NEWS · SEPTEMBER 28, 2026

Ein WebBroker-REST-Server in Delphi in fünf Schritten: nichts zu installieren

Derselbe minimale REST-Server wie im Beitrag zu TMS XData, neu gebaut ausschließlich mit dem, was Delphi mitbringt -- WebBroker und Indy in einer Konsolenanwendung, mit Routing, JSON und CORS von Hand geschrieben.

By Dr. Holger Flick

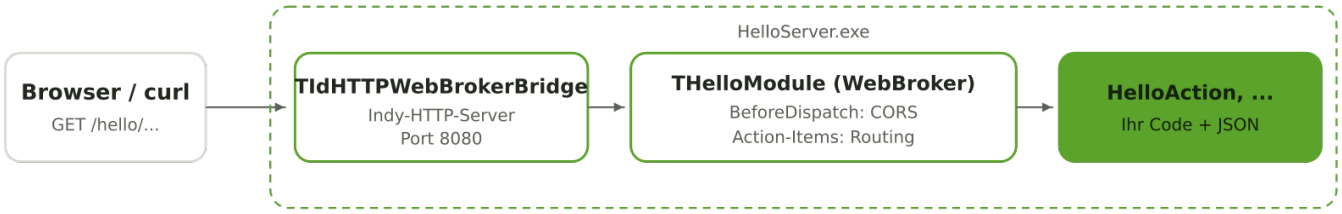
In meinem letzten Beitrag haben wir mit TMS XData einen vollständigen REST-Server in drei kleinen Quelldateien gebaut. Zu jeder Zeile davon stehe ich. TMS XData ist allerdings ein kommerzielles Produkt, und nicht jedes Team hat eine Lizenz -- oder möchte eine Fremdabhängigkeit für einen Service, der Hallo sagt und zwei Zahlen addiert. Die naheliegende Frage lautet also: Wie weit kommen Sie mit dem, was Delphi ohnehin mitbringt?

Ziemlich weit, wie sich zeigt. In diesem Beitrag bauen wir exakt denselben Service -- dieselben URLs, dasselbe JSON, dieselbe CORS-Konfiguration -- mit WebBroker, dem Web-Framework, das seit Jahrzehnten zu Delphi gehört, und dem mitgelieferten HTTP-Server von Indy. Auch diesmal ist es eine schlichte Konsolenanwendung: kein Wizard, keine Komponenten, die irgendwo abgelegt werden, und jede Zeile von Hand getippt.

Gleichzeitig ist dies der erste Teil einer kleinen Serie. WebBroker ist das Fundament, auf dem WebStencils, die Template-Engine von Embarcadero, aufsetzt, und dazu kommen wir im nächsten Beitrag. Zuerst brauchen wir aber einen Server, der antwortet.

Was wir bauen

Bevor wir Code schreiben, lohnt sich ein Blick darauf, welche Teile an einer Anfrage beteiligt sind, denn die Liste unterscheidet sich auf interessante Weise von der XData-Version. Diesmal gibt es kein http.sys. Indy öffnet einen gewöhnlichen Socket, nimmt die Verbindung an und parst HTTP selbst. Die Klasse `TIdHTTPWebBrokerBridge` wandelt die Indy-Anfrage anschließend in das Request-Objekt von WebBroker um und übergibt es an ein Web-Modul, das die passende Aktion auswählt und unseren Code aufruft.



Der Weg einer Anfrage: Indy nimmt sie an, BeforeDispatch ergänzt die CORS-Header, ein Action-Item ruft Ihren Handler auf

Vergleichen Sie das mit dem XData-Diagramm, und Ihnen fallen zwei Dinge auf. Die gesamte Kette lebt jetzt in unserer ausführbaren Datei -- das Betriebssystem stellt nur noch den Socket bereit. Und der Kasten ganz rechts ist größer geworden: WebBroker leitet die Anfrage zwar an unsere Methode weiter, aber Parameter umwandeln und JSON erzeugen ist jetzt unsere Aufgabe. Das ist der Tausch, auf den wir uns einlassen, und wir werden genau sehen, was er kostet.

Die fünf Schritte lauten: das Web-Modul anlegen, die Routen und Handler ergänzen, CORS hinzufügen, das Serverprogramm schreiben und den Service aufrufen. Sie brauchen Delphi und sonst nichts; WebBroker und Indy sind Teil der Installation.

Schritt 1: Das Web-Modul anlegen

In WebBroker werden Anfragen von einem Web-Modul bearbeitet, einer Klasse, die von `TWebModule` abstammt. Legen Sie eine neue Unit namens `HelloWebModule.pas` an. Gefüllt wird sie im nächsten Schritt; vorher müssen wir über eine kleine Datei sprechen.

Ein Web-Modul ist ein naher Verwandter des Datenmoduls. Seine Basisklasse `TCustomWebDispatcher` stammt von `TDataModule` ab, und genau wie ein Formular oder ein Datenmodul erwartet es eine Formulardatei (`.dfm`), die es beim Erzeugen lädt. Der Wizard legt eine an, und wir tun das auch. Erstellen Sie neben der Unit die Datei `HelloWebModule.dfm` mit diesem Inhalt:

```
object HelloModule: THelloModule
end
```

Ja, das ist die ganze Datei. Keine Komponenten, keine Eigenschaften. Sie muss nur existieren, damit die Direktive `{$R *.dfm}` in der Unit etwas zum Linken findet.

Sie fragen sich vielleicht, ob wir sie nicht ganz weglassen könnten. Entfernen Sie die Datei und die Zeile `{$R *.dfm}`, und das Programm kompiliert und startet trotzdem. Sogar die erste Anfrage gelingt, denn unsere Routen entstehen im Code. In der Konsole erscheint allerdings, sobald diese Anfrage eintrifft, Folgendes:

```
WebBroker server running on port 8080
Try: http://localhost:8080/hello/HelloService/Hello
Press Enter to stop.
Exception EResNotFound in module HelloServer.exe at 001000E3.
Resource THelloModule not found.
```

Ohne DFM wird die Anfrage beantwortet, aber die Konsole meldet <code>EResNotFound</code>: "Resource THelloModule not found."

WebBroker erzeugt das Web-Modul erst, wenn eine Anfrage eintrifft, und dessen Konstruktor findet die Ressource nicht. `TWebModule.Create` fängt die Exception ab und meldet sie nur, sodass der Server weiterläuft -- und die Meldung für jede neue Instanz des Web-Moduls erneut ausgibt. In einem vom Wizard erzeugten Modul, dessen Action-Items und Komponenten in der DFM stehen, bliebe das Modul durch dieselbe fehlende Ressource ohne weitere Warnung leer. Zwei Zeilen sind ein kleiner Preis für eine saubere Konsole, und sie stellen auch die IDE zufrieden: Öffnen Sie das Projekt in Delphi, und das Web-Modul erscheint im Designer wie jedes vom Wizard erzeugte.

Schritt 2: Die Routen und Handler ergänzen

Jetzt zur eigentlichen Arbeit. WebBroker leitet Anfragen über Action-Items weiter: Jedes `TWebActionItem` hat einen `PathInfo`, einen `MethodType` und einen `OnAction`-Event-Handler. Normalerweise legen Sie sie im Actions-Editor des Designers an. Wir erzeugen sie stattdessen im Konstruktor, so bleibt alles in einem Listing sichtbar:

```
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 NotFoundAction(Sender: TObject; Request: TWebRequest;
      Response: TWebResponse; var Handled: Boolean);
  public
    constructor Create(AOwner: TComponent); override;
  end;

implementation

{$R *.dfm}

{ 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);

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

end.
```

Die Nummern in den Kommentaren entsprechen den folgenden Anmerkungen:

1. `BeforeDispatch` ist ein Ereignis des Web-Moduls, das ausgelöst wird, bevor WebBroker sich überhaupt ein Action-Item ansieht. Damit ist es der richtige Ort für alles, was jede Antwort braucht. Den CORS-Code selbst schauen wir uns im nächsten Schritt an.
2. `AddRoute` ist eine kleine Hilfsmethode von uns, die ein Action-Item erzeugt und seine vier Eigenschaften setzt. `mtGet` beschränkt beide Routen auf `GET`, ein `POST` an dieselbe URL wird von ihnen also nicht beantwortet.
3. Die Aktion, bei der `Default` auf `True` steht, ruft WebBroker auf, wenn keine andere Aktion die Anfrage bearbeitet hat. Unsere antwortet mit einem 404 und einer JSON-Fehlermeldung, sodass ein Client immer JSON zurückbekommt, selbst bei einem Tippfehler in der URL.
4. Der CORS-Header. Er bekommt weiter unten einen eigenen Abschnitt.
5. Die Antwort auf den Preflight, ebenfalls weiter unten erklärt.
6. Hier zeigt sich der erste echte Unterschied zu XData. XData hat `A` und `B` aus dem Query-String geholt, in Integer umgewandelt und sich selbstständig beschwert, wenn das nicht klappte. In WebBroker liefert uns `Request.QueryFields` die rohen Strings, und `TryStrToInt` erledigt den Rest. Fehlt einer der Werte oder ist er keine Zahl, erhält der Client einen 400 mit einer Erklärung.

Die JSON-Seite ist dank System.JSON erfreulich kurz. `TJSONObject.Create.AddPair(...)` baut das Objekt in einem einzigen Ausdruck, und `SendJson` setzt Status, Content-Type und Inhalt und gibt das Objekt anschließend frei. Diese Besitzübernahme ist Absicht: Jeder Handler erzeugt sein JSON-Objekt und reicht es weiter, so muss niemand an sechs Stellen an ein `try..finally` denken.

Beachten Sie, dass keiner der Handler `Handled` setzt. Es kommt bereits mit `True` an, was so viel bedeutet wie „diese Aktion hat sich um die Anfrage gekümmert“. Auf `False` würden Sie es nur setzen, wenn WebBroker nach einer weiteren Aktion suchen soll.

Schritt 3: CORS hinzufügen

Der Server würde auch ohne die Methode `Cors` laufen, und `curl` würde den Unterschied nie bemerken. Eine Webanwendung schon. Browser setzen die Same-Origin-Policy durch: JavaScript, das von `http://local-host:3000` geladen wurde, darf keine Antworten von `http://localhost:8080` lesen, es sei denn, dieser Server erlaubt es ausdrücklich über CORS-Header. Im XData-Beitrag habe ich die E-Mails beschrieben, die ich genau dazu bekomme, und die Antwort hat sich nicht geändert.

Bei XData hat eine einzige Zeile Middleware zwei Aufgaben erledigt. WebBroker hat keine CORS-Komponente, also erledigen wir beide Aufgaben in `BeforeDispatch` von Hand:

1. Jede Antwort erhält den Header `Access-Control-Allow-Origin`. Das ist die eigentliche Erlaubnis.
2. Bei bestimmten Anfragen -- einem `DELETE` oder allem mit einem JSON-Body -- schickt der Browser zuerst eine sogenannte Preflight-Anfrage mit der Methode `OPTIONS` und fragt, ob die eigentliche Anfrage erlaubt ist. Wir beantworten sie sofort mit den erlaubten Methoden und Headern, einem `204 No Content` und `Handled := True`. Laut der Dokumentation von BeforeDispatch wird eine bereits bearbeitete Anfrage an kein Action-Item mehr weitergereicht, der Preflight erreicht unsere Routen also nie.

Unsere beiden Operationen sind einfache `GET`-Anfragen, die überhaupt keinen Preflight auslösen. Ich habe die Antwort trotzdem eingebaut, denn der erste `POST` mit JSON-Body, den Sie später ergänzen, wird sie brauchen, und ein vergessener Preflight erzeugt denselben verwirrenden Browserfehler wie ein fehlender Header.

> **Das Sternchen ist eine Entwicklungseinstellung**
>
> `Access-Control-Allow-Origin: *` erlaubt **jeder** Website, Ihren Server aus einem Browser heraus aufzurufen. Faktisch schaltet das den Schutz ab. Während der Entwicklung wollen Sie genau das, und für eine wirklich öffentliche, nur lesende API kann es richtig sein. In allen anderen Fällen senden Sie stattdessen den echten Origin Ihrer Webanwendung, zum Beispiel `https://app.example.com`. Deshalb steht die Warnung direkt im Quellcode, genau wie in der XData-Version.

Wie schon zuvor gilt: CORS ist eine Regel, an die sich Browser halten. Es authentifiziert niemanden, und `curl` ignoriert es vollständig.

Schritt 4: Das Serverprogramm schreiben

Jetzt stecken wir die Teile in der Projektdatei zusammen. Erstellen Sie `HelloServer.dpr` mit folgendem Inhalt:

```
program HelloServer;

{$APPTYPE CONSOLE}

uses
  System.SysUtils,
  Web.WebReq,
  IdHTTPWebBrokerBridge,
  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/HelloService/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.
```

Auch hier entsprechen die Nummern den Kommentaren:

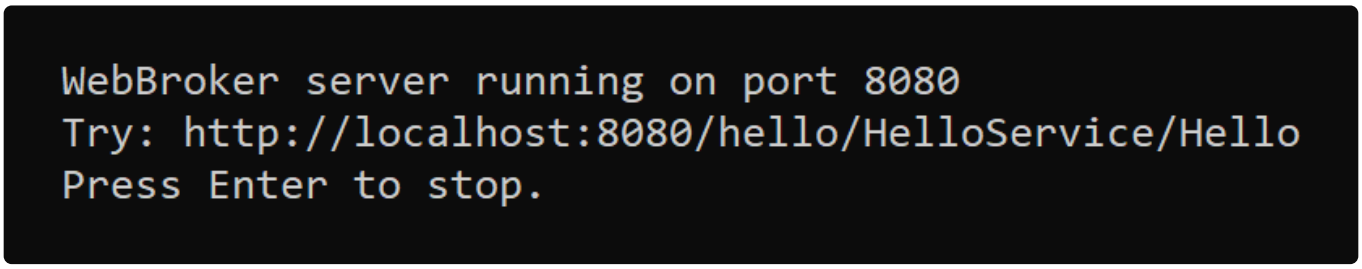
1. `WebRequestHandler` ist das Objekt, das Web-Module erzeugt und Anfragen an sie weiterreicht. Wir teilen ihm mit, welche Klasse es verwenden soll. Beachten Sie, dass wir eine Klasse zuweisen, keine Instanz: WebBroker erzeugt die Web-Module selbst, eines für jede gleichzeitig bearbeitete Anfrage, und hält sie zur Wiederverwendung vor. Ein Feld in `THelloModule` wird also nie von zwei parallelen Anfragen geteilt -- was wichtig wird, sobald wir später in dieser Serie eine Datenbankverbindung ergänzen.
2. `TIdHTTPWebBrokerBridge` ist der Indy-HTTP-Server mit eingebauter WebBroker-Brücke. Port setzen, `Active` auf `True` setzen, und er lauscht.

Sie fragen sich vielleicht, woher `WebRequestHandler` weiß, dass es mit Indy arbeiten soll. Die Unit `IdHTTPWebBrokerBridge` registriert sich in ihrem `initialization`-Abschnitt selbst -- derselbe Trick mit der Selbstregistrierung, den XData für unseren Service verwendet hat. Die Unit in der `uses`-Klausel ist die Verbindung.

Ich habe das Programm bewusst kleiner gehalten als das, was der WebBroker-Wizard erzeugt. Die Konsolenanwendung des Wizards ergänzt Befehle, um den Server zur Laufzeit zu starten, zu stoppen und den Port zu ändern. Das ist bequem, hat aber mit dem Beantworten von Anfragen nichts zu tun. Wir beginnen zu lauschen, warten auf Enter und hören wieder auf.

Schritt 5: Starten und den Service aufrufen

Kompilieren Sie das Projekt und starten Sie `HelloServer.exe`. Und hier kommt der Schritt, den ich nicht schreiben musste: Es gibt keine URL-Reservierung. Indy verwendet kein http.sys, also fragt Windows nicht, wer auf welcher URL lauschen darf, und es ist keine Administrator-Abfrage im Spiel. Je nach Ihren Einstellungen fragt die Windows Defender Firewall beim ersten Start, ob das Programm Verbindungen aus dem Netzwerk annehmen darf. Für Anfragen von Ihrem eigenen Rechner müssen Sie das nicht erlauben.



Der Server lauscht auf Port 8080; mit Enter beenden Sie ihn.

Öffnen Sie einen Browser und geben Sie `http://localhost:8080/hello/HelloService/Hello` ein, oder verwenden Sie `curl` in einem zweiten Konsolenfenster:

```
curl http://localhost:8080/hello/HelloService/Hello
```

Die Antwort ist ein kleines JSON-Dokument:

```
{"value":"Hello from WebBroker!"}
```

Ich habe dieselbe `value`-Hülle verwendet, die XData erzeugt, sodass ein Client, der gegen den XData-Server geschrieben wurde, nur den Port ändern muss. `TJSONObject.ToJSON` schreibt kompaktes JSON ohne Zeilenumbrüche; der Inhalt ist identisch. Die zweite Operation funktioniert ebenfalls:

```
curl "http://localhost:8080/hello/HelloService/Add?A=2&B=3"
```

Sie antwortet mit `{"value":5}`. Probieren Sie jetzt `Add?A=2&B=x`, und Sie erhalten einen 400 mit unserer Fehlermeldung; probieren Sie einen beliebigen anderen Pfad, und Sie erhalten den 404 der Default-Aktion. Schick.

Das ist ein funktionierender REST-Server. Drei Quelldateien, eine davon zwei Zeilen lang, und nichts zu installieren.

Was XData für uns erledigt hat und was wir jetzt selbst tun

Jetzt, da es beide Versionen gibt, sollten wir ehrlich über den Unterschied sprechen, denn er ist der eigentliche Inhalt dieses Beitrags. Die Tabelle listet jede Aufgabe auf und wer sie übernimmt.

Aufgabe	TMS XData	WebBroker
Routing	aus Interface- und Methodennamen abgeleitet	Action-Items mit `PathInfo`
Parameter	automatisch umgewandelt	`QueryFields` plus `TryStrToInt`
JSON	automatisch serialisiert	`System.JSON` von Hand
CORS und Preflight	`TCorsMiddleware`	`BeforeDispatch` von Hand
HTTP-Server	http.sys, braucht URL-Reservierung	Indy, im eigenen Prozess
Kosten	kommerzielle Lizenz	in Delphi enthalten

Bei zwei Operationen ist die handgeschriebene Version völlig überschaubar, und ich würde nicht zögern, sie für einen kleinen internen Service einzusetzen. Meiner Meinung nach ändert sich das Bild aber, wenn der Service wächst. Zwanzig Operationen mit jeweils ein paar Parametern bedeuten zwanzig Blöcke Umwandlungs- und Validierungscode, die es bei XData schlicht nicht gibt, dazu die Serialisierung ganzer Objekte und Listen -- und genau dort verdient sich ein Framework sein Geld. XData veröffentlicht außerdem eine OpenAPI-Beschreibung

des Service und arbeitet mit TMS Aurelius zusammen, sobald eine Datenbank im Spiel ist.

Das Gegenargument verdient sein volles Gewicht. Jede Zeile des WebBroker-Servers ist sichtbar und debugbar, es gibt keine Lizenz zu verlängern, und die Liste der Abhängigkeiten ist leer. Für ein Team, das jedes Fremdpaket rechtfertigen muss -- und mit einigen davon habe ich gearbeitet --, ist das keine Kleinigkeit. Entscheiden Sie nach Größe des Service und Budget, nicht aus Gewohnheit.

Wo dieser minimale Server endet

Wie bei der XData-Version habe ich das absolute Minimum gezeigt, und Sie sollten wissen, was fehlt: kein HTTPS, keine Authentifizierung, kein Logging, keine Datenbank. Indy kann HTTPS, greift dafür aber auf die OpenSSL-Bibliotheken zurück, die Sie zusammen mit Ihrer ausführbaren Datei ausliefern müssen. In

Produktion ist es außerdem üblich, einen Reverse Proxy vor den Service zu stellen und ihm TLS zu überlassen.

Die Konsolenanwendung ist auch nicht die einzige Möglichkeit, ein Web-Modul zu hosten. WebBroker unterstützt mehrere Arten von Webserver-Anwendungen, darunter ISAPI-DLLs für IIS und Apache-Module, und dasselbe `THelloModule` funktioniert in allen. Diese Portabilität ist eine der schönsten Eigenschaften des Designs.

> **Alternativen**
> WebBroker ist nicht die einzige kostenlose Option in Delphi. Open-Source-Frameworks wie DelphiMVCFramework, Horse und mORMot 2 setzen Routing, Serialisierung und Middleware auf einen einfachen HTTP-Server auf und sind in der Community gut etabliert. Außerhalb von Delphi ist ein Service dieser Größe mit Express auf Node.js oder FastAPI in Python ähnlich klein. Der Vorteil von WebBroker: Es ist bereits auf Ihrem Rechner, und WebStencils baut darauf auf.

Fazit

Wir wollten das XData-Beispiel ausschließlich mit Bordmitteln von Delphi nachbauen, und es hat dieselben fünf Schritte gebraucht -- abzüglich der URL-Reservierung, zuzüglich eines eigenen CORS-Handlers. Das Web-Modul braucht seine Formulardatei, selbst eine leere. Action-Items übernehmen das Routing, `BeforeDispatch` ist der Ort für Header, die jede Antwort braucht, und `System.JSON` macht handgeschriebenes JSON so kurz, dass es nicht wehtut.

> WebBroker gibt Ihnen die Anfrage und die Antwort. Alles dazwischen gehört Ihnen -- das ist zugleich sein Preis und sein Charme.

Im nächsten Beitrag behalten wir diesen Server und lassen ihn mit HTML antworten: Wir fügen WebStencils hinzu, die Template-Engine von Embarcadero, und rendern unsere erste Seite aus einem Template mit Layout, Schleife und Bedingung. Kopieren Sie die drei Dateien in einen Ordner, öffnen Sie `HelloServer.dpr`, und sorgen Sie dafür, dass Sie startklar sind!

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 —