NEWS · JULY 13, 2026

How a Web Server Actually Works: HTTP for Delphi Developers (4/5)

By Dr. Holger Flick

At the end of Part 3 we did something slightly ridiculous: we opened a raw TCP connection to a web server and typed `GET / HTTP/1.1` into it *by hand* — and the server answered us with a page. No browser, no framework, no components. Just text going down a socket and text coming back.

That little stunt was the whole point of this series so far. If you followed along, you have already spoken the protocol that runs the web. You just haven't been introduced to it properly.

That's today's job. In this post we take the exchange you typed by hand and name every part of it: the request, the response, the methods, the status codes, the headers. Then we flip to the other side of the wire and answer the question that makes web servers feel mysterious: *what is that program actually doing?* Spoiler — it's less than you think. And because you're a Delphi developer, we'll prove it in Delphi: a server-side request handler with WebBroker and a client that calls it with `System.Net.HttpClient`.

By the end, "web server" will have stopped being a magic box and become what it really is: a program listening on a port, reading structured text, and writing structured text back.

A web server is a program, not a place

Strip away the mythology and here is the entire job description. A web server is a program that listens on a TCP port — conventionally port 80, or port 443 for HTTPS — accepts connections, reads a text message called an **HTTP request**, and writes a text message called an **HTTP response**. Then it usually forgets the whole thing ever happened and waits for the next one.

Everything you learned in the earlier parts slots directly underneath this. The IP address and port from Part 1 tell clients *where* the server is. DNS from Part 2 turns a friendly name into that address. The TCP connection from Part 3 is the reliable byte pipe the two programs talk through. HTTP — the Hypertext Transfer Protocol, defined today in RFC 9110 — is simply the *language* spoken over that pipe: an agreed format for "here is what I want" and "here is your answer."

The following diagram shows one complete round trip — the shape of literally every page load, API call, and download on the web.

Read it left to right, top to bottom: the client asks, the server answers, over the same TCP pipe you built by hand last time. Notice what's *not* in the picture — no session, no login ceremony, no ongoing conversation. One question, one answer.

One sentence on security, as promised, and then we move on: HTTPS is exactly this same protocol run through an encrypted channel on port 443 — the messages are identical, they just travel wrapped in encryption.

Anatomy of the two messages

Both HTTP messages — request and response — follow the same simple layout, spelled out in RFC 9110: one special first line, then headers, then a blank line, then an optional body. Here is a real, minimal exchange — this is byte-for-byte the kind of thing you typed in Part 3, and the kind of thing the server sent back:

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

That's the entire request. Line by line:

1. **The request line** — `GET /hello HTTP/1.1`. Three pieces: the *method* (`GET` — what kind of action), the *path* (`/hello` — which resource), and the protocol version.
2. **Headers** — `Name: value` pairs, one per line. `Host` says which site you want (one server can host many); `Accept` says what formats the client can handle.
3. **A blank line** — the "I'm done talking, your turn" marker.
4. **A body** — absent here. A `GET` asks for something, so there's nothing to send.

And here is what comes back:

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

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

Same layout, different first line:

1. **The status line** — `HTTP/1.1 200 OK`: the version, a numeric *status code*, and a human-readable phrase.
2. **Headers** — here the server describes what it's sending: `Content-Type` says these bytes are HTML text, `Content-Length` says how many there are.
3. **Blank line**, then the **body** — the actual document.

The symmetry is worth seeing side by side, because once you see it, you can read any HTTP traffic dump for the rest of your career.

The takeaway from the diagram: there is exactly one message format to learn, used in both directions. The only asymmetry is the first line — the request states an *intent*, the response states a *verdict*.

Methods: the verbs of the conversation

The method is the first word of the request line, and for building an intuition you only need two of them. `GET` means *"give me this resource"* — no body, no side effects expected; it's what your browser sends when you type a URL. `POST` means *"here is some data, do something with it"* — it carries a body, like a submitted form or an uploaded file. There are more verbs — `PUT`, `DELETE` and friends, all defined in RFC 9110 — and in Part 5 they'll suddenly matter a lot, because REST APIs lean on them deliberately. For today: `GET` reads, `POST` sends.

Status codes: the server's answer vocabulary

The status code is the server's entire verdict compressed into three digits, and the full list groups neatly by the first digit. The four you'll meet constantly:

- **200 OK** — worked, here's your answer.
- **301 Moved Permanently** — it lives at a new address now; the `Location` header says where, and browsers follow it automatically.
- **404 Not Found** — I don't have anything at that path.
- **500 Internal Server Error** — your request was fine; *my* code blew up.

Here's the insight that separates people who understand HTTP from people who fear it: **a 404 is a successful conversation.** The connection worked, DNS worked, the server is alive, it read your request perfectly — and it answered, politely and correctly, "that resource doesn't exist." When you're debugging, a 404 tells you the entire network stack from Parts 1–3 is healthy and the problem is simply *which path you asked for*. That's a completely different investigation than a connection timeout, where no program answered at all.

Headers and Content-Type: what the bytes mean

Headers are the metadata channel — plain `Name: value` pairs that describe the message without being part of it. Of all of them, `Content-Type` deserves special respect: the body of a response is just bytes, and `Content-Type` is what tells the receiver *how to interpret them*. The same bytes served as `text/html` render as a page, as `text/plain` display as source code, and as `application/octet-stream` trigger a download. When Part 5 introduces JSON APIs, `Content-Type: application/json` is the header doing the heavy lifting.

There's one more deep property of HTTP to name before we switch to the server side, and it deserves its own box.

> **HTTP is stateless — and cookies exist to fake otherwise**

> Each request stands completely alone. Per the HTTP overview on MDN, there is no link between two requests on the same connection — the server answers and forgets. That's not a limitation; it's *why* the web scales: any server can answer any request without remembering you. But shopping carts and logins need memory, so cookies were invented: the server hands the client a small token in a response header, the client sends it back with every subsequent request, and the server uses it to look up "state" it stored itself. Sessions are bookkeeping layered *on top of* a stateless protocol — worth knowing exists, not needed for this series.

The "documents" mental model

Now walk around to the server side. The oldest and simplest job of a web server is *serving documents*, and the mental model is almost embarrassingly literal: the server takes the path from the request line and maps it onto a folder of files on disk. A request for `/docs/index.html` becomes a file read of `C:\www\docs\index.html`; the file's bytes become the response body, the file's type becomes the `Content-Type`, and a `200` goes on the status line. If the file isn't there — `404`.

What the diagram shows is that "serving a website" can be nothing more exotic than string mapping plus file I/O — a program any Delphi developer could write in an afternoon. For decades, that *was* the web.

And now the pivot — the single most important sentence in this post:

> **Nothing in HTTP says the response has to come from a file.** The server is free to *compute* the body with code, on the spot — and because the response looks exactly the same on the wire, the client cannot tell the difference and doesn't care.

Read a file and send bytes, or run a function and send bytes: same status line, same headers, same body format. Every dynamic website, every API, every backend service you've ever called is just a web server exercising that freedom. Hold that thought — it's the doorway Part 5 walks through.

> **You don't write web servers to serve files — and that's fine**
> Fairness note: for *static* documents, the problem is thoroughly solved. nginx, Apache HTTP Server, and Caddy are superb open-source servers that serve files with caching, compression, and TLS handled for you — battle-tested by a huge share of the web. You reach for *your own* server code for exactly one reason: when the response has to be **computed** — from a database, a calculation, your business logic. That's the case the rest of this post is about, and it's the only case worth writing code for.

Generating a response in Delphi: WebBroker

Delphi has shipped a framework for exactly this job for a very long time: WebBroker, built around the `Web.HTTPApp` unit, and still current in Delphi 13 Florence, Embarcadero's latest release. The design is pleasingly Delphi-shaped: you drop your logic into a `TWebModule` — a non-visual data-module-like container — which holds a collection of `TWebActionItem` entries. Each action item is matched against the path of the incoming request, and its `OnAction` event handler gets the request and response as parameters — you read one, you fill in the other.

One deliberate strength of WebBroker is that your web module doesn't know or care *how* it's being hosted. The Web Server Application wizard can wrap the very same module as an ISAPI DLL loaded into Microsoft IIS, an Apache module, or a stand-alone executable with its own built-in HTTP listener for development. The hosting scaffold is generated for you; your code lives in the module, which is the part worth reading. Here is the level that matters — a handler that answers `GET /hello` with generated HTML:

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

procedure TWebModule1.WebModule1HelloAction(Sender: TObject;
  Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
begin
  // Request is the parsed incoming message: method, path, headers, body.
  // Response is the outgoing message: we set its fields, WebBroker sends it.

  if Request.PathInfo = '/hello' then
  begin
    Response.ContentType := 'text/html; charset=utf-8';   // what the bytes mean
    Response.Content :=
      '<html><body><h1>Hello from Delphi!</h1>' +
      '<p>Generated at ' + DateTimeToStr(Now) + '</p></body></html>';
    // Status code defaults to 200 OK when you set Content.
  end
  else
  begin
    Response.StatusCode := 404;                            // honest answer:
    Response.Content := 'Nothing lives at this path.';     // "I heard you,
  end;                                                     //  I have no such thing"

  Handled := True;  // tell the dispatcher this action produced the response
end;
```

Map this straight back to the message anatomy and there is nothing left unexplained. `Request.PathInfo` is the path from the request line — documented as exactly that. `Response.Content` becomes the body; `Response.ContentType` becomes the `Content-Type` header; `Response.StatusCode` becomes the three-digit verdict on the status line. The `DateTimeToStr(Now)` is the small but profound detail: **this page did not exist until the request arrived.** It was computed. No file on disk corresponds to `/hello` — and the client will never know.

The flow through the framework looks like this end to end.

The picture makes the division of labor explicit: everything about listening on sockets, parsing the raw text into a `TWebRequest`, and writing your `TWebResponse` back down the wire belongs to the framework and its host. The only thing that belongs to *you* is the box on the right — the handler where a request becomes an answer. That's the entire promise of a server-side framework, in any language.

Calling it from Delphi: System.Net.HttpClient

Now the client side, and here modern Delphi gives you a clean, cross-platform HTTP client in `System.Net.HttpClient`: the `THTTPClient` class for code-first use, and its component wrapper `TNetHTTPClient` (unit `System.Net.HttpClientComponent`) if you prefer dropping it on a form and wiring events. Both speak to *any* HTTP server — ours, nginx, or a REST API on the other side of the planet — because, as we've now established, it's all the same protocol.

This snippet performs the same `GET /hello` we handled above, then reads all three parts of the response we've dissected — status line, a header, and the body:

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

procedure FetchHello;
var
  LClient: THTTPClient;
  LResp: IHTTPResponse;
begin
  LClient := THTTPClient.Create;
  try
    // One line sends the request line + headers and waits for the answer.
    LResp := LClient.Get('http://127.0.0.1:8080/hello');

    // The status line, parsed for you:
    Writeln('Status:       ', LResp.StatusCode, ' ', LResp.StatusText);
    // e.g. "Status: 200 OK" — or 404 if we ask for a path with no action

    // Any header, by name:
    Writeln('Content-Type: ', LResp.HeaderValue['Content-Type']);
    // e.g. "text/html; charset=utf-8" — what the body bytes mean

    // And the body itself, decoded to a string:
    Writeln('Body:         ', LResp.ContentAsString);
  finally
    LClient.Free;
  end;
end;
```

Every property maps onto a piece of the anatomy diagram: `StatusCode` and `StatusText` are the status line; `HeaderValue['Content-Type']` reaches into the header block; `ContentAsString` is the body. In Part 3 you extracted these by staring at raw text on a socket. Now a library does the staring — but you know *exactly* what it's parsing, which is the difference between using a library and being at its mercy.

And notice something quietly satisfying: this client cannot tell — has no way to tell — whether `/hello` came from a file named `hello.html` or from `DateTimeToStr(Now)` executing inside a `TWebModule`. The protocol hides the server's implementation completely. That is not an accident; it's the design.

The series so far

This post is the fourth of five; here's the full map of where we've been and where this ends up.

> **Networking for Delphi Developers — all five parts**
> 1. Part 1 — IP addresses, localhost, and ports: where programs live on a network. 2. Part 2 — DNS, the hosts file, and DHCP: how names become addresses. 3. Part 3 — TCP connections and sockets: the reliable byte pipe — and typing HTTP by hand. 4. **Part 4 — How a web server works: HTTP and WebBroker** *(you are here)* 5. Part 5 — Web services, REST, JSON, and TMS XData: when servers answer in data instead of documents.

The bottom line

Let's collapse the mystery one final time. A web server is a program listening on port 80 or 443. It reads a small text message with a fixed shape — method, path, version; headers; blank line; body — and writes one back — status code; headers; blank line; body. The status code is its verdict (and a `404` proves the conversation

worked), the headers are metadata, and `Content-Type` decides what the body's bytes mean. Each exchange stands alone, because HTTP is stateless. The response can come from a file on disk — or be generated by your code, and the client can't tell the difference. In Delphi, `TWebModule` actions put you on the answering side of that exchange, and `THTTPClient` puts you on the asking side, and both map one-to-one onto message parts you've now read in the raw.

> A web server isn't magic. It's a program that answers structured text with structured text — and you've already spoken its language by hand.

And here is the doorway we pass through tomorrow: if a server can *compute* a response, nothing forces that response to be HTML for humans. It can just as easily return JSON for programs — same protocol, same status codes, same headers, different `Content-Type`. **That's all a web service is.** In Part 5 we make that leap: REST, JSON, and building a real Delphi service layer with 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 —