NEWS · JULY 13, 2026

Wie ein Web Server wirklich funktioniert: HTTP für Delphi-Entwickler (4/5)

By Dr. Holger Flick

Am Ende von Teil 3 haben wir etwas leicht Absurdes getan: Wir haben eine rohe TCP-Verbindung zu einem Web Server geöffnet und `GET / HTTP/1.1` *von Hand* hineingetippt — und der Server hat uns mit einer Seite geantwortet. Kein Browser, kein Framework, keine Komponenten. Nur Text, der durch einen Socket hinausgeht, und Text, der zurückkommt.

Dieses kleine Kunststück war der eigentliche Sinn der bisherigen Serie. Wenn Sie mitgemacht haben, haben Sie bereits das Protokoll gesprochen, auf dem das Web läuft. Es wurde Ihnen nur noch nicht ordentlich vorgestellt.

Das holen wir heute nach. In diesem Beitrag nehmen wir den Austausch, den Sie von Hand getippt haben, und geben jedem Teil einen Namen: dem Request, der Response, den Methoden, den Status-Codes, den Headern. Dann wechseln wir auf die andere Seite der Leitung und beantworten die Frage, die Web Server so geheimnisvoll wirken lässt: *Was tut dieses Programm eigentlich?* Spoiler — weniger, als Sie denken. Und weil Sie Delphi-Entwickler sind, beweisen wir es in Delphi: ein serverseitiger Request-Handler mit WebBroker und ein Client, der ihn mit `System.Net.HttpClient` aufruft.

Am Ende wird „Web Server" keine magische Blackbox mehr sein, sondern das, was er wirklich ist: ein Programm, das auf einem Port lauscht, strukturierten Text liest und strukturierten Text zurückschreibt.

Ein Web Server ist ein Programm, kein Ort

Streifen Sie die Mythologie ab, und hier ist die komplette Stellenbeschreibung: Ein Web Server ist ein Programm, das auf einem TCP-Port lauscht — üblicherweise Port 80 oder Port 443 für HTTPS —, Verbindungen annimmt, eine Textnachricht namens **HTTP-Request** liest und eine Textnachricht namens **HTTP-Response** schreibt. Danach vergisst er die ganze Sache in der Regel wieder und wartet auf die nächste.

Alles, was Sie in den vorherigen Teilen gelernt haben, fügt sich direkt darunter ein. IP-Adresse und Port aus Teil 1 sagen den Clients, *wo* der Server ist. DNS aus Teil 2 verwandelt einen freundlichen Namen in diese Adresse. Die TCP-Verbindung aus Teil 3 ist die zuverlässige Byte-Röhre, durch die die beiden Programme miteinander sprechen. HTTP — das Hypertext Transfer Protocol, heute definiert in RFC 9110 — ist schlicht die *Sprache*, die über diese Röhre gesprochen wird: ein vereinbartes Format für „das will ich" und „hier ist Ihre Antwort".

Das folgende Diagramm zeigt einen kompletten Round Trip — die Grundform buchstäblich jedes Seitenaufrufs, API-Aufrufs und Downloads im Web.

Lesen Sie es von links nach rechts, von oben nach unten: Der Client fragt, der Server antwortet — über dieselbe TCP-Röhre, die Sie letztes Mal von Hand aufgebaut haben. Beachten Sie, was *nicht* im Bild ist — keine Session, keine Login-Zeremonie, kein fortlaufendes Gespräch. Eine Frage, eine Antwort.

Ein Satz zur Sicherheit, wie versprochen, dann geht es weiter: HTTPS ist genau dasselbe Protokoll, nur durch einen verschlüsselten Kanal auf Port 443 geführt — die Nachrichten sind identisch, sie reisen lediglich in Verschlüsselung verpackt.

Anatomie der beiden Nachrichten

Beide HTTP-Nachrichten — Request und Response — folgen demselben einfachen Aufbau, festgelegt in RFC 9110: eine besondere erste Zeile, dann Header, dann eine Leerzeile, dann ein optionaler Body. Hier ein echter, minimaler Austausch — Byte für Byte genau das, was Sie in Teil 3 getippt haben, und das, was der Server zurückgeschickt hat:

```
GET /hello HTTP/1.1
Host: www.example.com
Accept: text/html
```

Das ist der gesamte Request. Zeile für Zeile:

1. **Die Request-Zeile** — `GET /hello HTTP/1.1`. Drei Teile: die *Methode* (`GET` — welche Art von Aktion), der *Pfad* (`/hello` — welche Ressource) und die Protokollversion.
2. **Header** — `Name: Wert`-Paare, eines pro Zeile. `Host` sagt, welche Website Sie meinen (ein Server kann viele hosten); `Accept` sagt, welche Formate der Client verarbeiten kann.
3. **Eine Leerzeile** — das Zeichen für „ich bin fertig, Sie sind dran".
4. **Ein Body** — hier nicht vorhanden. Ein `GET` bittet um etwas, also gibt es nichts zu senden.

Und das kommt zurück:

```
HTTP/1.1 200 OK
Content-Type: text/html; charset=utf-8
Content-Length: 49

<html><body><h1>Hello from Delphi!</h1></body></html>
```

Gleicher Aufbau, andere erste Zeile:

1. **Die Status-Zeile** — `HTTP/1.1 200 OK`: die Version, ein numerischer *Status-Code* und eine menschenlesbare Phrase.
2. **Header** — hier beschreibt der Server, was er sendet: `Content-Type` sagt, dass diese Bytes HTML-Text sind, `Content-Length` sagt, wie viele es sind.
3. **Leerzeile**, dann der **Body** — das eigentliche Dokument.

Diese Symmetrie lohnt den direkten Vergleich, denn wenn Sie sie einmal gesehen haben, können Sie für den Rest Ihrer Laufbahn jeden HTTP-Traffic-Dump lesen.

Die Kernaussage des Diagramms: Es gibt genau ein Nachrichtenformat zu lernen, verwendet in beiden Richtungen. Die einzige Asymmetrie ist die erste Zeile — der Request formuliert eine *Absicht*, die Response ein *Urteil*.

Methoden: die Verben des Gesprächs

Die Methode ist das erste Wort der Request-Zeile, und für die Grundintuition brauchen Sie nur zwei davon. `GET` bedeutet *„gib mir diese Ressource"* — kein Body, keine Nebenwirkungen erwartet; das sendet Ihr Browser, wenn Sie eine URL eintippen. `POST` bedeutet *„hier sind Daten, mach etwas damit"* — er trägt einen Body, etwa ein abgeschicktes Formular oder eine hochgeladene Datei. Es gibt weitere Verben — `PUT`, `DELETE` und Verwandte, alle definiert in RFC 9110 — und in Teil 5 werden sie plötzlich sehr wichtig, weil REST-APIs sie ganz bewusst einsetzen. Für heute gilt: `GET` liest, `POST` sendet.

Status-Codes: das Antwortvokabular des Servers

Der Status-Code ist das komplette Urteil des Servers, komprimiert auf drei Ziffern, und die vollständige Liste gruppiert sich sauber nach der ersten Ziffer. Die vier, denen Sie ständig begegnen werden:

- **200 OK** — hat geklappt, hier ist Ihre Antwort.
- **301 Moved Permanently** — wohnt jetzt an einer neuen Adresse; der `Location`-Header sagt wo, und Browser folgen ihm automatisch.
- **404 Not Found** — unter diesem Pfad habe ich nichts.
- **500 Internal Server Error** — Ihr Request war in Ordnung; *mein* Code ist explodiert.

Und hier die Erkenntnis, die Menschen, die HTTP verstehen, von Menschen trennt, die es fürchten: **Ein 404 ist ein erfolgreiches Gespräch.** Die Verbindung hat funktioniert, DNS hat funktioniert, der Server lebt, er hat Ihren Request einwandfrei gelesen — und höflich und korrekt geantwortet: „Diese Ressource existiert nicht." Beim Debuggen sagt Ihnen ein 404, dass der gesamte Netzwerk-Stack aus den Teilen 1–3 gesund ist und das Problem schlicht darin liegt, *nach welchem Pfad Sie gefragt haben*. Das ist eine völlig andere Untersuchung als ein Connection Timeout, bei dem überhaupt kein Programm geantwortet hat.

Header und Content-Type: was die Bytes bedeuten

Header sind der Metadaten-Kanal — schlichte `Name: Wert`-Paare, die die Nachricht beschreiben, ohne Teil von ihr zu sein. Von allen verdient `Content-Type` besonderen Respekt: Der Body einer Response ist nur eine Folge von Bytes, und `Content-Type` sagt dem Empfänger, *wie er sie interpretieren soll*. Dieselben Bytes werden als `text/html` als Seite gerendert, als `text/plain` als Quelltext angezeigt und lösen als `application/octet-stream` einen Download aus. Wenn Teil 5 JSON-APIs einführt, ist `Content-Type: application/json` der Header, der die Hauptarbeit leistet.

Eine tiefe Eigenschaft von HTTP müssen wir noch benennen, bevor wir auf die Serverseite wechseln, und sie verdient ihre eigene Box.

> **HTTP ist zustandslos — und Cookies existieren, um das Gegenteil vorzutäuschen**
> Jeder Request steht vollkommen für sich allein. Laut der HTTP-Übersicht auf MDN gibt es keine Verknüpfung zwischen zwei Requests auf derselben Verbindung — der Server antwortet und vergisst. Das ist keine Einschränkung; es ist der Grund, *warum* das Web skaliert: Jeder Server kann jeden Request beantworten, ohne sich an Sie zu erinnern. Aber Warenkörbe und Logins brauchen Gedächtnis, also wurden Cookies erfunden: Der Server gibt dem Client in einem Response-Header ein kleines Token, der Client schickt es mit jedem folgenden Request zurück, und der Server nutzt es, um „Zustand" nachzuschlagen, den er selbst gespeichert hat. Sessions sind Buchführung, geschichtet *auf* ein zustandsloses Protokoll — gut zu wissen, dass es sie gibt, für diese Serie nicht nötig.

Das Denkmodell „Dokumente"

Wechseln Sie nun auf die Serverseite. Die älteste und einfachste Aufgabe eines Web Servers ist das *Ausliefern von Dokumenten*, und das Denkmodell ist fast schon peinlich wörtlich: Der Server nimmt den Pfad aus der Request-Zeile und bildet ihn auf einen Ordner mit Dateien auf der Festplatte ab. Ein Request für `/docs/index.html` wird zu einem Dateizugriff auf `C:\www\docs\index.html`; die Bytes der Datei werden zum Response-Body, der Dateityp wird zum `Content-Type`, und ein `200` kommt in die Status-Zeile. Ist die Datei nicht da — `404`.

Was das Diagramm zeigt: „Eine Website ausliefern" kann etwas so Unspektakuläres sein wie String-Mapping plus Datei-I/O — ein Programm, das jeder Delphi-Entwickler an einem Nachmittag schreiben könnte. Jahrzehntelang *war* das das Web.

Und jetzt die Wende — der wichtigste Satz dieses Beitrags:

> **Nichts in HTTP verlangt, dass die Response aus einer Datei kommt.** Dem Server steht es frei, den Body per Code zu *berechnen*, im Moment der Anfrage — und weil die Response auf der Leitung exakt gleich aussieht, kann der Client den Unterschied nicht erkennen, und es ist ihm auch egal.

Eine Datei lesen und Bytes senden, oder eine Funktion ausführen und Bytes senden: gleiche Status-Zeile, gleiche Header, gleiches Body-Format. Jede dynamische Website, jede API, jeder Backend-Dienst, den Sie je aufgerufen haben, ist nur ein Web Server, der von dieser Freiheit Gebrauch macht. Merken Sie sich diesen Gedanken — er ist die Tür, durch die Teil 5 geht.

> **Eigene Web Server schreibt man nicht, um Dateien auszuliefern — und das ist gut so**
> Der Fairness halber: Für *statische* Dokumente ist das Problem gründlich gelöst. nginx, Apache HTTP Server und Caddy sind hervorragende Open-Source-Server, die Dateien mit Caching, Kompression und

> TLS ausliefern, ohne dass Sie sich darum kümmern müssen — praxiserprobt auf einem riesigen Teil des Webs. Zu *eigenem* Server-Code greifen Sie aus genau einem Grund: wenn die Response **berechnet**
>
> werden muss — aus einer Datenbank, einer Kalkulation, Ihrer Geschäftslogik. Das ist der Fall, um den es im Rest dieses Beitrags geht, und der einzige, für den sich Code lohnt.

Eine Response in Delphi erzeugen: WebBroker

Delphi liefert seit sehr langer Zeit ein Framework für genau diese Aufgabe mit: WebBroker, aufgebaut um die Unit `Web.HTTPApp` und weiterhin aktuell in Delphi 13 Florence, Embarcaderos neuestem Release. Das Design ist erfreulich Delphi-typisch: Ihre Logik kommt in ein `TWebModule` — einen nicht-visuellen, datenmodulähnlichen Container —, das eine Sammlung von `TWebActionItem`-Einträgen enthält. Jedes Action-Item wird gegen den Pfad des eingehenden Requests abgeglichen, und sein `OnAction`-Event-Handler bekommt Request und Response als Parameter — Sie lesen das eine, Sie befüllen das andere.

Eine bewusste Stärke von WebBroker: Ihr Web-Modul weiß nicht und interessiert sich nicht dafür, *wie* es gehostet wird. Der Web Server Application Wizard kann dasselbe Modul als ISAPI-DLL für Microsoft IIS, als Apache-Modul oder als eigenständige ausführbare Datei mit eingebautem HTTP-Listener für die Entwicklung verpacken. Das Hosting-Gerüst wird für Sie generiert; Ihr Code lebt im Modul, und das ist der Teil, den es zu lesen lohnt. Hier ist die Ebene, auf die es ankommt — ein Handler, der `GET /hello` mit generiertem HTML beantwortet:

```
uses
  Web.HTTPApp, System.SysUtils;

procedure TWebModule1.WebModule1HelloAction(Sender: TObject;
  Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
begin
  // Request ist die geparste eingehende Nachricht: Methode, Pfad, Header, Body.
  // Response ist die ausgehende Nachricht: wir setzen ihre Felder, WebBroker sendet sie.

  if Request.PathInfo = '/hello' then
  begin
    Response.ContentType := 'text/html; charset=utf-8';   // was die Bytes bedeuten
    Response.Content :=
      '<html><body><h1>Hello from Delphi!</h1>' +
      '<p>Generated at ' + DateTimeToStr(Now) + '</p></body></html>';
    // Der Status-Code springt automatisch auf 200 OK, sobald Content gesetzt wird.
  end
  else
  begin
    Response.StatusCode := 404;                          // ehrliche Antwort:
    Response.Content := 'Nothing lives at this path.';    // "ich habe dich gehört,
  end;                                                   //  so etwas habe ich nicht"

  Handled := True;  // dem Dispatcher melden: diese Action hat die Response erzeugt
end;
```

Legen Sie das direkt neben die Nachrichtenanatomie, und es bleibt nichts Unerklärtes übrig. `Request.PathInfo` ist der Pfad aus der Request-Zeile — dokumentiert als genau das. `Response.Content` wird zum Body; `Response.ContentType` wird zum `Content-Type`-Header; `Response.StatusCode` wird zum dreistelligen Urteil in der Status-Zeile. Das `DateTimeToStr(Now)` ist das kleine, aber tiefgründige Detail: **Diese Seite existierte nicht, bevor der Request eintraf.** Sie wurde berechnet. Keine Datei auf der Festplatte entspricht `/hello` — und der Client wird es nie erfahren.

Der Weg durch das Framework sieht von Anfang bis Ende so aus.

Das Bild macht die Arbeitsteilung explizit: Alles rund um das Lauschen auf Sockets, das Parsen des rohen Texts in einen `TWebRequest` und das Zurückschreiben Ihrer `TWebResponse` auf die Leitung gehört dem Framework und seinem Host. Das Einzige, was *Ihnen* gehört, ist die Box rechts — der Handler, in dem aus einem Request eine Antwort wird. Das ist das gesamte Versprechen eines serverseitigen Frameworks, in jeder Sprache.

Der Aufruf aus Delphi: System.Net.HttpClient

Nun die Client-Seite, und hier bietet modernes Delphi einen sauberen, plattformübergreifenden HTTP-Client in `System.Net.HttpClient`: die Klasse `THTTPClient` für den Code-first-Einsatz und ihren Komponenten-Wrapper `TNetHTTPClient` (Unit `System.Net.HttpClientComponent`), falls Sie ihn lieber auf ein Formular ziehen und Events verdrahten. Beide sprechen mit *jedem* HTTP-Server — unserem, nginx oder einer REST-API am anderen Ende der Welt —, denn wie wir inzwischen festgestellt haben, ist es überall dasselbe Protokoll.

Dieses Snippet führt dasselbe `GET /hello` aus, das wir oben beantwortet haben, und liest dann alle drei Teile der Response, die wir seziert haben — Status-Zeile, einen Header und den Body:

```
uses
  System.Net.HttpClient, System.SysUtils;

procedure FetchHello;
var
  LClient: THTTPClient;
  LResp: IHTTPResponse;
begin
  LClient := THTTPClient.Create;
  try
    // Eine Zeile sendet Request-Zeile + Header und wartet auf die Antwort.
    LResp := LClient.Get('http://127.0.0.1:8080/hello');

    // Die Status-Zeile, fertig geparst:
    Writeln('Status:       ', LResp.StatusCode, ' ', LResp.StatusText);
    // z. B. "Status: 200 OK" — oder 404, wenn wir nach einem Pfad ohne Action fragen

    // Jeder Header, per Name:
    Writeln('Content-Type: ', LResp.HeaderValue['Content-Type']);
    // z. B. "text/html; charset=utf-8" — was die Body-Bytes bedeuten

    // Und der Body selbst, dekodiert als String:
    Writeln('Body:         ', LResp.ContentAsString);
  finally
    LClient.Free;
  end;
end;
```

Jede Property bildet auf ein Stück des Anatomie-Diagramms ab: `StatusCode` und `StatusText` sind die Status-Zeile; `HeaderValue['Content-Type']` greift in den Header-Block; `ContentAsString` ist der Body. In Teil 3 haben Sie diese Teile extrahiert, indem Sie rohen Text auf einem Socket angestarrt haben. Jetzt übernimmt eine Bibliothek das Starren — aber Sie wissen *genau*, was sie parst, und das ist der Unterschied zwischen dem Benutzen einer Bibliothek und dem Ausgeliefertsein an sie.

Und bemerken Sie etwas still Befriedigendes: Dieser Client kann nicht erkennen — hat keinerlei Möglichkeit zu erkennen —, ob `/hello` aus einer Datei namens `hello.html` kam oder aus einem `DateTimeToStr(Now)`, das in einem `TWebModule` ausgeführt wurde. Das Protokoll verbirgt die Implementierung des Servers vollständig. Das ist kein Zufall; das ist das Design.

Die Serie bis hierher

Dieser Beitrag ist der vierte von fünf; hier die vollständige Karte, wo wir waren und wohin das führt.

> **Netzwerkgrundlagen für Delphi-Entwickler — alle fünf Teile**
> 1. Teil 1 — IP-Adressen, localhost und Ports: wo Programme im Netzwerk wohnen. 2. Teil 2 — DNS, die hosts-Datei und DHCP: wie aus Namen Adressen werden. 3. Teil 3 — TCP-Verbindungen und Sockets: die zuverlässige Byte-Röhre — und HTTP von Hand getippt. 4. **Teil 4 — Wie ein Web Server funktioniert: HTTP und WebBroker** *(Sie sind hier)* 5. Teil 5 — Web Services, REST, JSON und TMS XData: wenn Server mit Daten statt mit Dokumenten antworten.

Das Fazit

Lassen Sie uns das Mysterium ein letztes Mal einstürzen. Ein Web Server ist ein Programm, das auf Port 80 oder 443 lauscht. Er liest eine kleine Textnachricht mit fester Form — Methode, Pfad, Version; Header; Leerzeile; Body — und schreibt eine zurück — Status-Code; Header; Leerzeile; Body. Der Status-Code ist sein Urteil (und ein `404` beweist, dass das Gespräch *funktioniert* hat), die Header sind Metadaten, und `Content-Type` entscheidet, was die Bytes des Bodys bedeuten. Jeder Austausch steht für sich, denn HTTP ist zustandslos. Die Response kann aus einer Datei auf der Festplatte kommen — oder von Ihrem Code erzeugt werden, und der Client kann den Unterschied nicht erkennen. In Delphi bringen `TWebModule`-Actions Sie auf die antwortende Seite dieses Austauschs und `THTTPClient` auf die fragende — und beide bilden eins zu eins auf Nachrichtenteile ab, die Sie nun im Rohformat gelesen haben.

> Ein Web Server ist keine Magie. Er ist ein Programm, das strukturierten Text mit strukturiertem Text beantwortet — und Sie haben seine Sprache bereits von Hand gesprochen.

Und hier ist die Tür, durch die wir morgen gehen: Wenn ein Server eine Response *berechnen* kann, zwingt ihn nichts, HTML für Menschen zurückzugeben. Er kann genauso gut JSON für Programme liefern — gleiches Protokoll, gleiche Status-Codes, gleiche Header, anderer `Content-Type`. **Genau das ist ein Web Service.** In Teil 5 machen wir diesen Sprung: REST, JSON und der Aufbau einer echten Delphi-Service-Schicht mit TMS XData.

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 —