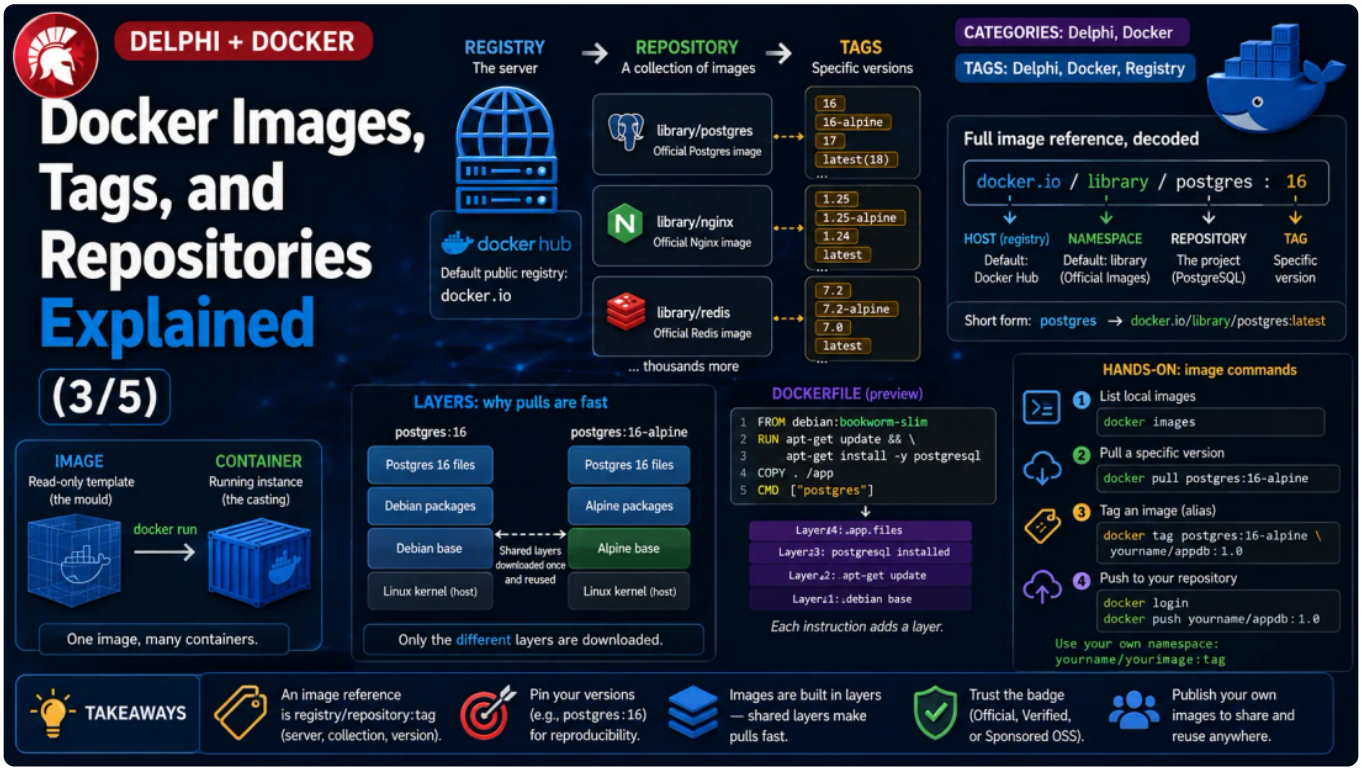
NEWS · AUGUST 26, 2026

Docker-Images, Tags und Repositories erklärt (3/5)

By Dr. Holger Flick



In Teil 2 haben wir etwas getan, das — wenn Sie kurz innehalten und darüber nachdenken — ein kleines bisschen Magie war. Wir haben `postgres` in einen `docker run`-Befehl getippt, und wenige Sekunden später lief ein vollständig konfigurierter PostgreSQL-Datenbankserver und beantwortete Abfragen einer FireDAC-Anwendung — ein Server, den niemand installiert hat, den niemand von einer Hersteller-Website heruntergeladen hat, für den sich niemand durch einen Setup-Assistenten geklickt hat. Wir haben nur das Wort `postgres` gesagt, und Docker hat eine Datenbank hervorgezaubert.

Woher kam sie also?

Diese eine Frage ist das gesamte Thema dieses Beitrags. Die Antwort dreht sich um drei Begriffe, die viele abschrecken — **Image**, **Registry**, **Repository** — plus **Tags** und **Layer**. Jeder davon beschreibt etwas Kleines und Vertrautes. Am Ende dieses Teils lesen Sie `docker.io/library/postgres:16` so, wie Sie einen Dateipfad lesen: Jedes Segment bedeutet etwas Konkretes, und Sie wissen genau, von welchem Server es kam, welche Version Sie bekommen haben und warum der zweite Pull fast augenblicklich fertig ist.

Dies ist die mittlere Station einer fünfteiligen Reise. Hier die vollständige Karte, damit Sie sehen, wo wir stehen.

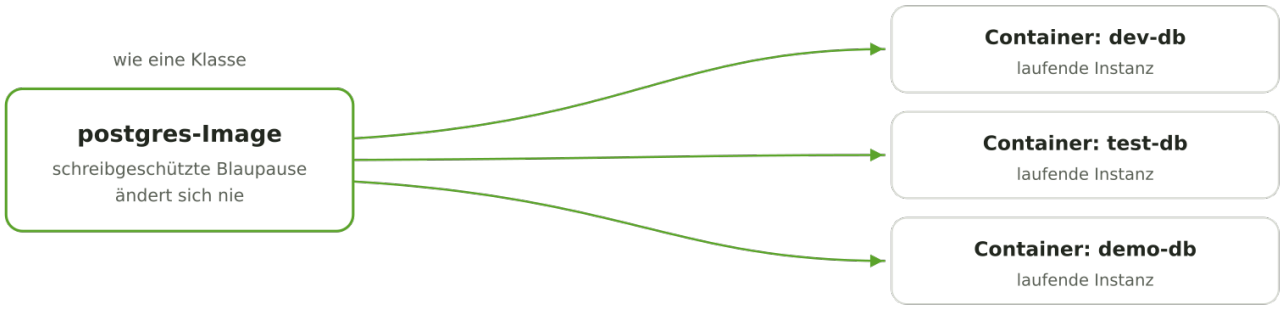
> **Diese Serie: Docker für Delphi-Entwickler**

> 1. Installation, hello-world und die Kernkonzepte — was ein Container wirklich ist 2. Ein Postgres-Container, den Ihre FireDAC-App erreicht — eine echte Datenbank, ohne Installation 3. **Images, Tags, Repositories und Registries — dieser Beitrag** — woher Container kommen 4. Docker Compose — ein ganzer Stack, beschrieben in einer Datei 5. Eine isolierte Multi-Service-Architektur — Web, Backend und Datenbank, sauber getrennt

Image versus Container: die Gussform und der Abguss

Bevor wir aufklären, woher `postgres` kam, muss eine Unterscheidung glasklar sein, denn alles Weitere hängt daran. In Teil 1 haben Sie **Container** gestartet. Was Sie *heruntergeladen* haben, war ein **Image**. Das ist nicht dasselbe, und der Unterschied ist exakt der, den ein Delphi-Entwickler bereits kennt: der zwischen einer Klasse und einem Objekt.

Ein Image ist, in Dockers eigenen Worten, "a standardized package that includes all of the files, binaries, libraries, and configurations to run a container" — ein standardisiertes Paket mit allen Dateien, Binärdateien, Bibliotheken und Konfigurationen, die zum Ausführen eines Containers nötig sind. Es ist schreibgeschützt und ändert sich nie — "once an image is created, it can't be modified": Einmal erstellt, kann ein Image nicht mehr verändert werden. Ein **Container** ist eine laufende Instanz, die *aus* einem Image erzeugt wird. Ein Image, viele Container — genau wie eine Klasse viele Objekte hervorbringt.



Ein schreibgeschütztes Image ist die Blaupause; jeder Container ist eine laufende Instanz davon

Behalten Sie dieses Bild im Kopf. Alles Folgende dreht sich um Images — wo sie liegen, wie sie benannt werden, wie sie versioniert werden und wie sie gebaut werden. Container sind nur das, was Sie bekommen, wenn Sie bei einem Image auf "Start" drücken.

Die Registry: der Server, von dem Images kommen

Als Sie also `postgres` getippt haben, hat Docker ein Image *abgeholt*. Woher? Von einer **Registry**. Eine Registry ist laut Docker-Dokumentation "a centralized location that stores and manages container images" — ein zentraler Ort, der Container-Images speichert und verwaltet. Es ist ein Server im Internet, dessen einzige Aufgabe darin besteht, Images auszuliefern, wenn ein `docker`-Client danach fragt.

Es gibt eine Registry, die Docker als Standard behandelt: Docker Hub, in den Docs beschrieben als "a public registry that anyone can use and is the default registry" — eine öffentliche Registry, die jeder nutzen kann und die als Standard dient. Damit ist das erste Stück Magie entzaubert: Als Sie `postgres` ohne Servernamen eingegeben haben, hat Docker stillschweigend Docker Hub angenommen und von dort gepullt. Es ist dieselbe Bequemlichkeit wie bei einer Shell, die das aktuelle Verzeichnis annimmt, wenn Sie einen bloßen Dateinamen tippen.

Docker Hub ist nicht die einzige Registry — es gibt weitere, und Sie können eine eigene private betreiben — aber weil sie der Standard ist, kommen die meisten ersten Pulls von dort.

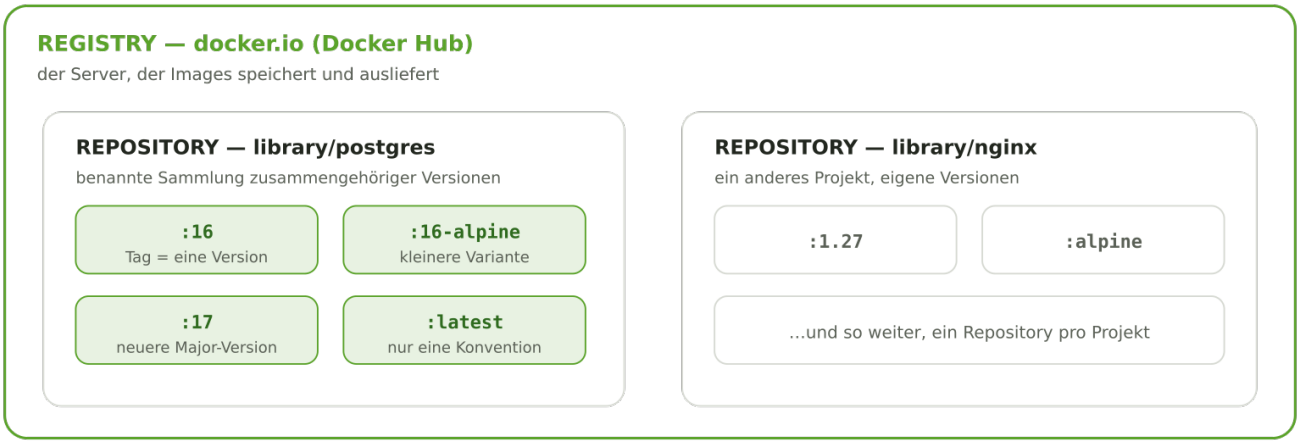
Das Repository: ein Git-Repo voller Image-Versionen

Innerhalb einer Registry sind Images in **Repositories** organisiert. Ein Repository ist "a collection of related container images within a registry" — eine Sammlung zusammengehöriger Container-Images innerhalb einer Registry; Dockers Docs vergleichen es mit "a folder for organizing images by project", einem Ordner zum Organisieren von Images nach Projekt. Das `postgres`-Repository enthält *jede Version* des PostgreSQL-Images: Version 16, Version 17, die schlanken Alpine-Builds — alles nebeneinander unter einem Namen.

Hier die Analogie, mit der es für einen Delphi-Entwickler klick macht — und ich lege Wert darauf, dass es eine Analogie *ist*, keine Gleichsetzung:

> **Das Git-Repo-Denkmodell — eine Analogie, nicht dasselbe**
> Stellen Sie sich ein Docker-Repository so vor wie ein **Git-Repository**: ein benannter Ort, der viele Versionen desselben Projekts enthält, und Sie checken die gewünschte Version über ihr Label aus. Ein Docker-Repository enthält viele Versionen eines Images; Sie pullen die gewünschte Version über ihren **Tag** (dazu gleich mehr). Es ist auch sehr nah an der Funktionsweise eines Paket- oder Komponenten-**Repositorys** in der Delphi-Welt — eine benannte Quelle, die viele versionierte Artefakte ausliefert. Die Mechanik darunter unterscheidet sich (Docker-Versionen sind keine Git-Commits), aber die *Form der Idee* — ein benanntes Zuhause für viele Versionen — ist identisch, und diese Form ist alles, was Sie brauchen, um sich sicher zu bewegen.

Die drei Begriffe verschachteln sich also sauber: Eine **Registry** ist der Server, ein **Repository** ist eine benannte Sammlung zusammengehöriger Images auf diesem Server, und — wie wir gleich sehen werden — ein **Tag** wählt eine bestimmte Version innerhalb dieses Repositorys aus.



Eine Registry enthält Repositories; jedes Repository enthält getaggte Versionen

Lesen Sie das Diagramm von außen nach innen: Der große Rahmen ist eine Registry, jede innere Karte ist ein Repository für ein Projekt, und innerhalb jedes Repositorys sind die kleinen Blöcke die einzelnen Versionen, die Sie pullen können. Diese Verschachtelung ist das gesamte Ablagesystem, das Docker verwendet.

Tags: eine bestimmte Version auswählen

Ein **Tag** ist das Versionslabel, das Sie an einen Repository-Namen anhängen, um zu sagen, *welches* Image Sie wollen. Es ist das `:16` in `postgres:16`. Lassen Sie es weg, füllt Docker für Sie einen Standard-Tag namens `latest` ein — genau das ist in Teil 2 passiert, als Sie ein bloßes `postgres` getippt haben.

Das PostgreSQL-Repository ist ein Docker Official Image, und seine Tag-Liste ist real und öffentlich. Dies sind alles echte, pullbare Tags zum Zeitpunkt des Schreibens:

- `postgres:16` — das neueste 16.x-Release auf der Standard-Debian-Basis
- `postgres:16-alpine` — dasselbe PostgreSQL 16, gebaut auf der winzigen Alpine-Linux-Basis (deutlich kleiner)
- `postgres:17` — das neueste 17.x-Release
- `postgres:latest` — derzeit PostgreSQL 18, weil das die Version ist, die die Maintainer aktuell mit `latest` etikettieren

Diese letzte Zeile leistet im Stillen eine Menge Arbeit — und führt direkt zur nützlichsten Warnung dieses gesamten Beitrags.

> **`latest` bedeutet NICHT 'am neuesten' oder 'automatisch aktualisiert'**
> Es ist das naheliegendste Missverständnis in Docker, und es erwischt jeden einmal. `latest` **ist nur ein konventioneller Tag-Name** — das Label, das Docker verwendet, wenn Sie keines angeben. Es ist kein Live-Zeiger, der dem neuesten Release folgt, und ein erneuter Pull aktualisiert einen laufenden Container nicht automatisch. Dockers eigene Anleitung ist deutlich, was die Konsequenz angeht: Für ein stabiles, reproduzierbares Setup sollten Sie eine konkrete Version festpinnen, statt sich auf `latest` zu verlassen — Docker erlaubt sogar das Pinnen über einen Image-*Digest*, damit "the image you're using is always the same", das verwendete Image also immer dasselbe ist. Die Build-Best-Practices merken an, dass ein `latest`-Tag im Kern bedeutet, dass die Maintainer "suggesting that image be used as the default" — dieses

> Image als Standard vorschlagen. Eine Bequemlichkeit, keine Versionsgarantie. **Faustregel: Pinnen Sie `postgres:16` in allem, das reproduzierbar sein soll; behandeln Sie `postgres:latest` als "gib mir, was die Maintainer gerade als Standard bezeichnen" — für ein schnelles Experiment in Ordnung, für ein Deployment, auf das Sie sich verlassen, riskant.**

Versionen zu pinnen ist die Reproduzierbarkeits-Gewohnheit, die Ihnen in sechs Monaten einen schlechten Nachmittag erspart — wenn `latest` still auf eine neue Major-Version weitergerollt ist und Ihre Schema-Migration darauf nicht vorbereitet war.

Die vollständige Image-Referenz, entschlüsselt

Jetzt können wir das Ganze lesen. Wenn Sie `postgres` tippen, expandiert Docker das zu einer vollständigen Image-Referenz mit präziser Struktur — Dockers Tag-Referenz gibt das Format als `[HOST[:PORT]/]NAME-SPACE/REPOSITORY[:TAG]` an. Jeder bloße Name, den Sie bisher getippt haben, war diese vollständige Form mit automatisch ausgefüllten Standardwerten.



ein bloßes `postgres` bedeutet `docker.io/library/postgres:latest` — drei Standardwerte werden automatisch ergänzt

docker.io/library/postgres:16 — jedes Segment einer vollständigen Image-Referenz

Zwei dieser Standardwerte verdienen ein Wort. Die Registry ist standardmäßig `docker.io` — Docker Hub — wie wir festgestellt haben. Und wenn der Namespace fehlt, setzt Docker `library` ein, das laut Tag-Referenz

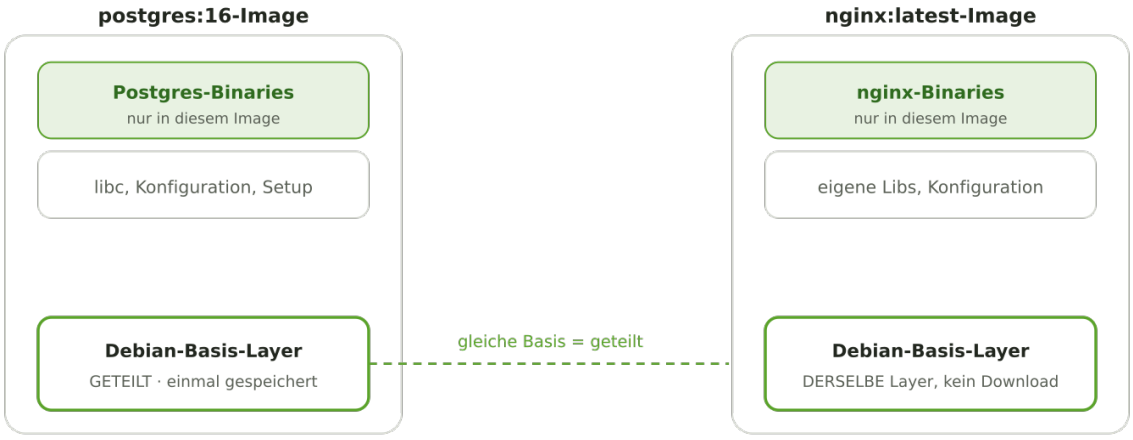
"reserved for Docker Official Images" ist — reserviert für Docker Official Images. Deshalb liegt das offizielle Postgres-Image unter `library/postgres`, und deshalb liegen Ihre *eigenen* Images stattdessen unter Ihrem

Docker-Hub-Benutzernamen, etwa `yourname/myapp` — dazu gleich mehr.

Layer: warum der zweite Pull schnell ist

Hier kommt der Teil, der das ganze System von "vernünftiger Ablage" zu "wirklich cleverem Engineering" macht. Ein Image ist kein monolithischer Klumpen. Laut Docker-Docs gilt: "container images are composed of layers. Each layer represents a set of file system changes that add, remove, or modify files" — Container-Images bestehen aus Layern, und jeder Layer repräsentiert eine Menge von Dateisystem-Änderungen, die Dateien hinzufügen, entfernen oder verändern. Layer sind gestapelt, schreibgeschützt und — entscheidend — **werden zwischen Images geteilt**.

Dieses Teilen ist der Gewinn. Weil zwei Images, die auf derselben Basis aufbauen (etwa demselben Debian-Fundament), *buchstäblich dieselben Basis-Layer wiederverwenden*, speichert und lädt Docker jeden Layer nur ein einziges Mal. Der erste Pull holt alles; das nächste Image, das eine Basis teilt, überspringt die Layer, die Sie bereits haben.



Images stapeln schreibgeschützte Layer, und ein geteilter Basis-Layer wird nur einmal geladen

Lesen Sie die beiden Stapel von unten nach oben: Beide Images sitzen auf *demselben* Debian-Basis-Layer, also hält Docker eine einzige Kopie auf der Platte, und beide Images zeigen darauf. Nur die oberen, image-spezifischen Layer unterscheiden sich. Deshalb fühlt sich Ihr **zweiter** Pull von allem, was eine gemeinsame Basis teilt, augenblicklich an — Sie laden nur die wenigen Layer herunter, die für Sie tatsächlich neu sind.

Das erklärt auch den praktischen Reiz der `-alpine`-Varianten. Alpine Linux ist eine bewusst winzige Basis, daher bringt `postgres:16-alpine` weit weniger Betriebssystem mit als das Debian-basierte `postgres:16`. Dasselbe PostgreSQL, kleinerer Fußabdruck — praktisch, wenn Image-Größe oder Download-Zeit zählen.

Eine kleine Vorschau: jede Anweisung ist ungefähr ein Layer

Man baut keine Images, um sie zu *benutzen*, und das richtige Erstellen von Images heben wir uns für später in der Serie auf. Aber ein Blick auf drei Zeilen macht die Layer-Idee greifbar. Images werden aus einer Textdatei mit Anweisungen gebaut, dem Dockerfile — und in erster Näherung fügt jede Anweisung einen Layer über dem vorherigen hinzu:

```
FROM postgres:16          # Layer 0: mit dem offiziellen Postgres-Image starten
COPY init.sql /init.sql   # Layer 1: eine Datei hinzufügen
ENV POSTGRES_DB=appdb     # Layer 2: einen Standard-Datenbanknamen setzen
```

Das ist eine *Vorschau*, keine Lektion — richtig bauen wir so etwas in einem späteren Teil. Der Punkt für jetzt ist schlicht: Layer sind nichts Abstraktes, sie sind das direkte Ergebnis von Build-Schritten, und einen unteren Layer wiederzuverwenden heißt, die frühere Arbeit aller anderen wiederzuverwenden.

Trusted Content: woran Sie erkennen, dass ein Image sicher ist

Da jeder auf Docker Hub veröffentlichen kann, ist eine berechtigte Frage: Woher weiß ich, dass das `postgres`-Image das *echte* ist und nicht etwas, das ein Fremder hochgeladen hat? Docker Hub beantwortet das mit sichtbaren Trusted-Content-Badges. **Docker Official Images** (wie `postgres`) sind Dockers eigene kuratierte

Repositories, "regularly updated" (regelmäßig aktualisiert) und "some of the most secure images on Docker Hub" — einige der sichersten Images auf Docker Hub. Darüber hinaus kennzeichnet ein **Docker-Verified-Publisher**-Badge Images "from commercial publishers verified by Docker" (von kommerziellen, durch Docker verifizierten Herausgebern), und ein **Docker-Sponsored-Open-Source**-Badge markiert "trusted, secure, and active open-source projects" — vertrauenswürdige, sichere und aktive Open-Source-Projekte. Wenn Sie eines dieser drei Badges auf einer Repository-Seite sehen, blicken Sie auf Inhalte, für die Docker bürgt — ein einfaches, positives Signal, dass Sie aus einer guten Quelle pullen.

Hands-on: Images lesen und bewegen

Genug Konzept — nutzen wir die Befehle, jeder einzelne geprüft gegen Dockers CLI-Referenz. Arbeiten Sie sie der Reihe nach durch.

Schritt 1 — Die vorhandenen Images auflisten

Sehen Sie nach, was auf Ihrem Rechner liegt, und lesen Sie die Spalten für Repository, Tag und Größe, die Sie jetzt verstehen — mit `docker images`:

```
docker images
```

Jede Zeile zeigt `REPOSITORY`, `TAG`, eine Image-`ID` und `SIZE`. Nach Teil 2 sehen Sie hier eine `postgres`-Zeile — das ist das Image, aus dem Ihr Datenbank-Container erzeugt wurde; es liegt im lokalen Speicher und muss nie wieder heruntergeladen werden.

Schritt 2 — Eine konkrete, gepinnte Version pullen

Holen Sie den kleineren Alpine-Build von PostgreSQL 16 explizit, mit `docker pull` und einem echten Tag:

```
docker pull postgres:16-alpine
```

Achten Sie auf die Ausgabe: Docker lädt jeden Layer separat herunter, und jeder Layer, den Sie bereits haben (etwa geteilt mit `postgres:16`), wird mit "Already exists" markiert und übersprungen. Dieses Überspringen ist das Layer-Sharing-Diagramm — live vor Ihren Augen.

Schritt 3 — Einem Image einen eigenen Namen geben

Legen Sie mit `docker tag` einen Alias für ein vorhandenes Image an; die Syntax lautet `docker tag SOURCE_IMAGE[:TAG] TARGET_IMAGE[:TAG]`:

```
docker tag postgres:16-alpine yourname/appdb:1.0
```

Das kopiert nichts — es fügt nur einen zweiten Namen hinzu, der auf dieselben Layer zeigt. Beachten Sie, dass das Ziel *Ihren* Namespace trägt (`yourname/`) statt `library/`, weil es für Ihr eigenes Repository bestimmt ist, nicht für das offizielle.

Schritt 4 — In das eigene Repository pushen (konzeptionell)

Um ein Image zu teilen, melden Sie sich an und `docker push`en es in ein Repository, das Ihnen gehört. Sie brauchen zuerst einen kostenlosen Docker-Hub-Account, dann:

```
docker login
docker push yourname/appdb:1.0
```

Docker lädt nur die Layer hoch, die Docker Hub noch nicht hat, und Ihr Image liegt nun im Repository `yourname/appdb` unter dem Tag `1.0` — pullbar von überall, von Ihnen oder Ihrem Team, genau so, wie Sie `postgres` gepullt haben. Der Kreis ist geschlossen: Sie sind jetzt *Publisher* auf derselben Registry, von der Sie bisher nur konsumiert haben.

Sie sind nicht auf Docker Hub beschränkt

Docker Hub ist der Standard und ein guter Startpunkt, aber fairerweise ist es eine Option unter mehreren starken, und das gesamte Modell ist offen. Der führende `HOST/`-Platz im Image-Referenz-Format existiert genau dafür, dass Sie auf eine beliebige Registry zeigen können. Glaubwürdige Alternativen sind die GitHub Container Registry (`ghcr.io`, die Images direkt neben Ihrem Quellcode auf GitHub hält), Quay von Red Hat, die eigenen Registries der Cloud-Anbieter — Amazon ECR, Azure Container Registry, Google Artifact Registry — sowie eine vollständig selbst gehostete Option: Dockers eigene Open-Source-Registry (das `registry`-Image), mit der Sie eine private Registry komplett auf eigener Hardware betreiben. Zu jeder davon pushen Sie mit demselben `docker push`, das Sie gerade gelernt haben — nur mit ausgeschriebenem Registry-Host im Image-Namen, z. B. `ghcr.io/yourname/appdb:1.0`. Wählen Sie danach, wo Ihr Team und Ihre Compliance-Regeln die Images haben wollen.

Fazit

Die Magie aus Teil 2 war keine Magie — es waren Standardwerte. `postgres` expandierte zu `docker.io/library/postgres:latest`: ein echtes Image, im Repository `library/postgres`, auf der Registry Docker Hub, in der Version, die die Maintainer gerade mit `latest` etikettieren. Jetzt können Sie jede Image-Referenz auf Anhieb lesen, Versionen bewusst statt zufällig wählen, verstehen, warum wiederholte Pulls schnell sind — und sogar eigene Images veröffentlichen.

> Eine Image-Referenz ist `registry/repository:tag` — der Server, die benannte Sammlung von Versionen und die eine Version, die Sie wollen. Pinnen Sie den Tag, wenn es darauf ankommt, nutzen Sie geteilte Layer für Geschwindigkeit, und vertrauen Sie dem Badge, bevor Sie pullen.

Die eine Gewohnheit zum Mitnehmen: **Pinnen Sie Ihre Versionen.** `postgres:16` in allem, worauf Sie sich verlassen; `latest` nur für Wegwerf-Experimente. Reproduzierbarkeit ist eine Entscheidung, die Sie im Tag treffen.

Das bereitet Teil 4 perfekt vor. Eine Datenbank von Hand zu betreiben — sich Image, Tag, Ports und Umgebungsvariablen zu merken — wird mühsam, sobald Sie mehr als einen Container haben, und eine echte Anwendung hat das immer. Als Nächstes hören wir also auf, lange `docker run`-Zeilen zu tippen, und **beschreiben stattdessen den gesamten Stack in einer einzigen Datei**: Teil 4 — Docker Compose.

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 —