NEWS · SEPTEMBER 30, 2026

WebStencils in Delphi: Ihre erste serverseitig gerenderte HTML-Seite

Eine Einführung in WebStencils, die Template-Engine, die seit RAD Studio 12.2 mit Delphi ausgeliefert wird -- wir ergänzen den WebBroker-Server aus dem vorherigen Beitrag um eine HTML-Seite mit Layout, Schleife und Bedingung.

By Dr. Holger Flick



Im vorherigen Beitrag haben wir einen kleinen REST-Server mit WebBroker gebaut, ohne irgendetwas, das nicht ohnehin in Delphi enthalten ist. Er antwortet mit JSON, und genau das möchte ein JavaScript-Frontend oder eine mobile App. Allerdings brauchen etliche Webanwendungen überhaupt kein separates Frontend. Ein Dashboard für das Support-Team, eine Admin-Seite für einen Dienst, eine Statusseite -- in vielen Fällen ist auf dem Server gerendertes HTML die einfachste Lösung, die überhaupt funktionieren kann.

Für diese Aufgabe bringt Delphi ein eigenes Werkzeug mit: WebStencils, eine Template-Engine, die Embarcadero mit RAD Studio 12.2 eingeführt hat. Sie schreiben gewöhnliche HTML-Dateien, markieren die Stellen, an denen Daten hingehören, mit einem `@` und lassen Delphi den Rest ausfüllen.

In diesem Beitrag ergänzen wir den bestehenden Server um unsere erste WebStencils-Seite: ein Layout, eine Seite, die es verwendet, eine Schleife über eine Liste von Objekten und eine Bedingung. Am Ende antwortet dieselbe ausführbare Datei auf einer URL mit JSON und auf einer anderen mit HTML.

Was WebStencils ist und was nicht

Bevor wir Templates schreiben, lohnt es sich zu wissen, wo WebStencils einzuordnen ist. WebBroker kann schon sehr lange HTML erzeugen: mit der Komponente `TPageProducer`, die Tags wie `<#name>` in einer HTML-Datei ersetzt, indem sie für jedes einzelne einen Event-Handler aufruft. Das funktioniert, und es laufen reichlich Anwendungen in Produktion, die das nutzen. WebStencils ist der moderne Nachfolger. Laut Marco Cantùs Einführung implementiert sein Prozessor dasselbe Interface wie der Page Producer und kann ihn ersetzen, die Templates sind aber deutlich ausdrucksstärker: Sie können Eigenschaften von Delphi-Objekten direkt lesen, über Listen iterieren und ein gemeinsames Layout verwenden.

Wenn Sie beides nebeneinander sehen möchten: David Cornelius hat zwei WebBroker-Demos veröffentlicht, die dieselbe Website einmal mit Page Producern und einmal mit WebStencils umsetzen. Für mich war das ein sehr guter Weg, zu verstehen, was sich geändert hat.

Der Ablauf einer WebStencils-Seite ist einfach. Der Prozessor liest ein Template, sucht die Daten heraus, die wir unter einem Namen registriert haben, und erzeugt HTML.



WebStencils führt ein Template mit benannten Delphi-Objekten zusammen; heraus kommt reines HTML

Beachten Sie, was im Diagramm fehlt: Es gibt kein JavaScript-Framework und keinen Build-Schritt. Der Browser bekommt fertiges HTML. Außerdem ist WebStencils nicht an WebBroker gebunden. Derselbe Prozessor funktioniert mit RAD Server, und Marco weist darauf hin, dass er jedes beliebige Textformat erzeugen kann, nicht nur HTML.

WebStencils besteht aus zwei Komponenten. `TWebStencilsProcessor` rendert eine Datei. `TWebStencilsEngine` hält gemeinsame Einstellungen und kann URLs auf Template-Dateien abbilden. Für unsere erste Seite genügt der Prozessor.

Schritt 1: Die Daten für die Seite

WebStencils liest Daten aus ganz gewöhnlichen Delphi-Objekten. Wir registrieren ein Objekt unter einem Namen, und das Template greift mit `@name.Property` auf seine Eigenschaften zu. Legen Sie eine neue Unit `HelloPageModel.pas` mit zwei kleinen Klassen an:

```
unit HelloPageModel;

interface

type
  TPageInfo = class
  private
    FTitle: string;
    FRenderedAt: string;
  public
```

```
    constructor Create(const ATitle: string);
    property Title: string read FTitle;
    property RenderedAt: string read FRenderedAt;
  end;

  TEndpoint = class
  private
    FPath: string;
    FDescription: string;
    FIsHtml: Boolean;
  public
    constructor Create(const APath, ADescription: string; AIsHtml: Boolean);
    property Path: string read FPath;
    property Description: string read FDescription;
    property IsHtml: Boolean read FIsHtml;
  end;

implementation

uses
  System.SysUtils;

{ TPageInfo }

constructor TPageInfo.Create(const ATitle: string);
begin
  inherited Create;
  FTitle := ATitle;
  FRenderedAt := FormatDateTime('yyyy-mm-dd hh:nn:ss', Now);
end;

{ TEndpoint }

constructor TEndpoint.Create(const APath, ADescription: string;
  AIsHtml: Boolean);
begin
  inherited Create;
  FPath := APath;
  FDescription := ADescription;
  FIsHtml := AIsHtml;
end;

end.
```

`TPageInfo` enthält den Seitentitel und den Zeitpunkt, zu dem die Seite gerendert wurde. `TEndpoint` beschreibt eine URL unseres Servers, und `IsHtml` verrät, ob sie HTML oder JSON liefert. Hier gibt es wirklich keine Magie: schlichte Klassen mit schreibgeschützten Eigenschaften. Wir brauchen weder published-Eigenschaften noch Attribute; die offiziellen WebStencils-Demos verwenden genau diese Art von Klassen mit public-Eigenschaften.

Schritt 2: Die Templates

Als Nächstes schreiben wir zwei HTML-Dateien in einem Ordner namens `templates`. Die erste ist das Layout, also der Rahmen, den alle Seiten unserer Website gemeinsam haben:

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>@info.Title</title>
  <style>
    body { margin: 0; padding: 3rem 1.5rem; background: #f5f5f7; color: #1d1d1f;
      font: 17px/1.5 -apple-system, BlinkMacSystemFont, "Segoe UI", Helvetica, Arial, sans-serif;
}
    main { max-width: 40rem; margin: 0 auto; padding: 2.5rem 3rem; background: #fff;
      border-radius: 18px; box-shadow: 0 4px 24px rgba(0, 0, 0, 0.06); }
    h1 { margin: 0 0 0.75rem; font-size: 2.25rem; font-weight: 700; letter-spacing: -0.02em; }
    p { color: #6e6e73; }
    a { color: #0066cc; text-decoration: none; }
    a:hover { text-decoration: underline; }
    ul { padding-left: 1.2rem; }
    li { margin: 0.4rem 0; }
    table { width: 100%; border-collapse: collapse; margin: 1.25rem 0; }
    th { text-align: left; font-size: 0.8rem; text-transform: uppercase;
      letter-spacing: 0.04em; color: #86868b; font-weight: 600; }
    th, td { padding: 0.65rem 0.5rem; border-bottom: 1px solid #e8e8ed; }
    tr:last-child td { border-bottom: 0; }
    hr { border: 0; border-top: 1px solid #e8e8ed; margin: 2rem 0 1rem; }
    small { color: #86868b; font-size: 0.8rem; }
  </style>
</head>
<body>
  <main>
    @RenderBody
    <hr>
    <small>Rendered by WebStencils at @info.RenderedAt -- write @@info to show a literal at
sign.</small>
  </main>
</body>
</html>
```

Drei Dinge sind erwähnenswert:

1. `@info.Title` wird durch die Eigenschaft `Title` des Objekts ersetzt, das wir als `info` registrieren. Das ist die gesamte Syntax, um einen Wert zu lesen.
2. `@RenderBody` markiert die Stelle, an der der Inhalt der eigentlichen Seite eingefügt wird.
3. `@@` erzeugt ein wörtliches `@`. Das brauchen Sie öfter, als Sie denken -- jede E-Mail-Adresse und viele CDN-URLs enthalten eines.

Der `<style>`-Block sorgt nur dafür, dass die Seite ansehnlich aussieht, und WebStencils lässt ihn in Ruhe -- mit einer Ausnahme, die Punkt 3 schön veranschaulicht. WebStencils beansprucht jedes `@` in einem Template für sich, auch die im CSS. Als ich eine `@media`-Regel ergänzt habe, kam sie im Browser als `media.` an, ohne jede Fehlermeldung, einfach als kaputtes CSS. Schreiben Sie stattdessen `@@media`.

Die zweite Datei ist die Seite selbst. Speichern Sie sie als `templates/home.html`:

```
@LayoutPage layout
<h1>@info.Title</h1>
<p>This page and the JSON service run in the same Delphi process.</p>
<ul>
  @ForEach (var endpoint in endpoints) {
  <li>
    <a href="@endpoint.Path">@endpoint.Path</a> -- @endpoint.Description
    @if (endpoint.IsHtml) {
      (HTML)
    } @else {
      (JSON)
    }
  </li>
  }
</ul>
```

Wieder die Anmerkungen:

1. `@LayoutPage layout` in der ersten Zeile besagt: Rendere mich innerhalb von `layout.html`. Der Name wird ohne Dateiendung angegeben.
2. `@ForEach (var endpoint in endpoints) { ... }` wiederholt den Block für jedes Element der Liste, die wir als `endpoints` registrieren. Innerhalb des Blocks ist `endpoint` das aktuelle Element.
3. `@if (endpoint.IsHtml) { ... } @else { ... }` wählt anhand einer booleschen Eigenschaft einen von zwei Blöcken aus.

Schritt 3: Die Seite im Web-Modul rendern

Jetzt verbinden wir die Templates mit dem Web-Modul aus dem vorherigen Beitrag. Im Konstruktor brauchen wir eine weitere Route:

```
  AddRoute('/hello', mtGet, HomeAction);
```

Außerdem benötigen wir eine Hilfsfunktion, die die Templates findet, und den Action-Handler selbst. Fügen Sie die Units `System.IOUtils`, `System.Generics.Collections`, `Web.Stencils` und `HelloPageModel` zur `uses`-Klausel des implementation-Abschnitts hinzu und ergänzen Sie dann Folgendes:

```
function TemplateFileName(const AName: string): string;
begin
  // the templates folder sits next to the executable
  Result := TPath.Combine(
    TPath.Combine(ExtractFilePath(ParamStr(0)), 'templates'), AName);
end;
```

```
procedure THelloModule.HomeAction(Sender: TObject; Request: TWebRequest;
  Response: TWebResponse; var Handled: Boolean);
var
  Processor: TWebStencilsProcessor;
  Endpoints: TObjectList<TEndpoint>;
begin
  Processor := TWebStencilsProcessor.Create(nil);
  try
    // (7) the page to render; it names its own layout
    Processor.InputFileName := TemplateFileName('home.html');

    // (8) the data -- True hands ownership to the processor
    Processor.AddVar('info', TPageInfo.Create('Hello from WebStencils!'), True);

    Endpoints := TObjectList<TEndpoint>.Create;
    Endpoints.Add(TEndpoint.Create('/hello/HelloService/Hello',
      'returns a greeting', False));
    Endpoints.Add(TEndpoint.Create('/hello/HelloService/Add?A=2&B=3',
      'adds two integers', False));
    Endpoints.Add(TEndpoint.Create('/hello', 'this page', True));
    Processor.AddVar('endpoints', Endpoints, True);

    // (9) Content runs the template engine and returns the HTML
    Response.ContentType := 'text/html; charset=utf-8';
    Response.Content := Processor.Content;
  finally
    Processor.Free;
  end;
end;
```

Die Nummerierung setzt dort an, wo der vorherige Beitrag aufgehört hat:

1. `InputFileName` ist das Template, das gerendert werden soll. Wir lassen es auf `home.html` zeigen, und der Prozessor folgt der `@LayoutPage`-Zeile von selbst zu `layout.html` im selben Ordner.
2. `AddVar` registriert ein Objekt unter einem Namen. Der dritte Parameter, `True`, übergibt den Besitz an den Prozessor: Wenn wir den Prozessor freigeben, gibt er `info` und die Liste frei. Da `TObjectList<TEndpoint>` seine Elemente besitzt, werden die Endpoints gleich mit freigegeben. Keine Speicherlecks und kein `try..finally` für jedes einzelne Objekt.
3. Das Lesen von `Content` startet die Engine und liefert das fertige HTML, das wir an die Antwort übergeben.

Sie fragen sich vielleicht, warum wir für jede Anfrage einen neuen Prozessor erzeugen, statt einen auf das Web-Modul zu legen. Das ginge; die offiziellen Demos machen genau das bei ihren einfacheren Beispielen. Ein frischer Prozessor pro Anfrage bedeutet aber, dass Daten aus einer Anfrage niemals in der nächsten auftauchen können. Meiner Meinung nach ist diese Garantie die geringen Kosten wert, pro Anfrage ein Objekt zu erzeugen, und für eine erste Seite bevorzuge ich die Variante, die mich nicht überraschen kann.

Zum Schluss fügen Sie `HelloPageModel` zur `uses`-Klausel von `HelloServer.dpr` hinzu. Die vollständigen Quelltexte finden Sie am Ende dieses Beitrags.

Schritt 4: Starten

Kompilieren und starten Sie den Server. Bevor Sie den Browser öffnen, kopieren Sie allerdings den Ordner `templates` neben `HelloServer.exe`. `TemplateFileName` sucht ihn dort, und wenn Sie in der IDE kompilieren, landet die ausführbare Datei in einem Ausgabeordner wie `Win32\Debug` und nicht neben Ihren Quelltexten. Ich verspreche Ihnen: Das ist der erste Fehler, über den Sie mit WebStencils stolpern werden, und er hat mit den Templates selbst nichts zu tun. Ohne den Ordner antwortet `/hello` mit einem 500 und der Standard-Fehlerseite "Internal Application Error" von WebBroker, auf der `Cannot create file "...\templates\home.html". The system cannot find the path specified` steht. Immerhin verrät die Meldung genau, wo der Server gesucht hat -- allerdings verrät sie das jedem Besucher, lassen Sie es in der Produktion also nicht dabei.



Der erste Fehler, dem Sie begegnen werden: kein Ordner <code>templates</code> neben der ausführbaren Datei, und WebBroker antwortet mit seiner Standard-500-Seite.

Öffnen Sie nun `http://localhost:8080/hello`. Die Überschrift lautet „Hello from WebStencils!“, die Liste zeigt unsere drei URLs, die letzte als HTML gekennzeichnet, und die Fußzeile verrät Ihnen, wann die Seite gerendert wurde. Laden Sie die Seite neu, und die Uhrzeit ändert sich: Das ist serverseitiges Rendering in seiner einfachsten Form. Die JSON-URLs aus dem vorherigen Beitrag funktionieren genau wie zuvor -- und sind jetzt anklickbar.



Die gerenderte Seite: <code>home.html</code> in <code>layout.html</code>, mit Schleife, Bedingung und der Renderzeit in der Fußzeile.

Bevor Sie Benutzereingaben rendern

In diesem Beitrag stammen alle Daten aus unserem eigenen Code. Sobald ein Wert aus einer Anfrage kommt -- ein Query-Parameter, ein Formularfeld, eine Datenbankspalte, die ein Benutzer befüllt hat --, spielt es eine Rolle, ob die Ausgabe HTML-kodiert wird, denn sonst kann ein Benutzer Markup und Skripte in Ihre Seite einschleusen. WebStencils HTML-kodiert standardmäßig jeden `@`-Wert: Eine Beschreibung wie `<b>bold</b> & "quoted"` kommt im Browser als `<b>bold</b> & "quoted"` an, und deshalb steht auch das `&` in unserem Add-Link im Seitenquelltext als `&`. Nur ein Backslash wie in `@\endpoint.Description` schaltet die Kodierung ab -- verwenden Sie ihn also nur für HTML, das Sie selbst erzeugt haben, und niemals für etwas, das ein Benutzer eingegeben hat.

Wo diese Einführung endet

Ich habe diese erste Seite bewusst klein gehalten, und WebStencils kann noch deutlich mehr. Es gibt `@Import` für wiederverwendbare Fragmente, `@switch` für Mehrfachverzweigungen und über `@query` Zugriff auf die Anfrage. Seit RAD Studio 13 arbeitet es außerdem mit der neuen Sitzungsverwaltung und den Authentifizierungskomponenten von WebBroker zusammen, die Embarcaderos Überblick über die Neuerungen in 13.0 beschreibt. Derselbe Überblick erwähnt eine Whitelist, die festlegt, auf welche Member ein Template zugreifen darf. Beachten Sie, dass sie ab Werk nur Nachfahren von `TDataSet` und `TStrings` einschränkt; jede öffentliche Eigenschaft Ihrer eigenen Klassen bleibt lesbar, bis Sie `TWebStencilsProcessor.Whitelist` selbst konfigurieren -- das lohnt sich, wenn Ihre Objekte Daten enthalten, die niemals in einer Seite landen sollten.

> **Alternativen**
> Serverseitige Templates gibt es natürlich nicht nur in Delphi. In der Delphi-Open-Source-Welt bringt DelphiMVCFramework eine eigene Template-Engine mit, und mORMot enthält eine Implementierung von Mustache, einer logikfreien Template-Sprache, die es für viele Sprachen gibt. Außerhalb von Delphi wäre dieselbe Seite ein Jinja-Template in Python oder eine Razor Page in ASP.NET Core. Wenn Ihr Team eines davon bereits kennt, wird sich die WebStencils-Syntax innerhalb von Minuten vertraut anfühlen.

Fazit

Wir haben unseren WebBroker-Server um serverseitig gerendertes HTML erweitert, mit zwei Template-Dateien, einer Unit mit zwei schlichten Klassen und einem Action-Handler. WebStencils liest Eigenschaften registrierter Objekte mit `@name.Property`, iteriert mit `@ForEach`, entscheidet mit `@if` und teilt einen Rahmen zwischen Seiten mit `@LayoutPage` und `@RenderBody`.

> Eine WebStencils-Seite ist HTML mit ein paar `@`-Zeichen darin. Die Delphi-Seite muss nur sagen, welche Objekte unter welchen Namen bereitstehen.

Im nächsten Beitrag ersetzen wir die fest verdrahtete Liste durch echte Daten: Wir legen eine SQLite-Datenbank im Code an -- Datei, Tabelle und Beispieldatensätze -- und rendern eine Kundentabelle direkt aus einer FireDAC-Abfrage. Bis dahin: Ändern Sie die Templates, ergänzen Sie `TPageInfo` um eine Eigenschaft und schauen Sie, was passiert. Viel Spaß!

Vollständiger Quellcode

Zur Referenz hier die beiden Dateien, die sich gegenüber dem vorherigen Beitrag geändert haben. Die unveränderte `HelloWebModule.dfm` und die Templates sind weiter oben abgebildet.

```
program HelloServer;

{$APPTYPE CONSOLE}

uses
  System.SysUtils,
  Web.WebReq,
  IdHTTPWebBrokerBridge,
  HelloPageModel in 'HelloPageModel.pas',
  HelloWebModule in 'HelloWebModule.pas' {HelloModule: TWebModule};
```

```
const
  PORT = 8080;

var
  Server: TIdHTTPWebBrokerBridge;
begin
  try
    // (1) tell WebBroker which class answers the requests
    WebRequestHandler.WebModuleClass := THelloModule;

    // (2) the HTTP server: Indy, bridged to WebBroker
    Server := TIdHTTPWebBrokerBridge.Create(nil);
    try
      Server.DefaultPort := PORT;
      Server.Active := True;

      WriteLn('WebBroker server running on port ', PORT);
      WriteLn('Try: http://localhost:8080/hello');
      WriteLn('Press Enter to stop.');
      ReadLn;

      Server.Active := False;
    finally
      Server.Free;
    end;
  except
    on E: Exception do
    begin
      WriteLn(E.ClassName, ': ', E.Message);
      ExitCode := 1;
    end;
  end;
end.
```

```
unit HelloWebModule;

interface

uses
  System.SysUtils,
  System.Classes,
  System.JSON,
  Web.HTTPApp;

type
  THelloModule = class(TWebModule)
  private
    procedure AddRoute(const APathInfo: string; AMethod: TMethodType;
      AHandler: THTTPMethodEvent; ADefault: Boolean = False);
    procedure SendJson(Response: TWebResponse; AStatusCode: Integer;
      AJson: TJSONObject);
    procedure Cors(Sender: TObject; Request: TWebRequest;
      Response: TWebResponse; var Handled: Boolean);
    procedure HelloAction(Sender: TObject; Request: TWebRequest;
      Response: TWebResponse; var Handled: Boolean);
    procedure AddAction(Sender: TObject; Request: TWebRequest;
      Response: TWebResponse; var Handled: Boolean);
    procedure HomeAction(Sender: TObject; Request: TWebRequest;
      Response: TWebResponse; var Handled: Boolean);
    procedure NotFoundAction(Sender: TObject; Request: TWebRequest;
      Response: TWebResponse; var Handled: Boolean);
  public
    constructor Create(AOwner: TComponent); override;
  end;

implementation
```

```
uses
  System.IOUtils,
  System.Generics.Collections,
  Web.Stencils,
  HelloPageModel;

{$R *.dfm}

function TemplateFileName(const AName: string): string;
begin
  // the templates folder sits next to the executable
  Result := TPath.Combine(
    TPath.Combine(ExtractFilePath(ParamStr(0)), 'templates'), AName);
end;

{ THelloModule }

constructor THelloModule.Create(AOwner: TComponent);
begin
  inherited;

  // (1) runs before any action -- adds the CORS headers
  BeforeDispatch := Cors;

  // (2) one action per URL
  AddRoute('/hello/HelloService/Hello', mtGet, HelloAction);
  AddRoute('/hello/HelloService/Add', mtGet, AddAction);
  AddRoute('/hello', mtGet, HomeAction);

  // (3) the default action answers everything else
  AddRoute('', mtAny, NotFoundAction, True);
end;

procedure THelloModule.AddRoute(const APathInfo: string; AMethod: TMethodType;
  AHandler: THTTPMethodEvent; ADefault: Boolean);
var
  Item: TWebActionItem;
begin
  Item := Actions.Add;
  Item.PathInfo := APathInfo;
  Item.MethodType := AMethod;
  Item.Default := ADefault;
  Item.OnAction := AHandler;
end;

procedure THelloModule.SendJson(Response: TWebResponse; AStatusCode: Integer;
  AJson: TJSONObject);
begin
  try
    Response.StatusCode := AStatusCode;
    Response.ContentType := 'application/json; charset=utf-8';
    Response.Content := AJson.ToJSON;
  finally
    AJson.Free;
  end;
end;

procedure THelloModule.Cors(Sender: TObject; Request: TWebRequest;
  Response: TWebResponse; var Handled: Boolean);
begin
  // (4) CORS: '*' allows EVERY origin, which effectively switches the
  // browser's cross-origin protection off for this server. Fine for
  // development; in production, replace '*' with the origin of your
  // web application, e.g. 'https://app.example.com'.
  Response.SetCustomHeader('Access-Control-Allow-Origin', '*');

  // (5) answer the browser's preflight request right here
  if SameText(Request.Method, 'OPTIONS') then
  begin
```

```
    Response.SetCustomHeader('Access-Control-Allow-Methods', 'GET, OPTIONS');
    Response.SetCustomHeader('Access-Control-Allow-Headers', 'Content-Type');
    Response.StatusCode := 204;
    Handled := True;
  end;
end;

procedure THelloModule.HelloAction(Sender: TObject; Request: TWebRequest;
  Response: TWebResponse; var Handled: Boolean);
begin
  SendJson(Response, 200,
    TJSONObject.Create.AddPair('value', 'Hello from WebBroker!'));
end;

procedure THelloModule.AddAction(Sender: TObject; Request: TWebRequest;
  Response: TWebResponse; var Handled: Boolean);
var
  A, B: Integer;
begin
  // (6) nobody converts the parameters for us -- we do it ourselves
  if TryStrToInt(Request.QueryFields.Values['A'], A) and
    TryStrToInt(Request.QueryFields.Values['B'], B) then
    SendJson(Response, 200,
      TJSONObject.Create.AddPair('value', TJSONNumber.Create(A + B)))
  else
    SendJson(Response, 400,
      TJSONObject.Create.AddPair('error', 'A and B must be integers'));
end;

procedure THelloModule.HomeAction(Sender: TObject; Request: TWebRequest;
  Response: TWebResponse; var Handled: Boolean);
var
  Processor: TWebStencilsProcessor;
  Endpoints: TObjectList<TEndpoint>;
begin
  Processor := TWebStencilsProcessor.Create(nil);
  try
    // (7) the page to render; it names its own layout
    Processor.InputFileName := TemplateFileName('home.html');

    // (8) the data -- True hands ownership to the processor
    Processor.AddVar('info', TPageInfo.Create('Hello from WebStencils!'), True);

    Endpoints := TObjectList<TEndpoint>.Create;
    Endpoints.Add(TEndpoint.Create('/hello/HelloService/Hello',
      'returns a greeting', False));
    Endpoints.Add(TEndpoint.Create('/hello/HelloService/Add?A=2&B=3',
      'adds two integers', False));
    Endpoints.Add(TEndpoint.Create('/hello', 'this page', True));
    Processor.AddVar('endpoints', Endpoints, True);

    // (9) Content runs the template engine and returns the HTML
    Response.ContentType := 'text/html; charset=utf-8';
    Response.Content := Processor.Content;
  finally
    Processor.Free;
  end;
end;

procedure THelloModule.NotFoundAction(Sender: TObject; Request: TWebRequest;
  Response: TWebResponse; var Handled: Boolean);
begin
  SendJson(Response, 404,
    TJSONObject.Create.AddPair('error', 'Not found: ' + Request.PathInfo));
end;

end.
```

INDEPENDENT · AD-FREE · READER-SUPPORTED

Free to read. Not free to make.

Every tutorial, article, and line of example code on Holger's Code is published without a paywall and without ads. This page is about what keeps it that way.

Professional work, published free

The content here is held to the same standard as professional client work: every tutorial is built as a real project, tested against current versions, documented step by step, and revisited when the frameworks move on. A single in-depth tutorial routinely takes days — and that's before the bills behind it: servers, storage, build infrastructure, licenses, and the hardware everything is verified on.

If an article or tutorial here has saved you an afternoon of debugging, taught you a technique, or helped you ship, it created real value for you. Supporting it isn't charity — it's paying for professional work you found useful, at whatever price you choose. And if you can't, keep reading anyway. That's what the free part means.

Ways to support

KO-FI

Buy me a coffee

A one-time tip — no account, no subscription, any amount.

BOOKS & COURSES

Buy the paid work

The paid tier of everything published free here — and you get something lasting in return.

SPREAD THE WORD

Share an article

Send a tutorial to a colleague, cite an article, subscribe to the RSS feed.

Nothing here goes behind a paywall, whether you chip in or not. But if something on this site earned its keep today, this is where to say so.

Support on Ko-fi —