构工具实现.

12

部署选项和 Docker

简介

您已经建立了 WebStencils 应用程序，并且在开发过程中运作良好。现在，实际问题来了：如何真正部署它?

答案或许会让你感到惊讶：将应用程序作为独立可执行文件运行，实际上就具备了生产就绪状态。你无需在上线前部署复杂的基础架构。独立模式能够有效应对实际流量，而且大多数应用程序都不会超出其承载能力.

也就是说，WebBroker 的架构赋予了您真正的灵活性。您可以从最简单的方案入手，随着需求的成长逐步升级到更复杂的部署。与那些将您锁定在特定基础设施的框架不同，WebBroker 应用程序可以以多种不同的方式运行，每种方式都有其自身的优缺点.

本章将探讨您的部署选项，重点介绍三种效果显着的方法：独立应用程序、基于 NGINX 的 FastCGI 以及 Docker 容器。我们还将讨论 IIS 和 Apache 等传统 Web 解决方案，以及为何需要仔细考虑它们.

我们的目标并非让您成为部署专家，而是帮助您选择最适合您情况的方法，并了解每种选项在实践中的实际意义.

了解您的选择

WebBroker 应用程序可以部署在多种配置中:

独立可执行文件(控制面板或基于窗体)
您的应用程序以独立的 Web 服务器运作。您需要建立一个 .exe 档案(Windows)或可执行文件(Linux)，该档案监听特定端口并直接处理 HTTP 请求。这是最简单的部署模型.

使用 NGINX 的 FastCGI(RAD Studio 13.0 起新增)
您的应用程序使用 FastCGI 协议作为独立进程运行。 NGINX 处理 HTTP 并将请求代理到您的应用程序。对于需要处理高流量的应用程序，这是建议的生产环境部署方案。它兼具简洁性和强大的请求处理能力.

Apache 模块或 IIS/ISAPI
您的应用程序编译为共享链接库（Apache 为 .so 文件，IIS 为 .dll 档案），并载入到 Web 服务器进程中.

Docker Container
您的独立应用程序将打包成容器镜像，其中包含所有依赖项。这确保了跨环境的一致性，并简化了部署基础架构.

这里的关键差异在于**有状态与无状态**:

- **有状态**(Standalone, FastCGI, Docker): 应用程序在请求间隙保持运作。内存持久化。会话自然运行。数据库联机可以进行联机池化.
- **无状态**(Apache module, IIS/ISAPI): 您的程序代码运行在 Web 服务器的进程空间。 Web 服务器可能会卸除并重载您的模块，导致所有内存状态遗失。会话需要数据库或文件存储，而处理数据库连接池则更为复杂.

对于大多数 WebStencils 应用程序而言，有状态部署是务实的选择.

独立部署

独立模式是 WebBroker 的默认模式。您的应用程序是一个包含自身 HTTP 服务器的普通可执行文件，无需外部 Web 服务器.

它是如何运作的

在 RAD Studio 中建立新的 WebBroker 应用程序时，您可以选择以下两种方式之一:

- **控制面板应用程序**: 以命令行程序形式运行。非常适合 Linux 服务器和 Windows 服务.

- **基于窗体的应用程序**: 包含一个用于停止/启动服务器的简易图形用户接口。仅适用于 Windows 系统.

两者都使用相同的底层 HTTP 服务器(Indy 的 TIdHTTPWebBrokerBridge)，只是接口不同.

这是独立服务器的核心部分:

```
procedure RunServer(APort: Integer);
var
  LServer: TIdHTTPWebBrokerBridge;
begin
  LServer := TIdHTTPWebBrokerBridge.Create(nil);
  try
    LServer.DefaultPort := APort;

    // Increase connection limits for better concurrency
    LServer.MaxConnections := 0;  // 0 = unlimited (default is limited)
    LServer.ListenQueue := 500;   // Backlog queue size (default is 15)
    LServer.KeepAlive := False;   // Disable HTTP Keep-Alive for simplicity

    LServer.Active := True;
    WriteLn('Server started on port ', APort);
    WriteLn('Press Enter to stop...');
    ReadLn;

    LServer.Active := False;
  finally
    LServer.Free;
  end;
end;
```

由于本指南并非对 WebBroker 的深入探讨，因此我们不会讨论其所有属性和高级优化方法，但一些最常用的调优参数包括 MaxConnections、ListenQueue 和 KeepAlive. 这些设定控制服务器如何处理并发请求等。需要注意的是，这些设定仅适用于您的应用程控 HTTP 服务器的情况，而不适用于使用 Apache 或 IIS 的情况.

生产可行性

现在你可能在想，"但是一个简单的独立可执行文件真的能处理生产环境的流量吗?"

没错。独立版模型比大多数开发者想象的更适合生产环境。在普通硬件上进行的测试表明，独立版 WebBroker 应用程序在标准工作负载下每秒可以处理超过一千个请求。这并非儿戏，而是名副其实的生产部署选项.

关键要求：必须在前端部署 NGINX 来处理 SSL/TLS。这是强制性的，并非可选项。 NGINX 负责终止 SSL 联机、有效率地处理静态文件，并提供生产级的 HTTP 层。您的 Delphi 应用程序应专注于业务逻辑.

不要过早地进行过度设计。先从独立部署开始。它更简单，也更容易调试。只有在有确凿的理由时才考虑其他方案，而不是因为独立部署看起来"太简单"而放弃使用于生产环境.

部署流程

在 Windows 和 Linux 上部署独立的 WebBroker 应用程序都很简单。流程几乎完全相同：将应用程序编译为 64 位可执行文件(Linux 用户可以使用 PAServer 进行交叉编译)，将可执行文件和资源文件夹复制到服务器，然后配置基本的运行时间设定.

资源文件夹(包含您的 HTML 范本、CSS、JavaScript、映像等)必须位于相对于执行文件的正确位置，或使用配置文件或环境变量透过绝对路径进行设定.


窍门

对于生产环境，请在 Linux 或 Windows 上建立 系统性服务，这样您的应用程序就可以在服务器启动时自动启动，并在崩溃时自动重新启动。这比手动运行可执行文件更可靠.

FastCGI 与 NGINX (RAD Studio 13.0+)

RAD Studio 13.0 中引入的 FastCGI 支持为在 NGINX 后端部署 WebBroker 应用程序提供了现代标准.

它是如何运作的

FastCGI 是 Web 服务器和应用程序进程之间的通讯协议。您的应用程序编译为常规的独立可执行文件(而非函式库或模块)，但它使用 FastCGI 而非 HTTP 与 NGINX 通讯.

架构很简单：你的应用程序作为后台进程运行，监听某个端口(通常为 9000)。 NGINX 接收来自客户端的 HTTP 请求，将其转换为 FastCGI 格式，然后转送给你的应用程序。你的应用程序透过 WebBroker 处理请求，并透过 FastCGI 将响应传送回 WebBroker。最后，NGINX 将最终的 HTTP 响应传回给客户端.

NGINX 配置

以下是 FastCGI 的最小 NGINX 配置:

```
# Define upstream FastCGI server
upstream fastcgi_backend {
    server 127.0.0.1:9000;
    keepalive 32;
}

server {
    listen 80;
    server_name myapp.example.com;

    location / {
        fastcgi_pass fastcgi_backend;

        # Required FastCGI parameters
        fastcgi_param SCRIPT_FILENAME $document_root$fastcgi_script_name;
        fastcgi_param QUERY_STRING $query_string;
        fastcgi_param REQUEST_METHOD $request_method;
        fastcgi_param CONTENT_TYPE $content_type;
        fastcgi_param CONTENT_LENGTH $content_length;
        fastcgi_param PATH_INFO $uri;
        fastcgi_param REQUEST_URI $request_uri;
        fastcgi_param SERVER_PROTOCOL $server_protocol;
        fastcgi_param REMOTE_ADDR $remote_addr;
        fastcgi_param SERVER_NAME $server_name;
        fastcgi_param SERVER_PORT $server_port;
    }
}
```

此配置指示 NGINX 将所有请求转送到您的 FastCGI 应用程序，并包含您的应用程序正确处理请求所需的 HTTP 参数。当然，这只是一个用作范例的最小配置，您的应用程序可能需要不同的 NGINX 配置.

备注

虽然 FastCGI 在 Windows 上理论上可以运行，但 NGINX 和 Apache 在 Linux 上的表现要好得多。对于 Windows 生产环境部署，可以考虑使用独立部署搭配 NGINX（运行在 Linux 上）作为反向代理，或使用 IIS.

传统 Web 服务器整合

Apache 模块和 IIS/ISAPI 扩充代表了 Web 应用程序的传统部署模型。您的 Delphi 程序代码会被编译成一个共享函式库，并加载到 Web 服务器的进程空间.

无状态的问题

这里有一个根本性的注意事项：当你的程序代码作为 Apache 模块或 IIS/ISAPI DLL 运行时，你无法控制 Web 应用程序的生命周期。 Web 服务器会控制它.

Web 服务器可能:

- 随时卸除并重新加载您的模块
- 运行模块的多个独立实例
- 回收应用程序集区(IIS)或进程(Apache)

每次发生这种情况时，内存中的所有状态都会遗失。这包括:

- **会话数据**: 除非持久化到数据库或文件中
- **数据库链接池**: 需要重建
- **内存快取**: 每次重载后都会清除

WebStencils 内建的会话管理 (TWebSessionManager) 将会话储存在内存中。这对于独立部署或 FastCGI 部署来说非常理想，因为进程会持续运作。但对于 Apache/IIS 模块，会话则难以预测.

解决方案：外部会话存储

如果必须使用 Apache 模块或 IIS/ISAPI，请实作应用程序外部持久化的会话储存。两种常见的方法有:

- **基于数据库的会话**: 将会话数据储存在数据库表（或 Redis）中。每个请求都会从数据库读取/写入会话状态.
- **基于档案的会话**: 将会话数据以档案的形式储存在磁盘上。 Web 服务器可以在进程重新启动后存取这些档案.

这两种方法都会增加复杂性和延迟，但当 Web 服务器控制进程生命周期时，它们是必要的。这也是 FastCGI(进程持续运行)通常优于基于模块的部署的原因之一.

备注

如果你的应用程序需要横向扩展，记忆体会话系统也无法满足需求，无论如何都必须实作基于数据库（或 Redis）的会话系统.

其他需要注意的事项

除了状态管理问题之外，Apache 和 IIS 模块还引入了组态复杂性:

路径处理: 在独立应用程序中，/login 指的是 http://yourserver:8080/login。在 Windows ISAPI 模块中，它可能指的是 http://yourserver/WebApp.dll/login。所有 URL 和复位向都需要考虑模块的脚本名称.

权限: Web 服务器的用户帐户需要对您的范本具有读取权限，对日志目录具有写入权限，以及相应的数据库权限。如果这些权限设定错误，有时会导致难以理解的错误.

除错: 您将使用 RAD Studio 的独立版本开发应用程序，但将其部署为模块。这有时会导致难以调试的意外行为。您需要高度依赖生产环境中的日志记录来发现问题，而这些问题有时在开发环境中难以重现.

关于传统 CGI 的说明

WebBroker 也支持传统的 CGI(Common Gateway Interface)，它是 FastCGI 的前身.

CGI 仍然可以作为与旧系统兼容的选项，但对于新部署而言，FastCGI 提供了 CGI 的所有优势，同时避免了这种老旧技术的效能损失和种种弊端。我们在此提及 CGI 仅为完整起见，但我们强烈建议您使用 FastCGI.

Docker 部署

Docker 代表了一种现代化的部署方法：将应用程序及其所有相依性打包到一个容器映像中，该映像可以在任何环境中一致地运行.

对于 WebStencils 应用程序，Docker 提供:

- **一致的环境**: 开发、测试和生产环境运行相同的版本
- **部署简单**: `docker run` 将您的应用程序部署到任何位置
- **依赖隔离**: 您的容器包含了所需的全部 Linux 函式库
- **可扩展性**: 轻松在负载平衡器之后启动多个容器

如何建立 Docker 映像

每个项目都不尽相同，但使用 Delphi 二进制文件建立 Docker 映像却非常简单。由于 Delphi 产生的是原生可执行文件，因此任何精简的 Linux 基础映像都能运行您的应用程序：无需复杂的运行时间依赖项.

以下是一个 WebStencils 应用程序的最小 Dockerfile:

```
FROM debian:13-slim
```

```
WORKDIR /app

# Copy your compiled Linux executable and resources
COPY Linux64/Release/WebStencilsDemo /app/
COPY resources /app/resources

RUN chmod +x /app/WebStencilsDemo

EXPOSE 8080

CMD ["/app/WebStencilsDemo"]
```

基本功能就这些了。五个必备指令：基础镜像、工作目录、复制档案、产生可执行文件、暴露端口和执行.

产生可用于生产环境的 Docker 映像

前面的范例虽然功能齐全，但在建置 Docker 映像方面缺少一些最佳实践。让我们来看看在 GitHub 储存库其中包含的 Dockerfile.

```
FROM debian:13-slim

# Install security updates and clean up in single layer
RUN apt-get update && \
    apt-get upgrade -y && \
    apt-get clean && \
    rm -rf /var/lib/apt/lists/* /tmp/* /var/tmp/*

# Create non-root user
RUN useradd -m -u 1001 appuser && \
    mkdir -p /app/logs /app/data /app/backup && \
    chown -R appuser:appuser /app

WORKDIR /app

# Copy application and set permissions
COPY --chown=appuser:appuser Linux64/Release/WebStencilsDemo /app/WebStencilsDemo
COPY --chown=appuser:appuser ../resources /app/resources

RUN chmod +x /app/WebStencilsDemo

# Switch to non-root user
```

```
USER appuser

# Environment variables
ENV APP_LOG_PATH=/app/logs/app.log \
    APP_RESOURCES_PATH=/app/resources \
    DOCKER_CONTAINER=1 \
    DEMO_MODE=true \
    DEMO_RESET_INTERVAL=900

EXPOSE 8080

VOLUME ["/app/logs", "/app/data"]

# Health check for web service (using bash built-in /dev/tcp - no dependencies
needed)
HEALTHCHECK --interval=30s --timeout=3s --start-period=10s --retries=3 \
    CMD bash -c 'exec 3<>/dev/tcp/localhost/8080 && echo -e "GET /health
HTTP/1.1\r\nHost: localhost\r\nConnection: close\r\n\r\n" >&3 && cat <&3 | grep -q
"healthy" || exit 1'

CMD ["/app/WebStencilsDemo"]
```

生产版本增加了:

- 基础镜像中的**安全性更新**
- **非 root 使用者**(容器不应该以 root 身分执行)
- 用于配置的**环境变量**
- **用于储存容器重新启动和更新后仍然存在的数据的卷**
- **健康检查**，以便 Docker/Kubernetes 可以检测您的应用程序是否有响应

如果您熟悉 Docker/Kubernetes，您仍然会注意到这个 Dockerfile 非常精简，即使它已经可以用于生产环境。这对 Delphi 来说是一个巨大的优势。由于它产生的是编译后的程序代码，因此几乎没有任何依赖项，这使得部署和维护都变得极为简单.

自动化建置方法

使用专用的建置配置，可以从 RAD Studio 自动建置 Docker 映像。这种方法很实用:

1. **Docker 建置配置** - 建立用于 Docker 的新建置配置
2. **自动化脚本** - 在产生后事件期间执行的 PowerShell 脚本
3. **映像建立** - 该脚本会自动建置您的 Docker 映像

工作流程：选择「Docker」配置，建置项目（Ctrl+Shift+F9），Delphi 将建立一个可用于生产环境的容器映像。无需手动执行 Docker 命令.

前提条件和设置

你需要:

- WSL2 已在 Windows 上安装并设定
- 在 WSL2 中安装了 Docker CLI（不建议在此工作流程中使用 Docker Desktop）
- 在 Delphi 中设定的 Linux 平台
- PAServer 用于交叉编译

运行容器化应用程序:

```
# Basic run
docker run -d -p 8080:8080 --name=myapp myapp:latest

# With persistent storage
docker run -d -p 8080:8080 \
  -v /host/logs:/app/logs \
  -v /host/data:/app/data \
  --name=myapp myapp:latest
```

磁盘区（-v 标志）确保您的数据和日志在容器重新启动和更新后仍然存在.

备注

上面的命令将本机文件夹绑定到容器，但它们也可以对应到 Docker 磁盘区（这被认为是一种更好的做法）。此外，这里映射了两个卷，因为 Dockerfile 范例就是这样配置的（日志卷和数据卷）。每个项目都可以有不同的配置和需求.

提供完整范例

WebStencils 演示仓库包含一个完整的 Docker 工作环境，其中包含所有配置文件、脚本和文件。我们建议您直接浏览仓库以了解具体实作细节，而不是在此处重复所有步骤:

GitHub 仓库: github.com/Embarcadero/WebStencilsDemos

该存储库包括:

- 完整的 Dockerfile
- PowerShell 自动脚本

- 建置配置设定
- 部署文件
- 故障排除指南

这是一个不错的起点实现方案，您可以根据自己的项目进行调整.

生产注意事项

无论你选择哪一种部署方式，一些生产环境的问题都是普遍存在的:

SSL/TLS 配置

切勿使用 HTTP 提供生产环境应用程序。即使是内部应用程序，SSL/TLS 也能有效防止凭证窃取和中间人攻击.

对于独立部署: 将 NGINX 放在前端作为反向代理处理 SSL。这通常是最简单的方法.

```
server {
    listen 443 ssl;
    server_name myapp.example.com;

    ssl_certificate /etc/letsencrypt/live/myapp.example.com/fullchain.pem;
    ssl_certificate_key /etc/letsencrypt/live/myapp.example.com/privkey.pem;

    location / {
        proxy_pass http://localhost:8080;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header X-Forwarded-Proto $scheme;
    }
}
```

Let's Encrypt 透过 certbot 提供免费证书.

对于 FastCGI: NGINX 直接处理 SSL。您的应用程序无需接触证书.

对于 Docker: 要么在单独的容器中运行 NGINX，要么使用能够终止 SSL 的负载平衡器.

窍门

如果您不太熟悉 NGINX，而您的需求并非企业级，那么像 Swag 或 NPM（NGINX 代理管理器）这样的项目将大大简化应用程序的安全性和重新导向流程.

记录和监控

生产应用需要可观察的行为.

无论采用单一实例部署或多实例部署，都应考虑集中式日志记录。 RAD Studio 提供了多种存取系统日志的方式，此外还有许多优秀的第三方日志记录解决方案可以扩展其功能，例如 LoggerPro, DataLogger, Spring4D logging 或 QuickLogger.

我们先前展示的 Docker 健康检查功能可以实现自动监控，编排系统可以自动重新启动不健康的容器.

特定环境配置

务必避免将配置数据硬编码到程序代码中。所有关键数据都应使用配置文件或环境变量.

```
// Read from environment variables
DatabaseHost := GetEnvironmentVariable('DB_HOST');
DatabasePassword := GetEnvironmentVariable('DB_PASSWORD');
ApiKey := GetEnvironmentVariable('API_KEY');

// Or from config file
Config := TIniFile.Create(
  TPath.Combine(AppPath, 'config.ini')
);
```

这样一来，无需更改程序代码即可在开发、测试和生产环境中使用不同的设定.

结论

WebBroker 的部署弹性非常出色，但这并不意味着所有选项都一样好。有状态部署选项（独立部署、FastCGI、Docker）更简单可靠，因为它们与 WebStencils 的会话管理和验证机制自然契合。这也允许您的项目将信息快取/储存在内存中，并自行管理应用程序的生命周期.

关键在于：独立部署真正具备生产就绪状态。它并非妥协或临时解决方案，而是一种能够有效处理实际流量的合法架构。在前端部署 NGINX 进行 SSL 处理，即可获得稳定的生产部署。大多数应用程序仅需此即可.

Docker 整合和 FastCGI 支持可在您的需求需要增加复杂性时提供清晰的升级路径。 Docker 提供部署一致性和云端平台整合。 FastCGI 只需极少的配置变更即可让您使用 NGINX 的高级 HTTP 功能.

传统的 Web 服务器模块（Apache、IIS）仍然是有效的选择，效能也很出色，但它们引入了状态管理的复杂性，在某些情况下，对于新应用程序而言，这种权衡并不值得.

13

使用 RAD 服务器与 WebStencils 集成

简介

RAD Server 也受益于新的 WebStencils 函式库. 已开发出特定的整合，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 服务器整合

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;
```

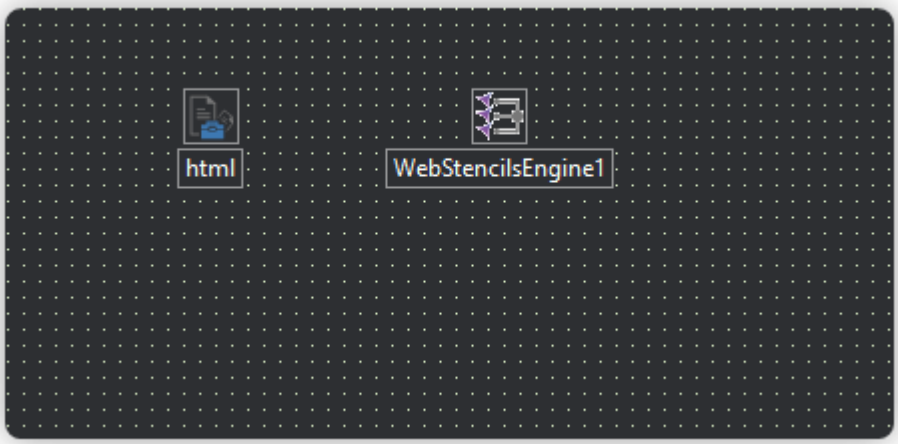
执行此 RAD Server 项目并开启浏览器，您应该能够存取 URL http://localhost:8080/testfile 并查看在变量 LTemplateFile 中定义的模板的渲染内容.

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.

14

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

WebStencils 在线范例

目前有一个基于 WebBroker 和 Docker 的在线范例演示程序可供查看于 wsdemo.embarcadero.com. 无需下载，无需设定：只需打开浏览器即可开始探索.

Embarcadero 部落格文章

目前已有许多关于 WebStencils 的文章。我们会定期更新，这是了解 WebStencils 所有最新消息和功能的最便捷方式.

在这里探索它

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
-

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

进一步扩展 HTMX

尽管 HTML 的声明式特性使得项目对 JavaScript 的依赖性大大降低，但在某些情况下，纯粹的客户端互动仍然需要 JavaScript。例如，考虑将网站从浅色模式切换到深色模式。我们可以在后端完成此操作，并将新的深色模式 HTML 发送回客户端，但由于现代 CSS 库，这在客户端也很容易实现.

当我们需要在客户端添加额外的功能和互动时，有许多微型 JS 函式库可以很好地与 HTML 和 WebStencils 整合.

AlpineJS

AlpineJS 是一个轻量级的 JavaScript 框架，旨在为 HTML 添加简单的互动功能。它提供了一种声明式的方式来操作 DOM 元素，而无需编写大量的 JavaScript 程序代码。它经常被拿来与 Vue 或 React 等框架比较，但它体积更小，更容易整合到现有的 HTML 中。 AlpineJS 非常适合在不增加大型框架复杂性的前提下，为网页新增 JavaScript 行为.

- 应用场景：模态框、下拉式选单、切换开关、窗体验证和其他用户接口交互.
- 主要特性：声明式语法、响应式数据、DOM 操作指令.

文件: AlpineJS Docs

Hyperscript

Hyperscript 是一种脚本语言，旨在简化 HTML 中的事件处理和逻辑。与需要更冗长语法的 JavaScript 不同，Hyperscript 可以直接嵌入到 HTML 属性中。它专注于使用类似自然语言的语法，使事件驱动程序设计更加直观，从而允许开发人员无需复杂的 JavaScript 程序代码即可建立动态行为.

- 应用场景：交互式按钮、窗体提交处理、元素可见性切换.
- 主要特性：自然语言语法、声明式事件、简化常见互动的处理.

文件: Hyperscript Docs

现在就试用 RAD Studio!

了解如何使用一套程序代码库轻松地为多个平台开发原生应用程序!

www.embarcadero.com