NEWS · JULY 10, 2026

IP-Adressen, Ports und localhost: Netzwerk-Grundlagen für Delphi-Entwickler (1/5)

By Dr. Holger Flick



Es gibt einen Moment, der früher oder später jeden Delphi-Desktop-Entwickler ereilt. Sie haben jahrelang ernsthafte Software gebaut — Formulare, Grids, Geschäftslogik, eine Datenbank, die über FireDAC oder, damals, die BDE brav ihren Dienst tat. Dann kommt die Anforderung: *Die App muss mit einer REST-API sprechen*, oder *der Dienst läuft jetzt auf einer anderen Maschine*. Sie verdrahten alles, drücken F9 — und bekommen:

> **Connection refused.**

Abgewiesen — von *wem*? Und *warum*? Die Fehlermeldung setzt ein mentales Modell voraus, das uns nie jemand vermittelt hat. Und das ist kein persönliches Versäumnis, sondern die natürliche Folge einer Laufbahn, in der Netzwerktechnik schlicht keine Rolle spielte. Desktop-Entwicklung brauchte sie nicht. Die Datenbank lag auf derselben Maschine oder war über einen Connection String erreichbar, den jemand aus der IT herüberreichte — und damit war die Geschichte erzählt. Wir haben Pointer, Generics und ORMs gelernt; niemand hat sich mit uns hingesetzt und erklärt, was eine IP-Adresse eigentlich *ist*.

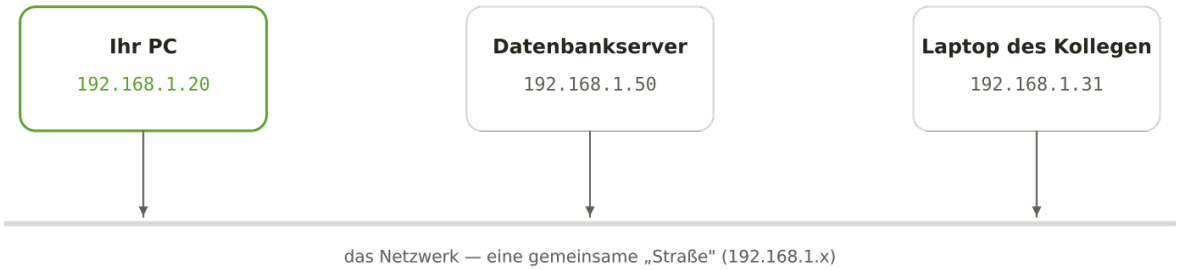
Diese fünfteilige Serie ist genau dieses Gespräch. Keine Subnetz-Arithmetik, keine OSI-Schichten-Liturgie, keine Diagramme von Paket-Headern — nur die praktischen Denkmodelle, die Fehler wie „connection refused" auf einen Blick lesbar machen. Teil 1 behandelt die drei Konzepte, auf denen alles Weitere auf-

baut: **IP-Adressen**, **Ports** und die merkwürdige kleine Adresse namens **localhost**. Am Ende können Sie `127.0.0.1:8080` ansehen, wissen genau, was jeder Teil bedeutet, und diagnostizieren die zwei häufigsten Verbindungsfehler, bevor Sie überhaupt eine Suchmaschine öffnen.

Eine IP-Adresse ist die Hausanschrift eines Geräts

Beginnen wir mit dem einfachsten denkbaren Bild: Ein Netzwerk ist eine Nachbarschaft, und jedes Gerät darin hat eine Hausanschrift. Diese Anschrift ist die **IP-Adresse** — IP steht für Internet Protocol, das Regelwerk, das bestimmt, wie Daten ihren Weg von einem Gerät zum anderen finden. Wenn Ihre App Daten an eine andere Maschine senden will, schickt sie sie nicht an „den Server" im abstrakten Sinn. Sie schickt sie an eine Adresse — genau wie einen Brief.

Hier ist das Bild, das Sie im Kopf behalten sollten — ein kleines Büronetzwerk, in dem jedes Gerät seine eigene Adresse hat.



Jedes Gerät im Netzwerk hat eine eigene IP-Adresse — wie Häuser in einer Straße

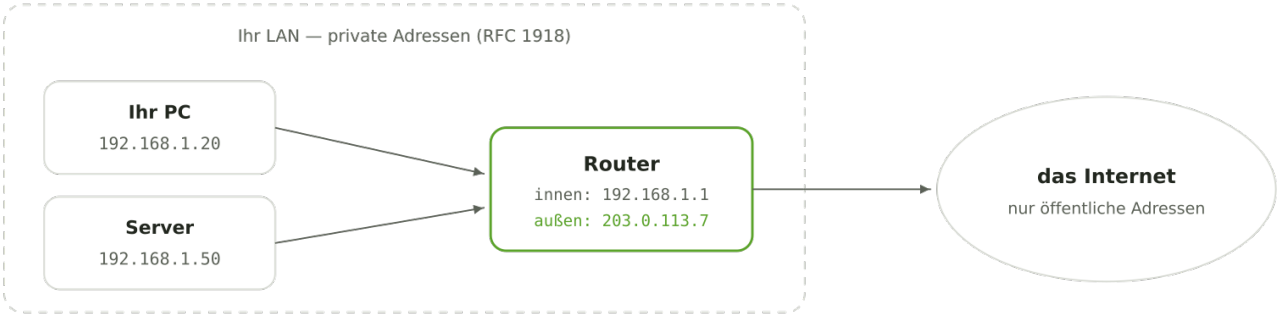
Drei Geräte, drei Adressen, eine gemeinsame Straße. Wenn Ihr PC den Datenbankserver erreichen will, adressiert er seine Daten an `192.168.1.50`, und das Netzwerk stellt sie zu — so wie die Post einen Brief an Hausnummer 50 zustellt, ohne dass Sie die Route des Zustellfahrzeugs kennen müssten.

Die Adressen, die Ihnen überall begegnen — vier Zahlen von 0 bis 255, durch Punkte getrennt, wie `192.168.1.50` — sind **IPv4**-Adressen, die Version des Internet Protocol, die seit den frühen 1980ern den Großteil des Verkehrs trägt. Es gibt auch einen Nachfolger mit deutlich längeren Adressen, IPv6, der existiert, weil der Welt die IPv4-Adressen ausgingen; Sie erkennen seine Adressen an den Doppelpunkten (`::1`, `2001:db8::1`) — und für diese Serie ist dieser eine Satz alles, was Sie darüber wissen müssen.

Private Adressen: Ihr LAN hat seine eigene Nummerierung

Hier ist eine Tatsache, die stillschweigend eine Menge Verwirrung erklärt: **Nicht jede IP-Adresse ist von überall erreichbar.** Bestimmte Adressbereiche sind für private Netzwerke reserviert — Ihr Zuhause, Ihr Büro-LAN — und sie haben nur *innerhalb* dieses Netzwerks eine Bedeutung. Die Reservierung ist offiziell: RFC 1918 legt `10.x.x.x`, `172.16.x.x`–`172.31.x.x` und den Bereich fest, den Sie tausendmal gesehen haben: `192.168.x.x`.

Wenn Sie also `192.168.1.20` in einer Konfigurationsdatei entdecken, wissen Sie ab sofort zwei Dinge auf einen Blick: Das ist eine Maschine in einem lokalen Netzwerk, und diese Adresse bedeutet außerhalb des Gebäudes gar nichts. Ihr Router ist das eine Gerät mit einem Fuß in beiden Welten — er hält die einzige **öffentliche** Adresse des Netzwerks auf der Internetseite und schleust den Verkehr zwischen beiden hin und her.



Private Adressen gelten nur in Ihrem LAN; der Router hält die eine öffentliche Adresse

Die Kernaussage dieses Bilds: Die Geräte links können sich gegenseitig über ihre `192.168.1.x`-Adressen erreichen, aber niemand draußen im Internet kann diese Nummern wählen — er würde stattdessen bei *seinen eigenen* `192.168.1.x`-Geräten landen, denn Millionen von Netzwerken verwenden dieselben privaten Bereiche. Deshalb ist „geben Sie dem Kunden doch einfach Ihre IP" auch selten so einfach, wie es klingt.

localhost: die Maschine spricht mit sich selbst

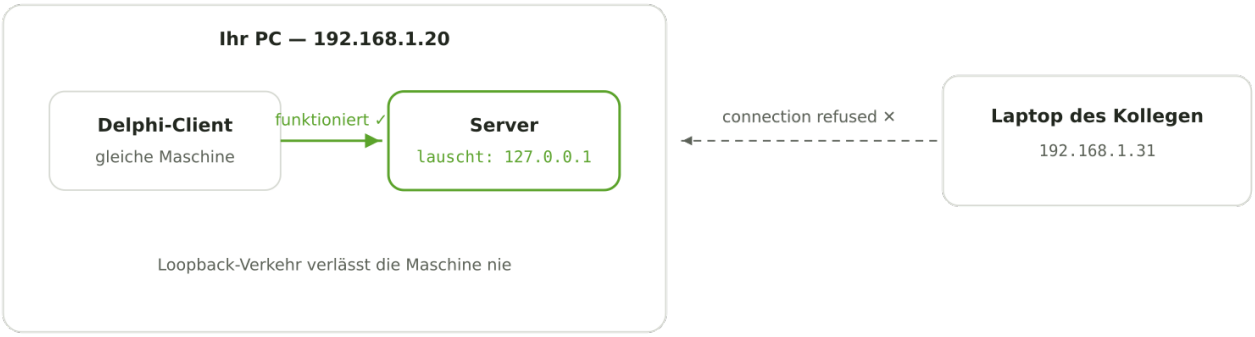
Es gibt eine Adresse, die seltsamer und nützlicher ist als alle anderen: `127.0.0.1`, allgemein bekannt unter dem Spitznamen **localhost**. Sie zeigt auf kein Gerät im Netzwerk — sie zeigt auf *genau diese Maschine*. Daten an `127.0.0.1` berühren weder Netzwerkkarte noch Kabel; das Betriebssystem schleift sie direkt innerhalb des Computers zurück. Deshalb heißt der gesamte reservierte Bereich `127.x.x.x` auch Loopback-Bereich (die Reservierung steht in RFC 1122, und der Name `localhost` selbst ist ein reservierter Name, der immer die lokale Maschine bezeichnet).

Warum sollte eine Maschine je mit sich selbst sprechen müssen? Weil das die perfekte Entwicklungsumgebung ist. Ihr Delphi-Client und der Dienst, den er aufruft, können beide auf Ihrem PC laufen, sich über `127.0.0.1` unterhalten und sich exakt wie ein echtes vernetztes Paar verhalten — ohne zweite Maschine, ohne Firewall-Drama, ohne Latenz.

Aber localhost verbirgt die klassische Falle des Fachs, und die verdient ihre eigene Warnbox.

!!! warning "Die localhost-Falle: „Bei mir funktioniert's" — wörtlich" Ein Serverprogramm entscheidet selbst, auf welchen Adressen es Verbindungen annimmt. Lauscht es nur auf `127.0.0.1` — ein sehr verbreiteter und aus Sicherheitssicht vernünftiger Standard —, dann können **nur Programme auf derselben Maschine es erreichen**. Ihr Delphi-Client läuft auf Ihrem PC wunderbar, und in dem Moment, in dem ein Kollege dieselbe Verbindung von seinem Laptop versucht: *connection refused*. Nichts ist kaputt; der Server öffnet nur die Loopback-Tür. „Bei mir funktioniert's" ist oft die exakte technische Wahrheit: Es funktioniert *nur* auf Ihrer Maschine, weil der Server sich dem Netzwerk nie angeboten hat. Um Verbindungen von anderen Maschinen anzunehmen, muss der Server auf der LAN-Adresse der Maschine lauschen (etwa `192.168.1.20`) oder auf der Sammeladresse `0.0.0.0`, die „jede Adresse dieser Maschine" bedeutet.

Dieses eine Diagramm lohnt sich auswendig zu lernen, denn es erklärt einen erstaunlichen Anteil aller „warum komme ich nicht drauf"-Tickets.



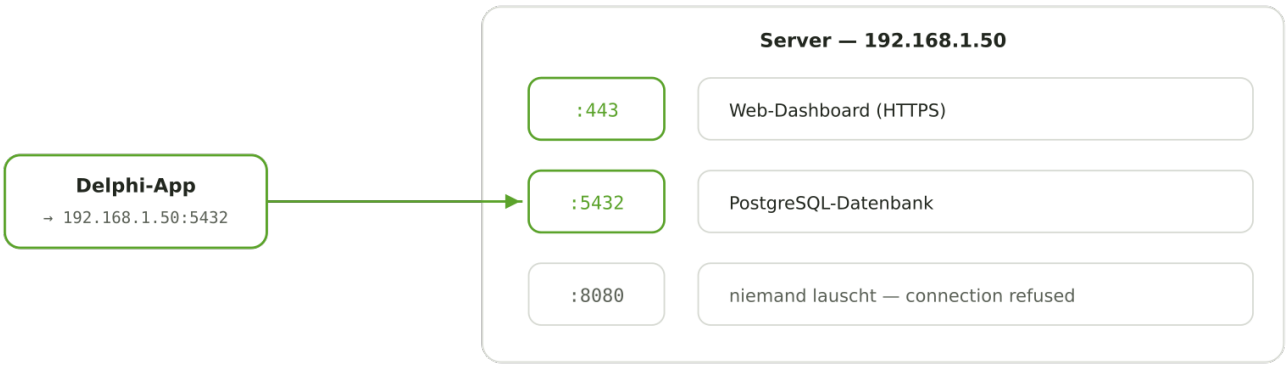
Ein Server, der auf 127.0.0.1 lauscht, antwortet nur der eigenen Maschine — das LAN erreicht ihn nicht

Lesen Sie es von links nach rechts: Der Client auf derselben Maschine verbindet sich problemlos, denn er kann an die Loopback-Tür klopfen. Der Laptop drüben im LAN wird abgewiesen — nicht wegen einer Firewall, nicht wegen eines Bugs, sondern weil der Server schlicht nicht auf der Adresse lauscht, die der Laptop erreichen kann.

Ports: eine Adresse, viele Türen

Eine IP-Adresse bringt Daten zur richtigen *Maschine* — aber auf einer Maschine laufen Dutzende Programme gleichzeitig. Ihr Datenbankserver betreibt nebenbei ein Web-Dashboard, einen Backup-Agenten und die Fernwartung. Wäre die Adresse die ganze Geschichte, kämen eingehende Daten am Gebäude an, ohne zu wissen, für welches Büro sie bestimmt sind.

Genau das lösen **Ports**. Wenn die IP-Adresse die Anschrift eines Mehrfamilienhauses ist, dann ist ein Port eine nummerierte Wohnungstür — und jedes Serverprogramm sucht sich eine Tür aus und *lauscht* dort. Ports sind 16-Bit-Zahlen, jede Maschine hat also die Türen 0 bis 65535, und die Kombination `Adresse:Port` — etwa `192.168.1.50:5432` — bezeichnet einen ganz bestimmten Gesprächsendpunkt: *diese Maschine, dieses Programm*.



Eine Maschine, eine IP-Adresse, viele nummerierte Türen — jede gehört genau einem lauschenden Programm

Achten Sie darauf, was das Diagramm zeigt: Die Delphi-App nennt beide Hälften — die Adresse *und* die Tür — und landet punktgenau bei der Datenbank. Klopfen Sie stattdessen an `:8080`, wo niemand lauscht, weist das Betriebssystem selbst Sie ab. Ports erklären auch, wie Ihr PC zehn Gespräche gleichzeitig führt, ohne sie durcheinanderzubringen: Jedes ist ein eigenes Paar von Endpunkten.

Manche Türnummern sind per Konvention berühmt. Die IANA-Port-Registry — IANA ist die Nummern-Verwaltungsinstanz des Internets — verzeichnet, welcher Dienst traditionell welchen Port nutzt: Webserver auf **80** (HTTP) und **443** (HTTPS), PostgreSQL auf **5432** und so weiter. Das sind Vorgaben, keine Gesetze — jedes Programm darf auf jedem freien Port lauschen —, aber diese Konventionen sind der Grund, warum ein Browser einen Webserver findet, ohne dass Sie je `:443` eintippen. (Der Vollständigkeit halber: Die Ports 0–1023 bilden den „System"-Bereich, für den das Lauschen üblicherweise erhöhte Rechte erfordert, siehe RFC 6335 — gut zu wissen, wenn Ihr Dienst sich auf mysteriöse Weise nicht an Port 80 binden kann.)

Was die zwei klassischen Fehler wirklich bedeuten

Mit Adressen und Ports im Gepäck erklären sich die zwei Fehler aus der Einleitung fast von selbst — sie sind die zwei Seiten derselben Tür-Metapher:

- **Connection refused** — Sie haben an eine Tür geklopft, hinter der *niemand lauscht*. Die Maschine existiert und hat geantwortet, aber die Antwort lautete: „Kein Programm besitzt diesen Port." Unter Windows ist das der Socket-Fehler 10061, WSAECONNREFUSED: "No connection could be made because the target machine actively refused it." Übliche Ursachen: Der Server läuft nicht, er liegt auf einem anderen Port als gedacht — oder er lauscht auf `127.0.0.1`, und Sie klopfen von einer anderen Maschine.
- **Port already in use** — Sie wollten eine Tür *öffnen*, die bereits einem anderen Programm gehört. Pro Port gibt es nur einen Lauscher. Unter Windows ist das der Fehler 10048, WSAEADDRINUSE. Übliche Ursachen: eine zweite Instanz Ihres eigenen Servers oder ein Altprozess, der nie beendet wurde.

Machen wir „connection refused" mit ein paar Zeilen moderner RTL greifbar — keine Fremdkomponenten, nur die plattformübergreifende Unit `System.Net.Socket`, die mit Delphi ausgeliefert wird (aktuelles Release: Delphi 13 Florence). Diese kleine Sonde klopft an eine Tür und berichtet, was passiert:

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

procedure ProbePort(const AHost: string; APort: Word);
var
  LSocket: TSocket;
begin
  LSocket := TSocket.Create(TSocketType.TCP);
  try
    try
      // An eine bestimmte Tür klopfen: Adresse + Port.
      LSocket.Connect('', AHost, '', APort);
      Writeln(Format('%s:%d – a program is listening.', [AHost, APort]));
      LSocket.Close;
    except
      on E: ESocketError do
        // Dort lauscht niemand -> das OS weist die Verbindung ab.
        Writeln(Format('%s:%d – %s', [AHost, APort, E.Message]));
    end;
  finally
    LSocket.Free;
  end;
end;
```

Rufen Sie `ProbePort('127.0.0.1', 5432)` mit laufendem lokalem PostgreSQL auf, und Sie bekommen die freundliche Antwort; rufen Sie `ProbePort('127.0.0.1', 9999)` auf, und die Abweisung kommt aus `TSocket.Connect` zurück — als `ESocketError`, genau die Exception, die Ihre echten Anwendungen abfangen und in eine menschenlesbare Meldung übersetzen sollten („Läuft der Server? Stimmt der Port?"), statt rohen Socket-Text

an Benutzer durchzureichen. Für Dienste, die HTTP sprechen, greifen Sie stattdessen zu `TNetHTTPClient` oder `THTTPClient` — viel mehr dazu in den Teilen 4 und 5.

> **Dieses Wissen ist Toolkit-unabhängig**
> Alles in diesem Beitrag ist eine Wahrheit auf Betriebssystem-Ebene, kein Delphi-Feature. Dieselben Adressen, Ports und Fehler gelten, wenn Sie mit dem Open-Source-Stack Free Pascal / Lazarus, mit Python oder Node.js bauen — und wenn Sie eines Tages den Verkehr selbst *beobachten* wollen, ist der Open-Source-Analyzer Wireshark das Standard-Mikroskop der Branche. Einmal das Modell lernen, überall anwenden.

Sehen Sie es auf Ihrer eigenen Maschine

Nichts festigt ein Denkmodell so sehr, wie es auf der eigenen Hardware laufen zu sehen. Öffnen Sie also ein Terminal (`cmd` oder PowerShell unter Windows) und nehmen Sie sich fünf Minuten für diese vier Schritte.

Schritt 1 — Finden Sie Ihre eigenen Adressen

Führen Sie Windows' `ipconfig` aus, um jede Adresse zu sehen, die Ihre Maschine gerade hält:

```
ipconfig
```

Suchen Sie unter Ihrem aktiven Adapter die Zeile `IPv4 Address` — Sie werden mit an Sicherheit grenzender Wahrscheinlichkeit eine private RFC-1918-Adresse wie `192.168.1.20` sehen. Das sind *Sie*, so wie Ihr LAN Sie kennt. (Unter macOS oder Linux zeigen `ifconfig` oder `ip addr` dasselbe.)

Schritt 2 — Beweisen Sie, dass Loopback existiert

Nutzen Sie `ping`, das „bist du da?"-Werkzeug, gegen die Loopback-Adresse:

```
ping 127.0.0.1
```

Die Antworten kommen deutlich unter einer Millisekunde zurück — denn das „Netzwerk", das hier durchquert wird, liegt vollständig in Ihrer Maschine. Pingen Sie anschließend Ihre eigene LAN-Adresse aus Schritt 1 und dann die Maschine eines Kollegen: dasselbe Werkzeug, drei verschiedene Entfernungen.

Schritt 3 — Sehen Sie, wer an welcher Tür lauscht

Führen Sie `netstat` aus, um jeden Port aufzulisten, der aktuell einem lauschenden Programm gehört — samt Prozess-ID des Besitzers:

```
netstat -ano | findstr LISTENING
```

Jede Zeile zeigt ein lokales `Adresse:Port`-Paar. Und jetzt bedeuten die Adressen etwas: `0.0.0.0:5432` heißt „PostgreSQL, antwortet auf jeder Adresse dieser Maschine", während `127.0.0.1:8000` heißt „dieser Dienst antwortet nur auf Loopback — für das LAN unsichtbar". Sie lesen gerade exakt die Information, die die localhost-Falle von vorhin auflöst.

Schritt 4 — Dieselbe Sicht, auf die moderne Art

Auf aktuellem Windows liefert PowerShells `Get-NetTCPConnection` dieselben Daten als filterbare Objekte:

```
Get-NetTCPConnection -State Listen | Sort-Object LocalPort
```

Nehmen Sie einen beliebigen Port aus der Liste und fragen Sie sich: Welches Programm ist das, und hat es `127.0.0.1` oder `0.0.0.0` gewählt? Wenn Sie das beantworten können, hat dieser Beitrag seinen Zweck erfüllt.

> **Die Serie: Netzwerk-Grundlagen für Delphi-Entwickler**
> Fünf Teile, ein praktisches Fundament — keine Paket-Dumps, keine Mathematik, nur die Modelle, die Netzwerkfehler lesbar machen. 1. IP-Adressen, Ports und localhost **— Sie sind hier** 2. DNS, die hosts-Datei, Servernamen und DHCP — wie aus Namen Adressen werden 3. TCP-Verbindungen und Sockets — was eine „Verbindung" eigentlich ist 4. Wie ein Webserver funktioniert: HTTP und WebBroker — das Protokoll hinter jeder API 5. Web Services, REST, JSON und TMS XData — alles in der Praxis

Das Wichtigste in Kürze

Drei Konzepte, und der Nebel um Netzwerkfehler lichtet sich. Eine **IP-Adresse** identifiziert eine Maschine — private Adressen im Stil von `192.168.x.x` existieren nur innerhalb Ihres LANs, und `127.0.0.1` (localhost) ist immer *genau diese Maschine*, niemals von einer anderen aus erreichbar. Ein **Port** identifiziert ein Programm auf dieser Maschine, `Adresse:Port` benennt also einen exakten Gesprächsendpunkt. Und die zwei klassischen Fehler sind nur die Tür-Metapher im Klartext: *connection refused* heißt, an dieser Tür lauscht niemand; *port already in use* heißt, jemand besitzt sie bereits.

> Eine IP-Adresse findet das Gebäude. Ein Port findet die Tür. `Connection refused` heißt, Sie haben an eine Tür geklopft, hinter der niemand steht — und jetzt wissen Sie genau, welche drei Dinge zu prüfen sind.

Und hier ist der Faden, den wir als Nächstes aufnehmen: Im echten Leben tippen Sie fast nie `192.168.1.50` — Sie tippen `db.mycompany.local` oder `api.example.com`. Irgendwo zwischen Ihrem Tastendruck und der Verbindung wird aus einem *Namen* eine *Adresse*. Diese Übersetzungsmaschinerie — DNS, die hosts-Datei und die Frage, wie Ihre Maschine überhaupt zu ihrer eigenen Adresse kam (DHCP) — ist Teil 2, und dort löst sich eine ganze zweite Familie von „warum komme ich nicht drauf"-Rätseln auf. Bis dahin.

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 —