BE SOFTWARE ARCHITECTURE WISE · KAPITEL 0

Einstieg: Wie Software funktioniert

Dieses Kapitel ist der Einstieg in den Kurs. Es setzt voraus, dass Sie Apps und Websites genutzt, aber nie programmiert haben, und erklärt von Grund auf, was Software ist, wie sie Daten darstellt und verarbeitet, wie Code organisiert ist, wie Computer über Netzwerke kommunizieren, wo Daten gespeichert werden, wie getrennte Systeme über APIs zusammenarbeiten, was Server und die Cloud sind und wie Teams Software entwickeln und ausliefern. Erfahrene Fachleute können es als gemeinsames Vokabular überfliegen; alle anderen sollten es vor Kapitel 1 lesen, weil jedes spätere Kapitel auf diesen Ideen aufbaut.

bewiseapps.com/de/software-architecture-wise/foundations/ · Version 2026-10-26. Nur Bildungsinformation, keine professionelle Beratung.

AUF EINEN BLICK

Das lernen Sie

- Schritt für Schritt erklären, was geschieht, wenn ein Programm läuft, vom Quellcode über das Betriebssystem bis zum Prozessor.
- Die gängigen Datenstrukturen unterscheiden und mit der O-Notation vorhersagen, wie stark eine Operation mit wachsender Datenmenge langsamer wird.
- Beschreiben, wie Funktionen, Module, Objekte und Schnittstellen Code aufteilen, und den Unterschied zwischen einer Bibliothek und einem Framework erklären.
- Nachvollziehen, was geschieht, wenn jemand eine Webadresse eintippt, und dabei DNS, TCP, TLS und HTTP mit ihrem jeweiligen Beitrag benennen.
- Dateien, relationale Datenbanken und die wichtigsten NoSQL-Familien vergleichen und erklären, was eine Transaktion garantiert.
- Erklären, was eine API ist, wie ein REST-Aufruf über HTTP funktioniert und wann eine asynchrone Nachricht besser passt als ein synchroner Aufruf.
- Physische Server, virtuelle Maschinen, Container und die wichtigsten Cloud-Servicemodelle unterscheiden.
- Eine einzelne Nutzeraktion durch jede Schicht eines realen Systems verfolgen und die Entscheidungen erkennen, die seine Architektur ausmachen.

Bevor Sie beginnen

Keine. Es genügt, mit der Nutzung von Apps und Websites vertraut zu sein.

Dauer

Etwa 3 Stunden Lesezeit; 45 Minuten für den Test.

Schlüsselbegriffe

Prozess · Compiler und Interpreter · Datenstruktur · O-Notation (Big-O) · Schnittstelle · Framework · Abhängigkeit · IP-Adresse und Port · DNS · HTTP und HTTPS · Latenz · Datenbank und Transaktion · API · Container und Cloud

Warum hier beginnen

Bei Softwarearchitektur geht es um die großen Entscheidungen in einem Softwaresystem: was seine Hauptteile sind, wie sie miteinander kommunizieren, wo Daten liegen und welche Qualitäten, etwa Geschwindigkeit, Zuverlässigkeit und Sicherheit, der Entwurf erreichen muss. Diese Entscheidungen werden in einem bestimmten Vokabular getroffen: Prozesse und Threads, Indizes und Caches, Schnittstellen und Abhängigkeiten, HTTP und DNS, Transaktionen, APIs, Container und Pipelines. Architekten verwenden diese Wörter präzise, und vieles im Kurs setzt voraus, dass Sie wissen, was jedes davon wirklich bedeutet und was es kostet.

Dieses Kapitel liefert dieses Vokabular. Es ist für Menschen geschrieben, die nie programmiert haben, und jeder Abschnitt beginnt mit einer kurzen Analogie aus dem Alltag, bevor er den tatsächlichen Mechanismus erklärt. Zugleich soll es so genau sein, dass erfahrene Fachleute jeder Aussage zustimmen würden. Wo ein Thema viel mehr Tiefe verdient, führt das Kapitel es ein und verweist auf den späteren Abschnitt, der es gründlich behandelt.

Ein Programm auf einem Computer

Die ersten drei Abschnitte bleiben innerhalb eines einzelnen Rechners. Was Software ist erklärt den Unterschied zwischen Hardware und Software, wie von Menschen geschriebener Quellcode zu Anweisungen wird, die ein Prozessor ausführen kann (mithilfe von Compilern, Interpretern und virtuellen Maschinen), was das Betriebssystem tut und warum Daten im Arbeitsspeicher bei einem Neustart verschwinden, während Daten auf der Festplatte erhalten bleiben. Zum Schluss verfolgt er ein neunzeiliges Python-Programm durch jeden Schritt seines Lebens.

Daten und Algorithmen zeigt, wie Computer Zahlen und Text darstellen, und stellt die alltäglichen Datenstrukturen vor: Listen, Dictionaries, Sets, Queues, Stacks, Bäume und Graphen. Mithilfe der O-Notation erklärt er, warum ein Entwurf für die Suche nach einem Kunden auch bei einer Million Kunden schnell bleibt, während ein anderer zusammenbricht, und warum es Indizes und Caches gibt. Code organisieren steigt dann von Funktionen zu Modulen, Objekten und Schnittstellen auf, erklärt den Unterschied zwischen einer Bibliothek und einem Framework, warnt vor den versteckten Kosten von Abhängigkeiten und gibt einen ersten Blick auf Kopplung und Kohäsion, die zwei Ideen, die das gesamte Softwaredesign durchziehen.

Viele Computer arbeiten zusammen

Die nächsten drei Abschnitte verlassen den einzelnen Rechner. Wie Computer kommunizieren erklärt Pakete, IP-Adressen und Ports, TCP und UDP, wie DNS Namen in Adressen übersetzt und wie HTTP-Anfragen und -Antworten funktionieren, abgesichert durch TLS. Er macht außerdem einen Punkt deutlich, der viele überrascht: Die Latenz hat eine Untergrenze, die durch die Lichtgeschwindigkeit gesetzt ist, sodass ein Hin- und Rückweg zwischen London und Sydney nie schnell sein kann, ganz gleich, wie viel Bandbreite Sie kaufen.

Daten speichern behandelt Dateien und Datenbanken: Tabellen, Schlüssel und SQL-Abfragen, Indizes, Transaktionen und die ACID-Garantien (veranschaulicht an einer Banküberweisung, die nie nur halb stattfinden darf) sowie die wichtigsten NoSQL-Familien und wann welche passt. APIs und Integration erklärt, wie getrennte Systeme zusammenarbeiten: eine API als Vertrag,

REST und JSON über HTTP, Schlüssel und Tokens, Webhooks, der Unterschied zwischen dem Warten auf eine Antwort und dem Senden einer Nachricht über eine Queue und warum Breaking Changes an einer API allen schaden, die von ihr abhängen.

Wo Software läuft und wie sie entsteht

Server und die Cloud zeigt, was ein Server wirklich ist, wie virtuelle Maschinen und Container es vielen Workloads ermöglichen, sich einen physischen Rechner zu teilen, und was Cloud-Anbieter verkaufen: von der Miete nackter Maschinen bis dahin, Ihren Code nur dann auszuführen, wenn ein Ereignis eintrifft. Er führt Regionen, Availability Zones, Content Delivery Networks und die zwei Wege ein, mit Wachstum umzugehen: ein größerer Rechner oder mehr Rechner. Wie Software entsteht wendet sich von der Technik der Praxis zu: Anforderungen, Versionskontrolle mit Git, die Testpyramide, Umgebungen, Continuous Integration und Continuous Delivery, agile Arbeitsweisen und DevOps. Zum Schluss verfolgt er ein einziges Tippen auf „Bestellen“ in einer Essensliefer-App durch alle Teile des Kapitels, vom Smartphone über DNS, HTTPS, einen Load Balancer, Services, eine Datenbanktransaktion, einen Zahlungsanbieter und eine Queue bis zu einer Push-Benachrichtigung.

Die Idee zum Mitnehmen

Jedes System in diesem Kapitel besteht aus Teilen, die ihre Details hinter einer vereinbarten Grenze verbergen: das Betriebssystem hinter Systemaufrufen, ein Modul hinter seiner Schnittstelle, ein Webserver hinter HTTP, eine Datenbank hinter SQL, ein Zahlungsanbieter hinter seiner API, eine Cloud-Plattform hinter ihren Services. Grenzen ermöglichen es Menschen, große Systeme zu bauen, ohne jedes Detail zu verstehen, und an ihnen liegen die wichtigen Abwägungen: Ein Funktionsaufruf ist schnell und zuverlässig, ein Aufruf über ein Netzwerk dagegen langsamer und kann fehlschlagen; eine Abhängigkeit spart Aufwand, bringt aber Risiken mit sich; ein Cache spart Zeit, kann aber veraltete Daten liefern.

Zu entscheiden, wo diese Grenzen verlaufen, was sie überquert und welche Abwägungen man in Kauf nimmt, das ist Softwarearchitektur. Jeder Abschnitt endet mit einem kurzen Hinweis darauf, warum sein Thema für die Architektur wichtig ist, und Kapitel 1 beginnt genau dort, wo dieses Kapitel endet.

ABSCHNITT 0.1 VON 0.8 · EINSTEIGER · 15 MIN. LESEZEIT

Programme, Code und wie ein Computer sie ausführt

DIE IDEE IN KÜRZE

Ein Computer ist wie eine Restaurantküche. Der Koch (der Prozessor) kann nur geschriebenen Rezepten folgen, und zwar exakt, Wort für Wort. Die Arbeitsfläche (der Arbeitsspeicher) hält bereit, was der Koch gerade benutzt, und wird jeden Abend abgeräumt; die Vorratskammer (der Massenspeicher) bewahrt Dinge dauerhaft auf. Software ist die Sammlung der Rezepte. Menschen schreiben Rezepte in einer Sprache, die sie lesen können, und ein Übersetzer überträgt sie in die eigene, sehr einfache Sprache des Kochs. Die Küchenleitung (das Betriebssystem) entscheidet, welches Rezept die Aufmerksamkeit des Kochs bekommt, und sorgt dafür, dass sich die Stationen nicht gegenseitig in die Quere kommen.

WAS SIE WISSEN MÜSSEN

- Hardware ist die physische Maschine (CPU, RAM, Massenspeicher, Netzwerkkarte); Software ist die Gesamtheit der Anweisungen und Daten, die ihr sagen, was sie tun soll.
- Ein Programm ist eine Liste präziser Anweisungen; der Computer tut genau, was dort steht, daher werden Fehler in den Anweisungen zu Bugs.
- Programmierer schreiben Quellcode in Hochsprachen, den Compiler, Interpreter oder Bytecode-VMs in Maschinencode übersetzen, den die CPU ausführen kann.
- Das Betriebssystem teilt CPU, Arbeitsspeicher, Dateien und Netzwerk unter den Programmen auf und setzt die Sicherheit durch; Programme fordern seine Dienste über Systemaufrufe an.
- Ein laufendes Programm ist ein Prozess mit eigenem Speicher; Threads innerhalb eines Prozesses teilen sich diesen Speicher, was nützlich ist, aber Nebenläufigkeitsfehler verursachen kann.
- RAM ist schnell, verliert seinen Inhalt aber beim Neustart; alles, was nicht verloren gehen darf, muss in einen dauerhaften Speicher geschrieben werden, meist eine Datenbank.

Hardware und Software

Ein Computer ist eine Maschine, die Anweisungen sehr schnell befolgt. Die physischen Teile, die man anfassen könnte, sind die Hardware; die Anweisungen, die diesen Teilen sagen, was sie tun sollen, sind die Software. Ein Laptop, ein Smartphone, der Server, auf dem die Website Ihrer Bank läuft, und der Chip in einem smarten Thermostat sind allesamt Computer, gebaut aus denselben vier Arten von Teilen.

Die CPU (Central Processing Unit, also der Prozessor) führt Anweisungen aus. Ein moderner Prozessor hat mehrere *Kerne*, von denen jeder seinen eigenen Strom von Anweisungen abarbeiten kann, und jeder Kern führt Milliarden einfacher Schritte pro Sekunde aus. Der RAM (Random Access Memory, meist einfach *Arbeitsspeicher* genannt) enthält die Anweisungen und Daten, an denen die CPU gerade arbeitet. Er ist schnell, aber flüchtig: Schaltet man den Strom ab, ist sein Inhalt verschwunden. Der **Massenspeicher** (ein Solid-State-Laufwerk oder eine sich drehende Festplatte) behält Dateien auch ohne Strom, ist aber viel langsamer erreichbar. Eine **Netzwerkkarte** (kabelgebunden oder WLAN) sendet Daten an andere Computer und empfängt Daten von ihnen.

Denken Sie an eine Restaurantküche. Der Koch ist die CPU: der Einzige, der tatsächlich kocht. Die Arbeitsfläche ist der RAM: Alles, was der Koch in diesem Moment benutzt, liegt dort griffbereit, wird aber am Ende des Abends abgeräumt. Vorratskammer und Gefriertruhe sind der Massenspeicher: groß, dauerhaft und ein paar Schritte entfernt. Die Durchreiche zum Gastraum ist die Netzwerkkarte, durch die Bestellungen hereinkommen und Gerichte hinausgehen. Die Rezepte sind die Software.

Die vier Teile, die jeder Computer hat

Teil	Was er tut	Küchen-analogie	Was passiert, wenn er der Engpass ist
CPU (Prozessor)	Führt Anweisungen aus: Rechnen, Vergleichen, Daten verschieben	Der Koch	Arbeit staut sich; Seiten und Berichte werden langsam, obwohl Arbeitsspeicher frei ist
RAM (Arbeitsspeicher)	Enthält laufende Programme und die Daten, die sie nutzen	Die Arbeitsfläche	Das System beginnt, Daten auf die Festplatte auszulagern, und wird drastisch langsamer, oder Programme werden beendet
Massenspeicher (SSD oder Festplatte)	Bewahrt Dateien und Datenbanken dauerhaft auf	Vorratskammer und Gefriertruhe	Speichern und Laden von Daten wird langsam; eine volle Festplatte kann eine Datenbank stoppen
Netzwerkkarte	Sendet Daten an andere Rechner und empfängt Daten von ihnen	Die Durchreiche zum Gastraum	Anfragen warten darauf, gesendet oder empfangen zu werden; Nutzer bemerken Verzögerungen

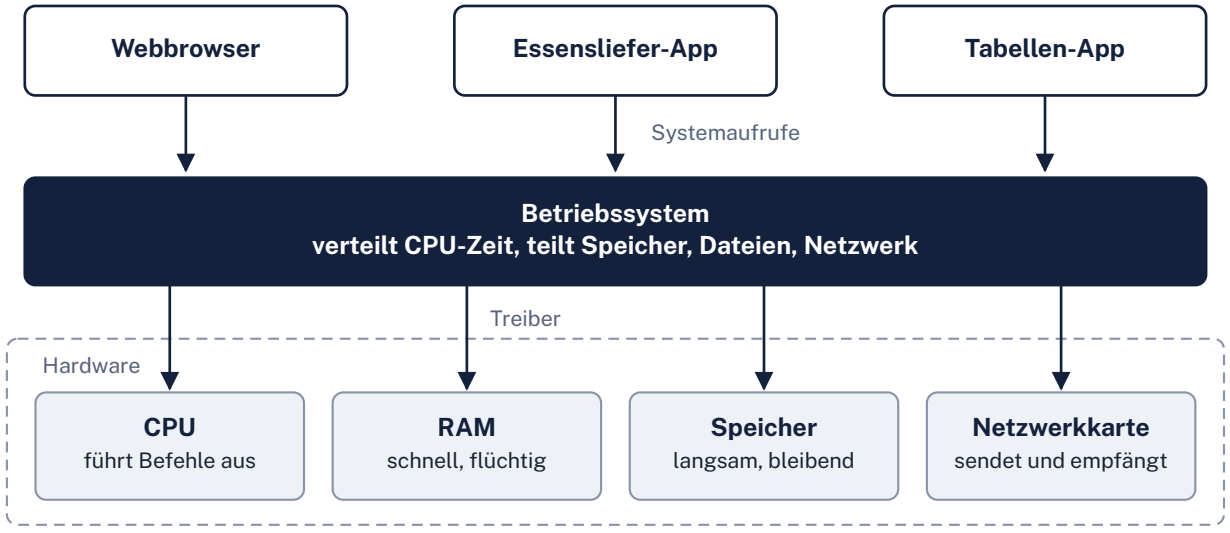
Was ein Programm ist

Ein Programm ist eine Liste präziser Anweisungen, denen ein Computer folgt, um eine Aufgabe zu erledigen: eine Rechnung berechnen, eine Webseite anzeigen, eine Zahlung senden. Das entscheidende Wort ist *präzise*. Ein menschlicher Koch kommt mit „nach Geschmack salzen“ zurecht; ein Computer nicht. Er tut genau das, was ihm gesagt wird, in genau der angegebenen Reihenfolge, und nichts darüber hinaus. Wenn eine Anweisung verlangt, durch eine Zahl zu teilen, die sich als null herausstellt, zuckt der Computer nicht mit den Schultern und macht weiter: Das Programm bricht mit einem Fehler ab, es sei denn, der Programmierer hat weitere Anweisungen geschrieben, was in diesem Fall zu tun ist.

Auf der untersten Ebene wiederholt die CPU einen einzigen Zyklus, solange sie eingeschaltet ist: die nächste Anweisung aus dem Speicher holen, entschlüsseln, was sie bedeutet, sie ausführen und zur nächsten übergehen. Jede Anweisung ist winzig: zwei Zahlen addieren, zwei Werte vergleichen, einen Wert von einer Stelle im Speicher an eine andere kopieren, zu einer anderen Anweisung springen, wenn ein Vergleich wahr war. Alles, was ein Computer tut, vom Abspielen eines Films bis zum Training eines KI-Modells, besteht aus Milliarden dieser kleinen Schritte.

Software ist der weitere Begriff: Programme zusammen mit den Daten, der Konfiguration und der Dokumentation, die sie brauchen. Eine *Anwendung* (oder App) ist Software, die für einen Menschen etwas Nützliches tut, etwa eine Banking-App. *Systemsoftware*, etwa das unten beschriebene Betriebssystem, existiert vor allem, um andere Software auszuführen.

Schichten: Anwendungen, das Betriebssystem und die Hardware



Anwendungen steuern die Hardware nie direkt: Sie wenden sich an das Betriebssystem, das die Hardware fair und sicher unter ihnen aufteilt.

Quellcode und Programmiersprachen

CPUs verstehen nur Maschinencode: als Zahlen codierte Anweisungen, die für eine Prozessorfamilie spezifisch sind. Intel- und AMD-Chips in den meisten Laptops und Servern verwenden eine Familie (x86-64); die Chips in Smartphones, neueren Macs und vielen Cloud-Servern eine andere (Arm). Maschinencode, der auf der einen läuft, läuft nicht auf der anderen. Maschinencode von Hand zu schreiben ist langsam, fehleranfällig und kommt heute fast nie vor.

Stattdessen schreiben Menschen Quellcode in einer **Programmiersprache**: einer genau definierten Notation, die sich teils wie Englisch und teils wie Mathematik liest. Python, JavaScript, Java, C#, Go, Rust und C sind weit verbreitete Beispiele. Man nennt sie *Hochsprachen*, weil Programmierer damit „addiere die Preise in diesem Warenkorb“ in ein oder zwei Zeilen statt in Dutzenden Maschinenbefehlen ausdrücken können und weil sie Details der jeweiligen CPU verbergen. Quellcode ist reiner Text, in Dateien gespeichert, und er ist das, was Programmierer lesen, schreiben, prüfen und in der Versionskontrolle ablegen (einem System, das eine Historie jeder Änderung führt; mehr dazu im letzten Abschnitt dieses Kapitels).

Sprachen unterscheiden sich darin, was sie leicht machen. Python wird für seine Lesbarkeit geschätzt und dominiert in Data Science und KI-Arbeit; JavaScript ist die Sprache, die jeder Webbrowser ausführt; Java und C# sind in großen Geschäftssystemen verbreitet; Go ist für Netzwerkdienste beliebt; C und Rust kommen dort zum Einsatz, wo jede Mikrosekunde und jedes Byte zählt, etwa in Betriebssystemen und Datenbanken. Keine Sprache ist in allem die beste, und die Wahl ist teils eine technische Entscheidung, teils eine Frage, welche Fähigkeiten ein Team mitbringt.

Vom Quellcode zum laufenden Programm: Compiler, Interpreter und virtuelle Maschinen

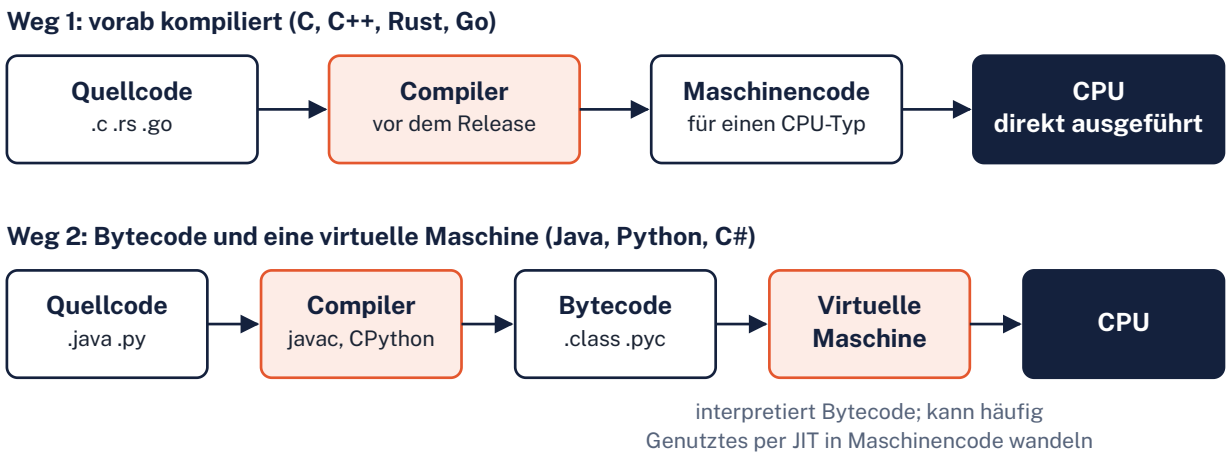
Quellcode muss in etwas umgewandelt werden, das die CPU ausführen kann. Es gibt drei grundlegende Ansätze, und reale Sprachen kombinieren sie oft.

Ein Compiler ist ein Programm, das den gesamten Quellcode eines Programms vor der Ausführung in eine andere Form übersetzt. Sprachen wie C, C++, Rust und Go werden meist *vorab* (ahead of time) in Maschinencode für eine bestimmte CPU und ein bestimmtes Betriebssystem kompiliert. Das Ergebnis ist eine *ausführbare* Datei, die schnell startet und schnell läuft, aber Sie müssen für jede Art von Rechner, die Sie unterstützen wollen, eine eigene Version kompilieren, und der Compiler weist das Programm zurück, wenn er bestimmte Arten von Fehlern findet.

Ein Interpreter ist ein Programm, das Quellcode (oder etwas sehr Ähnliches) liest und dessen Anweisungen gleich beim Lesen ausführt, Schritt für Schritt. Interpretation macht es leicht, Dinge auszuprobieren und dasselbe Programm auf jedem Rechner auszuführen, auf dem der Interpreter vorhanden ist, auf Kosten der Geschwindigkeit.

Viele populäre Sprachen liegen dazwischen. Java-Quellcode wird von einem Werkzeug namens `javac` in Bytecode kompiliert: kompakte Anweisungen für einen gedachten, idealisierten Computer statt für eine reale CPU. Die **Java Virtual Machine** (JVM) führt diesen Bytecode dann auf dem realen Rechner aus, auf dem sie installiert ist; so läuft ein Java-Programm unverändert unter Windows, macOS und Linux. Die JVM beobachtet, welche Teile des Programms am häufigsten laufen, und kompiliert diese Teile während der Laufzeit in Maschinencode, eine Technik namens *Just-in-time*-Kompilierung (JIT). C# auf .NET funktioniert ähnlich. Auch die Standardimplementierung von Python, CPython, kompiliert Ihre `.py`-Dateien zu Bytecode (wobei der Bytecode importierter Module in `.pyc`-Dateien zwischengespeichert wird) und führt diesen Bytecode dann in ihrer eigenen Interpreterschleife aus.

Zwei Wege vom Quellcode zur CPU



Vorab-Kompilierung erzeugt Maschinencode für eine Art von CPU; Kompilierung zu Bytecode lässt eine Datei überall laufen, wo eine passende virtuelle Maschine installiert ist.

ACHTUNG

Zwei Bedeutungen von „virtuelle Maschine“

Die hier beschriebenen virtuellen Maschinen von Sprachen (die JVM, die .NET-Laufzeitumgebung, der CPython-Interpreter) sind Programme, die Bytecode ausführen. Sie unterscheiden sich von den virtuellen Maschinen, die Cloud-Anbieter vermieten und die einen ganzen Computer nachbilden, damit sich ein physischer Server in mehrere aufteilen lässt. Beide heißen virtuelle Maschinen; die zweite Art wird in Server und die Cloud erklärt.

Die Ansätze im Vergleich

Ansatz	Typische Sprachen	Stärken	Kosten
Vorab-Kompilierung in Maschinencode	C, C++, Rust, Go	Schneller Start und schnelle Ausführung; eine eigenständige Datei zum Deployen	Eigene Version je CPU und Betriebssystem bauen; langsamerer Zyklus aus Ändern und Ausprobieren
Bytecode plus virtuelle Maschine mit JIT	Java, Kotlin, C#	Ein Build läuft auf vielen Plattformen; JIT kann lang laufende Services sehr schnell machen	Die Laufzeitumgebung muss installiert sein; langsamerer Start und zusätzlicher Speicher, während der JIT warmläuft
Bytecode plus Interpreter	Python (CPython), Ruby	Schnell geschrieben und ausprobiert; sehr portabel	Bei aufwendigen Berechnungen meist langsamer; geschwindigkeitskritische Teile werden oft in C geschrieben und aus Python aufgerufen
JIT-kompiliert im Browser oder auf dem Server	JavaScript (V8 in Chrome und Node.js)	Läuft überall, wo ein Browser läuft; schnell genug für die meisten Webaufgaben	Die Performance kann weniger vorhersehbar sein als bei kompiliertem Code

Die Aufgabe des Betriebssystems

Wenn Sie einen Computer einschalten, ist das erste größere Programm, das startet, das Betriebssystem (Operating System, OS): Windows, macOS, Linux, Android oder iOS. Die meisten Server im Internet laufen unter Linux. Das Betriebssystem hat vier Hauptaufgaben.

Erstens **teilt es die CPU auf**. Auf einem Laptop laufen vielleicht ein paar Dutzend Programme, aber es gibt nur eine Handvoll Kerne, also schaltet das OS jeden Kern viele Male pro Sekunde zwischen Programmen um, so schnell, dass sie alle gleichzeitig zu laufen scheinen. Zweitens **verwaltet es den Arbeitsspeicher**: Es gibt jedem Programm einen eigenen Bereich und hindert ein Programm daran, den Bereich eines anderen zu lesen oder zu überschreiben. Drittens stellt es **Dateien und Geräte** bereit: Programme bitten das OS, eine Datei zu öffnen, etwas auf dem Bildschirm darzustellen oder Daten über das Netzwerk zu senden, und das OS kümmert sich über kleine Programme namens *Treiber* um die jeweilige Hardware. Viertens **setzt es die Sicherheit durch**: Benutzer, Berechtigungen und die Regel, dass eine App Ihre Fotos nicht lesen darf, solange Sie es nicht erlauben.

Programme stellen diese Anfragen über *Systemaufrufe*, eine feste Menge von Diensten, die das OS anbietet. Das ist das erste Beispiel in diesem Kurs für eine wichtige Architekturidee: eine Schicht, die unübersichtliche Details hinter einer stabilen Menge von Anfragen verbirgt. Wer eine Essensliefer-App schreibt, muss nicht wissen, welche Marke von Netzwerkchip das Smartphone hat.

Prozesse, Threads, Arbeitsspeicher und Festplatte

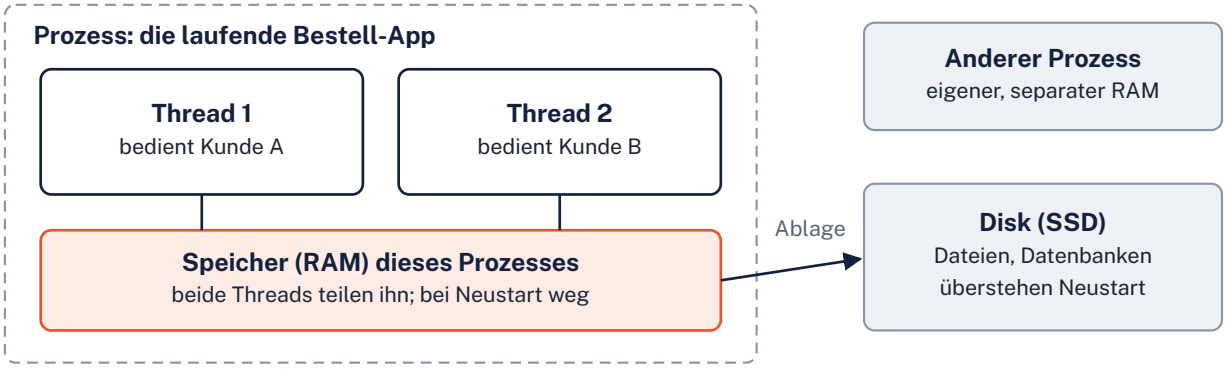
Eine Programmdatei auf der Festplatte ist nur eine Sammlung gespeicherter Anweisungen. Wenn Sie sie starten, erzeugt das OS einen Prozess: eine laufende Instanz des Programms mit eigenem privatem Speicher, eigener Identität und eigenem Anteil an CPU-Zeit. Öffnen Sie zwei Browserfenster, haben Sie möglicherweise mehrere Prozesse; starten Sie einen Webserver, läuft er vielleicht als ein Prozess oder als viele.

Innerhalb eines Prozesses kann es mehrere Threads geben. Ein Thread ist eine Abfolge von Anweisungen, die gerade ausgeführt wird. Ein Prozess mit mehreren Threads kann mehrere Dinge gleichzeitig tun, etwa die Anfragen zweier Kunden zur selben Zeit bearbeiten, und alle seine Threads teilen sich den Speicher des Prozesses. Diese gemeinsame Nutzung ist bequem, aber riskant: Wenn zwei Threads dasselbe Datenelement im selben Moment ohne Abstimmung ändern, kann das Ergebnis auf eine Weise falsch sein, die nur gelegentlich auftritt. Solche *Nebenläufigkeitsfehler* gehören zu den am schwersten zu findenden, und das ist ein Grund, warum viele Systeme getrennte Prozesse bevorzugen oder Entwürfe, in denen jede Anfrage unabhängig bearbeitet wird.

Der Unterschied zwischen Arbeitsspeicher und Festplatte erklärt einen großen Teil des Softwaredesigns. Daten im RAM sind schnell erreichbar, in etwa einer Zehntelmikrosekunde, verschwinden aber, wenn der Prozess endet oder abstürzt oder der Rechner neu startet. Daten auf der Festplatte bleiben erhalten, aber der Zugriff dauert viel länger: Dutzende Mikrosekunden oder mehr auf einem Solid-State-Laufwerk und mehrere Millisekunden auf

einer sich drehenden Festplatte. Deshalb muss jede Information, die nicht verloren gehen darf, etwa eine Bestellung oder eine Zahlung, in einen dauerhaften Speicher geschrieben werden, meist über eine Datenbank, bevor das System dem Kunden „erledigt“ meldet.

Ein Prozess, seine Threads, sein Speicher und die Festplatte



Threads in einem Prozess teilen sich seinen Speicher, getrennte Prozesse nicht; nur was auf die Festplatte gespeichert wird, übersteht einen Neustart.

```
PRICE_PER_ITEM = 12.50   # pounds
VAT_RATE = 0.20          # 20 per cent

def total_with_vat(quantity):
    subtotal = quantity * PRICE_PER_ITEM
    return subtotal * (1 + VAT_RATE)

quantity = int(input("How many items? "))
print("Total to pay:", round(total_with_vat(quantity), 2))
```

Ein winziges Python-Programm: den Preis einer Bestellung einschließlich Mehrwertsteuer berechnen

AUSGEARBEITETES BEISPIEL

Ausgearbeitetes Beispiel: was geschieht, wenn das Programm läuft

Angenommen, der obige Code ist als `order.py` gespeichert, Sie tippen `python order.py` ein und geben dann 3 ein.

1. Start	Das Betriebssystem erzeugt einen neuen Prozess, gibt ihm Speicher und startet darin den Python-Interpreter (selbst ein kompiliertes Programm).
2. Kompilieren	CPython liest `order.py` von der Festplatte, prüft, ob es gültiges Python ist, und kompiliert es zu Bytecode. Eine fehlende Klammer oder ein fehlender Doppelpunkt würde hier als *Syntaxfehler* gemeldet, bevor irgendetwas läuft.
3. Definieren	Der Interpreter führt den Bytecode von oben aus. Er legt 12,5 und 0,2 unter den Namen `PRICE_PER_ITEM` und `VAT_RATE` im Speicher ab und registriert die Funktion `total_with_vat`, ohne sie schon auszuführen.
4. Eingabe	`input()` bittet das OS, von der Tastatur zu lesen. Der Prozess wartet, ohne CPU zu verbrauchen, bis Sie 3 eintippen und die Eingabetaste drücken. `int()` wandelt den Text „3“ in die Zahl 3 um.
5. Berechnen	Die Funktion läuft: 3 mal 12,5 ergibt 37,5, dann 37,5 mal 1,2 ergibt 45,0. Jeder Schritt besteht aus einer Handvoll Bytecode-Anweisungen, von denen jede auf der CPU zu vielen Maschinenbefehlen wird.
6. Ausgabe	`print()` bittet das OS, `Total to pay: 45.0` auf dem Bildschirm anzuzeigen.
7. Ende	Das Programm erreicht seine letzte Zeile, der Prozess endet, und das OS fordert seinen Speicher zurück. Nichts wurde auf die Festplatte gespeichert, also ist das Ergebnis weg, sofern Sie es nicht abgeschrieben haben.

Selbst dieses neunzeilige Programm ist für Speicher, Tastatur und Bildschirm auf das Betriebssystem angewiesen und auf eine Laufzeitumgebung der Sprache, die es für die CPU übersetzt.

Hätten Sie `three` statt 3 eingetippt, würde `int()` mit einem *Laufzeitfehler* scheitern und das Programm abbrechen: Der Code hat nie festgelegt, was mit ungültigen Eingaben geschehen soll.

Das Programm speichert Geld der Kürze halber als Gleitkommazahl. Reale Finanzsysteme speichern ganze Pence oder verwenden einen Dezimaltyp, aus Gründen, die der nächste Abschnitt erklärt.

Bugs: wenn die Anweisungen falsch sind

Ein Bug ist ein Fehler in Software, der dazu führt, dass sie sich anders verhält als beabsichtigt. Weil Computer Anweisungen exakt befolgen, ist fast jeder Bug eine Lücke zwischen dem, was der Programmierer gemeint, und dem, was er tatsächlich geschrieben hat, oder eine Situation, an die niemand gedacht hat.

Es hilft, drei Arten zu unterscheiden. Ein **Syntaxfehler** verletzt die Grammatik der Sprache (eine fehlende Klammer), sodass der Compiler oder Interpreter das Programm gar nicht erst ausführt; solche Fehler fallen sofort auf. Ein **Laufzeitfehler** tritt auf, während das Programm läuft, etwa bei einer Division durch null oder beim Versuch, eine nicht vorhandene Datei zu lesen; das Programm bricht dann meist mit einer Fehlermeldung ab. Ein **Logikfehler** ist der gefährlichste: Das Programm läuft fröhlich weiter und liefert die falsche Antwort, berechnet zum Beispiel die Mehrwertsteuer doppelt. Nichts stürzt ab, daher decken ihn nur Tests, sorgfältige Reviews oder ein unzufriedener Kunde auf.

Professionelle Teams rechnen mit Bugs und bauen Schutzmechanismen ein: automatisierte Tests, Code-Reviews, schrittweise Releases und ein Monitoring, das bemerkt, wenn die Fehlerraten steigen. Viele dieser Schutzmechanismen sind architektonischer Natur, und sie tauchen im gesamten Kurs auf.

WOHER DAS KOMMT

Die Motte in der Maschine

Ingenieure nannten Störungen schon lange vor den Computern „Bugs“; Thomas Edison verwendete das Wort in den 1870er-Jahren. Die bekannteste Geschichte aus der Informatik stammt vom 9. September 1947, als das Bedienpersonal des Computers Harvard Mark II eine Motte fand, die in einem seiner Relais gefangen war, sie ins Logbuch klebte und „First actual case of bug being found“ dazuschrieb. Der Witz funktioniert gerade deshalb, weil der Begriff bereits gebräuchlich war. Grace Hopper, die an der Maschine arbeitete und später frühe Programmiersprachen prägte, machte die Geschichte bekannt.

VERTIEFUNG

Vertiefung: Sie können den Bytecode sehen

Die Standardbibliothek von Python enthält ein Modul namens `dis`, das den Bytecode jeder Funktion anzeigt. Wenn Sie `dis.dis(total_with_vat)` auf das obige Beispiel anwenden, erhalten Sie Anweisungen wie `LOAD_FAST` (eine lokale Variable holen), `LOAD_GLOBAL` (einen Namen auf Modulebene holen), `BINARY_OP` (multiplizieren oder addieren) und `RETURN_VALUE`. Die genauen Anweisungen ändern sich zwischen Python-Versionen, weshalb `.pyc`-Dateien an die Version gebunden sind, die sie erzeugt hat.

Warum das für die Architektur wichtig ist

Bei Architektur geht es um die großen Entscheidungen, die teuer zu ändern sind, und mehrere davon beruhen auf den Ideen dieses Abschnitts. Die Wahl von Sprache und Laufzeitumgebung beeinflusst Startzeit, Speicherverbrauch, welche Bibliotheken Sie nutzen können und wen Sie einstellen können. Die Trennung zwischen Arbeitsspeicher und Festplatte entscheidet, was dauerhaft gespeichert werden muss und wie sich ein System nach einem Absturz erholt. Prozesse und Threads sind die Einheiten, die laufen, ausfallen und skalieren: Ein späteres Kapitel fragt, ob ein System ein einziger Prozess oder viele zusammenarbeitende sein sollte. Und der geschichtete Aufbau des Betriebssystems, der die Hardware hinter einer stabilen Menge von Anfragen verbirgt, ist das erste Beispiel für ein Prinzip, dem Sie als Information Hiding wiederbegegnen werden.

Der nächste Abschnitt betrachtet, womit Programme eigentlich arbeiten: Daten und die Strukturen und Algorithmen, die die Arbeit damit schnell oder langsam machen.

PRÜFEN SIE IHR VERSTÄNDNIS

▼ Warum verschwinden die Daten eines Programms, wenn es abstürzt, sofern sie nicht gespeichert wurden?

Während ein Programm läuft, liegen seine Daten im RAM, der zum Prozess gehört und geleert wird, wenn der Prozess endet oder der Rechner neu startet. Nur Daten, die in einen Speicher wie eine Festplatte oder Datenbank geschrieben wurden, bleiben erhalten.

▼ Was ist der Unterschied zwischen einem Compiler und einem Interpreter?

Ein Compiler übersetzt das gesamte Programm vor der Ausführung in eine andere Form (Maschinencode oder Bytecode); ein Interpreter führt die Anweisungen aus, während er sie liest. Viele Sprachen, etwa Java und Python, kompilieren zuerst zu Bytecode und führen diesen Bytecode dann in einer virtuellen Maschine aus.

▼ Warum kann ein auf einem Windows-Laptop gebautes Java-Programm meist unverändert auf einem Linux-Server laufen?

Java wird zu Bytecode kompiliert, nicht zu Maschinencode für eine bestimmte CPU. Jeder Rechner, auf dem eine Java Virtual Machine installiert ist, kann diesen Bytecode ausführen.

▼ Warum bitten Programme das Betriebssystem, eine Datei zu lesen oder Daten zu senden, statt die Festplatte oder Netzwerkkarte selbst zu steuern?

Das Betriebssystem teilt die Hardware sicher unter den Programmen auf, setzt Berechtigungen durch und verbirgt die Details jedes Geräts hinter Systemaufrufen, sodass Anwendungen nicht wissen müssen, auf welcher Hardware sie laufen.

QUELLEN

- The Python Tutorial (Python documentation): https://docs.python.org/3/tutorial/index.html
- Python glossary: bytecode, interpreter and related terms: https://docs.python.org/3/glossary.html
- dis: disassembler for Python bytecode: https://docs.python.org/3/library/dis.html
- The Java Virtual Machine Specification, Java SE 21 Edition: https://docs.oracle.com/javase/specs/jvms/se21/html/index.html

ABSCHNITT 0.2 VON 0.8 · EINSTEIGER · 16 MIN. LESEZEIT

Daten, Datenstrukturen und Algorithmen und warum die Geschwindigkeit von ihnen abhängt

DIE IDEE IN KÜRZE

Stellen Sie sich vor, Sie suchen in einem Postamt nach einem bestimmten Brief. Liegen die Briefe auf einem Haufen, müssen Sie sie einzeln durchsehen, und ein größerer Haufen dauert länger. Sind sie nach Postleitzahl in Fächer sortiert, gehen Sie direkt zum richtigen Fach, ganz gleich, wie viele Briefe es sind. Datenstrukturen sind die Arten, Daten anzuordnen (Haufen, Fächer, Warteschlangen, Stapel), und Algorithmen sind die Methoden, mit ihnen zu arbeiten. Wer sie gut wählt, entscheidet darüber, ob ein System beim Wachsen schnell bleibt.

WAS SIE WISSEN MÜSSEN

- Computer speichern alles als Bits; ein Byte besteht aus acht Bits, und Datentypen (Ganzzahlen, Gleitkommazahlen, Boolesche Werte, Strings) legen fest, wie sie zu lesen sind.
- Gleitkommazahlen sind nur Näherungen, daher sollte Geld als Ganzzahl der kleinsten Einheit oder mit einem Dezimaltyp gespeichert werden.
- Text wird als Unicode-Zeichen gespeichert, die als Bytes codiert sind, meist in UTF-8, wo ein Zeichen ein bis vier Bytes belegen kann.
- Datenstrukturen wie Listen, Dictionaries, Sets, Queues, Stacks, Bäume und Graphen machen jeweils andere Operationen schnell.
- Die O-Notation beschreibt, wie der Aufwand eines Algorithmus mit den Daten wächst: O(1), O(log n), O(n), O(n log n), $O(n^2)$.
- Indizes und Caches tauschen zusätzlichen Platz (und den Aufwand, sie aktuell zu halten) gegen schnellere Antworten.

Alles besteht aus Bits

Computer speichern alles als Bits: Binärziffern, die entweder 0 oder 1 sind. Physisch kann ein Bit eine winzige elektrische Ladung sein, ein magnetischer Bereich auf einer Festplatte oder ein Lichtimpuls in einer Glasfaser, aber für Software ist es einfach einer von zwei Werten. Acht Bits ergeben ein Byte, das $2^8 = 256$ verschiedene Muster aufnehmen kann, von 00000000 bis 11111111.

Die Größen bauen darauf auf. Ein Kilobyte sind etwa tausend Bytes, ein Megabyte etwa eine Million, ein Gigabyte etwa eine Milliarde und ein Terabyte etwa eine Billion. (Streng genommen bedeuten die Standardpräfixe Kilo, Mega und Giga Potenzen von 1000; Speicher wird oft in Potenzen von 1024 gemessen, wofür es die Einheiten Kibibyte, Mebibyte und Gibibyte gibt. Der Unterschied spielt für den Entwurf selten eine Rolle, erklärt aber, warum eine „1 TB“-Festplatte in manchen Dateimanagern weniger anzeigt.) Eine Seite reiner Text umfasst ein paar Kilobyte, ein Handyfoto ein paar Megabyte, eine Stunde HD-Video ein paar Gigabyte.

Für sich genommen bedeuten Bits nichts. Das Muster 01000001 könnte die Zahl 65 sein, der Buchstabe A, Teil einer Farbe in einem Foto oder Teil einer Anweisung für die CPU. Bedeutung erhalten Bits erst durch die Vereinbarung, wie sie zu lesen sind, und genau das ist die Idee eines Datentyps.

Datentypen: Zahlen, Text und wahr oder falsch

Ein Datentyp sagt einem Programm, welche Art von Wert ein Datenelement ist und welche Operationen daher sinnvoll sind. Jede Programmiersprache hat einige Grundtypen, und die von Python sind typisch.

Ganzzahlen (Integers) sind ganze Zahlen wie 3, 0 oder -42 und dienen zum Zählen: Artikel in einem Warenkorb, Sitze in einem Flug. **Gleitkommazahlen** (Floats) wie 12,5 oder 0,2 stellen Dezimalzahlen näherungsweise im Binärsystem dar. Weil viele Dezimalzahlen keine exakte binäre Form haben, ergibt `0.1 + 0.2` in Python `0.30000000000000004`. Für Physik und Grafik ist das in Ordnung, für Geld aber falsch, weshalb Finanzsysteme Beträge als ganze Zahlen in Pence oder Cent speichern oder einen Dezimaltyp verwenden (Python hat `decimal.Decimal`). **Boolesche Werte** (Booleans) haben nur zwei Werte, `True` und `False`, und halten Ja-Nein-Fakten fest wie „Ist diese Bestellung bezahlt?“. **Strings** sind Text, etwa der Name eines Kunden. Reale Systeme fügen weitere Typen hinzu: Datum und Uhrzeit, Geldbeträge und Binärdaten wie Bilder.

Typen sind über ein einzelnes Programm hinaus wichtig. Wenn zwei Systeme Daten austauschen, müssen sie sich auf Typen ebenso einigen wie auf Werte: Ist eine Postleitzahl eine Zahl oder Text (Text: Manche beginnen mit Buchstaben, und manche numerischen Codes beginnen mit einer Null)? Steht ein Datum in der Reihenfolge Tag-Monat oder Monat-Tag, und in welcher Zeitzone? Viele teure Integrationsfehler sind Unstimmigkeiten bei Typen.

Text und UTF-8

Text wird gespeichert, indem jedes Zeichen eine Nummer erhält und diese Nummern dann als Bytes abgelegt werden. Der internationale Standard **Unicode** weist weit über hunderttausend Zeichen aus den Schriftsystemen der Welt, dazu Symbolen und Emojis, jeweils eine Nummer

zu, einen sogenannten Codepunkt. Eine *Codierung* legt dann fest, wie diese Nummern zu Bytes werden. Die heute vorherrschende Codierung, im Web und in den meisten neuen Systemen, ist UTF-8.

UTF-8 verwendet eine variable Anzahl von Bytes pro Zeichen: ein Byte für die englischen Grundbuchstaben, Ziffern und Satzzeichen (sodass Text, der nur diese Zeichen verwendet, in UTF-8 Byte für Byte identisch mit der viel älteren ASCII-Codierung ist), zwei Bytes für Zeichen wie `é`, drei für Zeichen wie `€` und die meisten chinesischen, japanischen und koreanischen Zeichen und vier für Emojis und einige seltenere Schriften. Das hat eine praktische Folge: Die Anzahl der Zeichen in einem String und die Anzahl der Bytes, die er belegt, können sich unterscheiden. Eine Datenbankspalte, die auf 50 Bytes begrenzt ist, fasst einen 50 Zeichen langen Namen in griechischer Schrift möglicherweise nicht.

Der klassische Textfehler sind verstümmelte Zeichen (zum Beispiel `CafÃ©` statt `Café`), die entstehen, wenn Bytes, die in einer Codierung geschrieben wurden, in einer anderen gelesen werden. Die Abhilfe ist einfach zu formulieren: Verwenden Sie überall UTF-8 und geben Sie dies ausdrücklich an, wann immer Text eine Grenze überquert, etwa eine Datei, eine Datenbank oder eine Netzwerknachricht.

Datenstrukturen: Arten, Daten anzuordnen

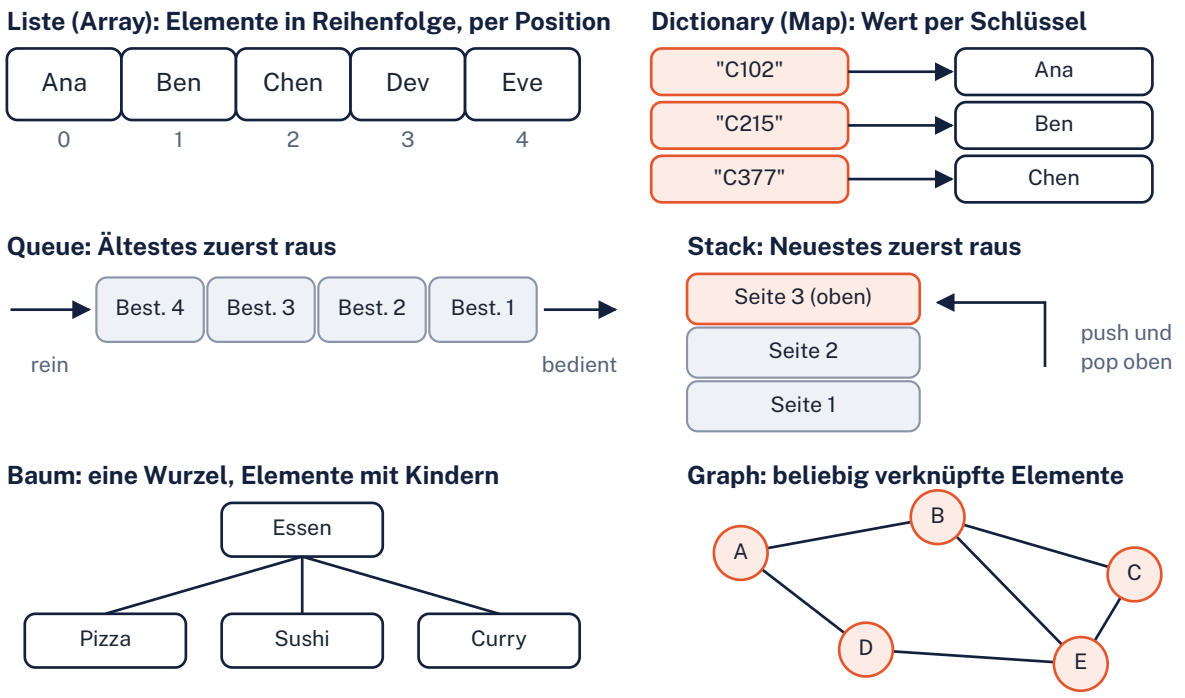
Programme haben selten mit nur einem Wert zu tun. Eine Datenstruktur ist eine Art, viele Werte im Speicher so zu organisieren, dass die benötigten Operationen schnell sind. Die richtige Wahl kann den Unterschied ausmachen zwischen einer Seite, die sofort lädt, und einer, die nie lädt. Für jede gibt es eine Analogie aus dem Alltag.

Eine **Liste** (in vielen Sprachen *Array* genannt) hält Elemente in einer Reihenfolge, wie eine nummerierte Einkaufsliste. Das fünfte Element abzurufen geht sofort, weil der Computer berechnen kann, wo es liegt. Ein Element anhand seines Werts zu finden, bedeutet dagegen, die Elemente einzeln zu prüfen. Ein Dictionary (auch *Map* oder *Hashtabelle* genannt) speichert Werte unter eindeutigen Schlüsseln, wie eine Garderobe: Geben Sie die Marke C377 ab, und Sie bekommen sofort den richtigen Mantel, ohne dass jemand alle Mäntel durchsehen muss. Ein **Set** ist eine Sammlung ohne Duplikate, wie eine Gästeliste, und beantwortet die Frage „Steht diese Person darauf?“ schnell.

Eine **Queue** (Warteschlange) funktioniert wie die Schlange am Postschalter: Wer zuerst kommt, wird zuerst bedient. Systeme nutzen Queues für Arbeit, die darauf wartet, erledigt zu werden, etwa Bestellungen, die auf die Küche warten. Ein **Stack** (Stapel) funktioniert wie ein Tellerstapel: Was zuletzt kommt, geht zuerst. Die Zurück-Schaltfläche Ihres Browsers verwendet einen: Die zuletzt besuchte Seite ist die erste, zu der Sie zurückkehren. Ein **Baum** hat eine einzige Wurzel mit Zweigen darunter, wie ein Stammbaum oder die Ordner auf Ihrem

Computer; Bäume sind auch das Mittel, mit dem Datenbanken Daten für eine schnelle Suche sortiert halten. Ein **Graph** besteht aus Elementen (Knoten), die durch Verbindungen (Kanten) in beliebigem Muster verknüpft sind, wie eine Straßenkarte oder ein soziales Netzwerk, und beantwortet Fragen wie „Was ist der kürzeste Weg?“ oder „Wer sind Freunde von Freunden?“.

Sechs gängige Datenstrukturen



Jede Struktur ordnet dieselbe Art von Daten anders an, wodurch jeweils andere Fragen schnell beantwortet werden.

Wofür jede Struktur gut ist

Struktur	Schnell bei	Langsam bei	Typische Verwendung
Liste / Array	Element an einer Position abrufen; am Ende anfügen	Element anhand des Werts finden (muss durchsuchen); vorne einfügen	Ein geordneter Warenkorb mit Artikeln; eine Seite mit Suchergebnissen
Dictionary / Map	Finden, Hinzufügen oder Entfernen per Schlüssel	Ausgabe der Elemente in sortierter Reihenfolge	Kunden nach ID; Einstellungen nach Name; ein Cache
Set	Zugehörigkeit prüfen; Duplikate entfernen	Reihenfolge beibehalten (die meisten Sets haben keine)	Welche Nutzer bereits abgestimmt haben; eindeutige Tags
Queue	Hinten anfügen und vorne entnehmen	Zugriff auf Elemente in der Mitte	Bestellungen, die auf die Küche warten; Jobs, die auf einen Worker warten
Stack	Oben ablegen und oben entnehmen	Zugriff auf weiter unten liegende Elemente	Rückgängig-Verlauf; die Zurück-Schaltfläche; Verfolgen von Funktionsaufrufen
Baum (sortiert)	Suchen nach Wert und Auflisten eines Bereichs in Reihenfolge	Aufwendiger im Gleichgewicht zu halten	Datenbankindizes; Ordner; Menüs
Graph	Beziehungen verfolgen	Sehr große Graphen brauchen spezialisierte Speicherung	Routen und Karten; soziale Netzwerke; Empfehlungen

```
basket = ["pizza", "cola", "pizza"]           # list: ordered, duplicates allowed
print(basket[0])                               # by position: "pizza"

customers = {"C102": "Ana", "C377": "Chen"}    # dictionary: key to value
print(customers["C377"])                       # by key: "Chen"

voted = {"ana@example.com", "ben@example.com"}  # set: no duplicates
print("ana@example.com" in voted)              # True

from collections import deque
kitchen = deque(["order 1", "order 2"])        # queue
kitchen.append("order 3")                      # joins at the back
print(kitchen.popleft())                       # served first: "order 1"
```

Dieselben Strukturen in Python

Algorithmen

Ein Algorithmus ist eine präzise, schrittweise Methode zur Lösung eines Problems, die immer zu einem Ende kommt: wie ein Rezept, aber ohne Interpretationsspielraum. Eine Namensliste sortieren, die kürzeste Route zwischen zwei Adressen finden, prüfen, ob ein Passwort übereinstimmt, und entscheiden, welche Werbung angezeigt wird: All das erledigen Algorithmen.

Für dasselbe Problem gibt es meist mehr als einen Algorithmus, und sie können sich enorm darin unterscheiden, wie viel Arbeit sie leisten. Denken Sie an die Suche nach einem Namen in einem gedruckten Telefonbuch. Sie könnten auf Seite eins beginnen und jeden Eintrag lesen, bis Sie ihn finden: korrekt, aber langsam. Oder Sie schlagen das Buch in der Mitte auf, prüfen, ob der Name vor oder nach dieser Seite kommt, verwerfen die Hälfte, die ihn nicht enthalten kann, und wiederholen das. Weil das Telefonbuch sortiert ist, halbiert jeder Blick die verbleibenden Seiten. Diese zweite Methode heißt **binäre Suche**, und bei einem Telefonbuch mit einer Million Namen braucht sie etwa 20 Blicke statt bis zu einer Million.

Binäre Suche: das Problem halbieren

Hervorgehoben: das je Schritt verglichene mittlere Element. Grau: ausgeschlossen. Übrig: 16, 7, 3, 1

In einer sortierten Liste schließt jeder Vergleich die Hälfte des Rests aus, sodass 16 Elemente höchstens 4 oder 5 Blicke brauchen.

O-Notation: wie der Aufwand mit den Daten wächst

Fachleute beschreiben, wie die Kosten eines Algorithmus wachsen, mit der O-Notation (Big-O). Sie sieht von der genauen Zeit auf einem bestimmten Rechner ab und stellt nur eine Frage: Wenn die Datenmenge, genannt *n*, zehn- oder tausendmal größer wird, wie viel mehr Arbeit fällt dann an? Sie beschreibt den ungünstigsten Fall, sofern nichts anderes gesagt wird, und ignoriert konstante Faktoren, was zugleich ihre Stärke ist (sie erfasst die Form des Wachstums) und ihre Schwäche (zwei O(n)-Algorithmen können sich in der Praxis trotzdem um den Faktor zehn unterscheiden).

O(1), konstante Zeit: Der Aufwand hängt nicht von n ab. Einen Schlüssel in einem Dictionary nachzuschlagen ist im Durchschnitt O(1). **O(log n)**, logarithmisch: Der Aufwand wächst um einen Schritt, wann immer sich n verdoppelt, wie bei der binären Suche. **O(n)**, linear: doppelte Daten, doppelter Aufwand, wie beim Durchsuchen einer Liste. **O(n log n)**: etwas schlechter als linear; gute Allzweck-Sortieralgorithmen arbeiten in dieser Größenordnung. **O(n^2)**, quadratisch: doppelte Daten, vierfacher Aufwand. Das tritt typischerweise auf, wenn ein Programm für jedes Element über alle anderen Elemente läuft, etwa wenn es jeden Kunden mit jedem anderen Kunden vergleicht, um Duplikate zu finden.

Wie viele Grundschritte ungefähr anfallen, wenn die Daten wachsen

n (Elemente)	O(1)	O(log n)	O(n)	O(n log n)	O(n^2)
10	1	etwa 3	10	etwa 33	100
1,000	1	etwa 10	1,000	etwa 10.000	1,000,000
1,000,000	1	etwa 20	1,000,000	etwa 20.000.000	1,000,000,000,000

Lesen Sie die letzte Spalte genau. Bei einer Milliarde einfacher Schritte pro Sekunde braucht ein Algorithmus mit einer Million zum Quadrat eine Billion Schritte: etwa eine Viertelstunde reiner Rechenzeit und in der Praxis weit länger. Dieselbe Aufgabe mit einer O(n log n)-Methode dauert einen kleinen Bruchteil einer Sekunde. Code, der im Test mit 1000 Kunden gut aussah, kann bei einer Million zum Stillstand kommen, und keine noch so große Menge zusätzlicher Hardware rettet einen quadratischen Algorithmus auf Dauer.

AUSGEARBEITETES BEISPIEL

Ausgearbeitetes Beispiel: einen Kunden unter einer Million finden

Eine Essensliefer-App speichert 1.000.000 Kunden. Jedes Mal, wenn eine Bestellung eingeht, muss sie den Kunden anhand seiner ID finden. Vergleichen Sie zwei Entwürfe und nehmen Sie an, dass jede Prüfung 0,1 Mikrosekunden dauert.

Entwurf A: eine Liste	`for c in customers: if c.id == wanted: ...` durchsucht die Liste, bis ein Treffer gefunden ist.
Prüfungen pro Suche (A)	Bis zu 1.000.000; etwa 500.000 im Durchschnitt.
Zeit pro Suche (A)	Etwa 500.000 x 0,1 Mikrosekunden = 50 Millisekunden.
Bei 200 Bestellungen pro Sekunde (A)	200 x 50 ms = 10 Sekunden CPU-Arbeit pro Sekunde: Der Server kommt nicht hinterher, sodass sich Anfragen stauen und in einen Timeout laufen.
Entwurf B: ein Dictionary mit der ID als Schlüssel	`customers_by_id[wanted]` berechnet, wo der Eintrag liegt, und geht direkt dorthin.
Prüfungen pro Suche (B)	Etwa 1, ob es tausend Kunden sind oder eine Million.
Zeit pro Suche (B)	Deutlich unter einer Mikrosekunde.
Kosten von Entwurf B	Zusätzlicher Speicher für das Dictionary sowie der Aufwand, es aufzubauen und aktuell zu halten, wann immer ein Kunde hinzukommt oder sich ändert.

Die beiden Entwürfe liefern identische Antworten; nur die Datenstruktur unterscheidet sich, und das macht aus einem Scheitern komfortable Reserven.

Datenbanken bieten dieselbe Wahl. Ohne Index bedeutet das Finden einer Zeile anhand einer Spalte, die Tabelle zu durchsuchen; mit Index sind es wenige Schritte durch einen sortierten Baum. Der Abschnitt über das Speichern von Daten kommt darauf zurück.

Durchsuchen oder Nachschlagen

Ein Durchsuchen erfordert Arbeit proportional zur Anzahl der Elemente; ein Dictionary geht direkt zum richtigen Slot.

Platz gegen Zeit

Das Dictionary im ausgearbeiteten Beispiel ist schneller, weil es zusätzlichen Speicher nutzt, um die Daten im Voraus zu organisieren. Das ist eine der häufigsten Abwägungen in der Informatik: Oft lässt sich Geschwindigkeit mit Platz erkaufen oder Platz sparen, indem man Zeit aufwendet.

Ein Index sind zusätzliche Daten neben den Hauptdaten, die eine bestimmte Art von Frage schnell beantworten, wie das Stichwortverzeichnis hinten in einem Buch. Er beschleunigt Lesezugriffe, belegt aber Speicher und muss bei jedem Schreibvorgang aktualisiert werden, sodass das Einfügen in eine Tabelle mit zehn Indizes langsam ist. Ein Cache hält eine Kopie der Antwort auf eine aufwendige Frage an einem schnellen Ort bereit, etwa im Arbeitsspeicher, sodass die nächste Person, die fragt, sie sofort erhält. Caches sind der Grund, warum eine beliebte Produktseite schnell lädt, selbst wenn die Datenbank dahinter ausgelastet ist, aber sie werfen eine schwierige Frage auf: Wenn sich die zugrunde liegenden Daten ändern, wie schnell muss sich die Kopie mitändern? Ein paar Sekunden lang einen veralteten Preis anzuzeigen, mag akzeptabel sein; einen veralteten Kontostand anzuzeigen, möglicherweise nicht.

Kompression ist der umgekehrte Tausch: CPU-Zeit aufwenden, um Daten in weniger Bytes zu pressen, was Speicher und Übertragungszeit im Netzwerk spart.

ACHTUNG

Verbreitete Irrtümer

- **„Die O-Notation sagt Ihnen, wie schnell Code läuft."** Sie sagt Ihnen, wie die Kosten wachsen. Bei zehn Elementen kann eine $O(n^2)$-Methode schneller sein als eine raffinierte O(n log n)-Methode; die O-Notation zählt, wenn n groß wird.
- **„Schnellere Hardware behebt langsame Algorithmen."** Ein doppelt so schneller Rechner lässt einen quadratischen Algorithmus in derselben Zeit nur etwa 1,4-mal so viele Daten verarbeiten.
- **„Dictionary-Zugriffe sind immer sofort erledigt."** Sie sind im Durchschnitt O(1), brauchen aber mehr Speicher als eine Liste und haben keine Reihenfolge, sodass „alle Kunden, deren Namen mit C beginnen" trotzdem ein Durchsuchen oder eine sortierte Struktur erfordert.
- **„Floats sind für Geld in Ordnung."** Rundungsfehler summieren sich über Millionen von Transaktionen. Verwenden Sie Ganzzahlen der kleinsten Einheit oder einen Dezimaltyp.

VERTIEFUNG

Vertiefung: Durchschnitt, ungünstigster Fall und Hashing

Ein Dictionary erreicht im Durchschnitt O(1), indem es jeden Schlüssel durch eine *Hashfunktion* schickt, die den Schlüssel in eine Zahl verwandelt, die einen Slot auswählt. Zwei Schlüssel können im selben Slot landen (eine *Kollision*); Implementierungen fangen das ab, und in einem pathologischen Fall, in dem viele Schlüssel kollidieren, verschlechtern sich die Zugriffe in Richtung O(n). Das ist nicht nur Theorie: Angreifer haben Webservern Anfrageparameter geschickt, die gezielt so gebaut waren, dass sie kollidieren, weshalb moderne Sprachen, auch Python, ihr String-Hashing randomisieren. Datenbankindizes verwenden meist einen balancierten Baum namens B-Baum, der Zugriffe in O(log n) ermöglicht und anders als eine Hashtabelle auch Bereiche und sortierte Reihenfolge unterstützt.

Warum das für die Architektur wichtig ist

Architekten schreiben selten Sortieralgorithmen, treffen aber ständig Entscheidungen, deren Folgen genau diese sind. Welche Fragen muss das System schnell beantworten, und haben die Daten die Indizes, um sie zu beantworten? Wie groß werden die Daten in drei Jahren sein, und welche Teile des Entwurfs wachsen dabei linear oder schlechter? Wo lohnt sich Caching trotz seiner Komplexität, und wie veraltet dürfen gecachte Daten sein? Welche Komponente bricht zuerst, wenn das Datenvolumen um das Zehnfache wächst?

Das sind Fragen zu Performance und Skalierbarkeit, die Kapitel 2 präzisiert, sowie zu Caching und zur Wahl der Datenbank, die Kapitel 7 ausführlich behandelt. Der nächste Abschnitt wendet sich von den Daten dem Code zu und der Frage, wie Code organisiert wird, damit Menschen ihn verstehen und ändern können.

PRÜFEN SIE IHR VERSTÄNDNIS

▼ Warum ergibt `0.1 + 0.2` in den meisten Sprachen nicht genau 0,3?

Gleitkommazahlen werden binär gespeichert, und viele Dezimalbrüche, darunter 0,1 und 0,2, haben keine exakte binäre Form, sodass winzige Rundungsfehler auftreten.

▼ Welche Datenstruktur eignet sich für eine Rückgängig-Schaltfläche in einem Texteditor, und warum?

Ein Stack: Jede Aktion wird oben abgelegt, und „Rückgängig“ entnimmt die jüngste zuerst (zuletzt hinein, zuerst heraus).

▼ Wenn ein O(n)-Vorgang für 100.000 Elemente eine Sekunde dauert, wie lange ungefähr für 1.000.000?

Etwa zehn Sekunden, weil linearer Aufwand proportional zu den Daten wächst.

▼ Womit bezahlen Sie die Geschwindigkeit, die Ihnen ein Datenbankindex verschafft?

Mit zusätzlichem Speicher und zusätzlicher Arbeit bei jedem Einfügen, Aktualisieren und Löschen, um den Index aktuell zu halten.

QUELLEN

- Python tutorial: data structures: https://docs.python.org/3/tutorial/datastructures.html
- Python wiki: time complexity of built-in operations: https://wiki.python.org/moin/TimeComplexity
- Unicode HOWTO (Python documentation): https://docs.python.org/3/howto/unicode.html
- RFC 3629: UTF-8, a transformation format of ISO 10646: https://www.rfc-editor.org/rfc/rfc3629

ABSCHNITT 0.3 VON 0.8 · EINSTEIGER · 16 MIN. LESEZEIT

Funktionen, Module, Objekte, Schnittstellen, Bibliotheken und Frameworks

DIE IDEE IN KÜRZE

Denken Sie an ein großes Restaurant. Statt eines Kochs, der alles macht, gibt es Stationen (Grill, Patisserie, Salate), jede mit eigenem Personal, eigenen Werkzeugen und Rezepten. Bestellungen wandern über eine Durchreiche mit einem Bonsystem von Station zu Station, sodass der Grill nicht wissen muss, wie der Patissier arbeitet. Code ist genauso organisiert: kleine benannte Aufgaben (Funktionen) werden zu Stationen (Modulen) gruppiert, und die Stationen verkehren miteinander nur über vereinbarte Bons (Schnittstellen). So kann eine Station ihre Arbeitsweise ändern, ohne den Rest der Küche durcheinanderzubringen.

WAS SIE WISSEN MÜSSEN

- Eine Funktion ist ein benanntes, wiederverwendbares Stück Code mit Eingaben und einer Ausgabe; mit ihr lässt sich eine Regel einmal schreiben und an einer Stelle ändern.
- Module und Pakete fassen zusammengehörigen Code zusammen und trennen einen kleinen öffentlichen Teil von privaten Details, die sich frei ändern dürfen.
- Klassen bündeln Daten mit den Methoden, die darauf arbeiten; Kapselung hält die Daten privat, sodass die Regeln immer durchgesetzt werden.
- Eine Schnittstelle ist ein Vertrag zwischen Teilen; Code, der von einer Schnittstelle statt von einer Implementierung abhängt, kann Implementierungen leicht austauschen.
- Eine Bibliothek rufen Sie auf; ein Framework ruft Ihren Code auf. Diese Inversion of Control macht Frameworks leistungsfähig, aber schwerer zu ersetzen.
- Abhängigkeiten, die Paketmanager wie pip und npm beschaffen, sparen Aufwand, bringen aber transitiven Code, Sicherheitsrisiken und Wartungsarbeit mit sich.
- Gute Struktur bedeutet lose Kopplung und hohe Kohäsion; das hält Änderungen günstig und lässt Teams unabhängig arbeiten.

Warum Code überhaupt organisieren?

Ein kleines Skript mit zwanzig Zeilen lässt sich in einer Minute von oben nach unten lesen. Ein echtes System, etwa eine Essenslieferplattform, kann Hunderttausende oder Millionen Zeilen umfassen, geschrieben von Dutzenden Menschen über viele Jahre. Niemand kann das im Kopf

behalten. Die einzige Möglichkeit, an einem solchen System zu arbeiten, besteht darin, es in Teile zu zerlegen, die sich einzeln verstehen, testen und ändern lassen, mit klaren Regeln dafür, wie die Teile einander nutzen dürfen.

Programmiersprachen bieten dafür eine Leiter von Bausteinen, vom Kleinen zum Großen: Funktionen, Module und Pakete, Klassen und Objekte und die Schnittstellen zwischen ihnen. Über dem Code, den wir selbst schreiben, liegen Bibliotheken und Frameworks, die andere geschrieben haben. Dieser Abschnitt steigt diese Leiter hinauf. Dieselben Ideen, in größerem Maßstab angewandt, sind der Gegenstand der Softwarearchitektur.

Funktionen: eine Aufgabe benennen und wiederverwenden

Eine Funktion ist ein benanntes Stück Code, das eine Aufgabe erledigt. Sie nimmt *Eingaben* entgegen (Parameter oder Argumente genannt), verrichtet ihre Arbeit und gibt meist eine *Ausgabe* zurück (ihren Rückgabewert). Denken Sie an einen Kaffeevollautomaten: Sie geben Bohnen, Wasser und eine Getränkewahl hinein; er erledigt seine Arbeit; heraus kommt ein Kaffee. Sie müssen nicht wissen, wie die Pumpe funktioniert, um ihn zu benutzen.

Funktionen bringen drei Vorteile. **Wiederverwendung**: Die Mehrwertsteuerberechnung wird einmal geschrieben und vom Checkout, von der Rechnung und von der Erstattungsmaske aufgerufen. **Lesbarkeit**: `total_with_vat(basket)` sagt, was geschieht, ohne zu zeigen, wie. **Änderung an einer Stelle**: Wenn sich der Mehrwertsteuersatz ändert, ändert sich eine Funktion, nicht zwanzig Kopien derselben Rechnung, verstreut über den Code (Kopien, die bald auseinanderlaufen würden).

Eine gute Funktion tut eine Sache, hat einen Namen, der sagt, was sie tut, erhält die benötigten Eingaben als Parameter, statt nach verborgenen Daten zu greifen, und ist kurz genug, um sie auf einen Blick zu erfassen. Eine Funktion, die ihre Ausgabe nur aus ihren Eingaben berechnet, ohne sonst etwas zu verändern, heißt *reine* Funktion; reine Funktionen sind besonders leicht zu testen, weil dieselben Eingaben immer dieselbe Ausgabe liefern.

```
VAT_RATE = 0.20

def total_with_vat(prices_in_pence):
    """Return the total of a basket in pence, including VAT."""
    subtotal = sum(prices_in_pence)
    return round(subtotal * (1 + VAT_RATE))

checkout_total = total_with_vat([1250, 399])   # 1979
invoice_total = total_with_vat([5000])         # 6000
```

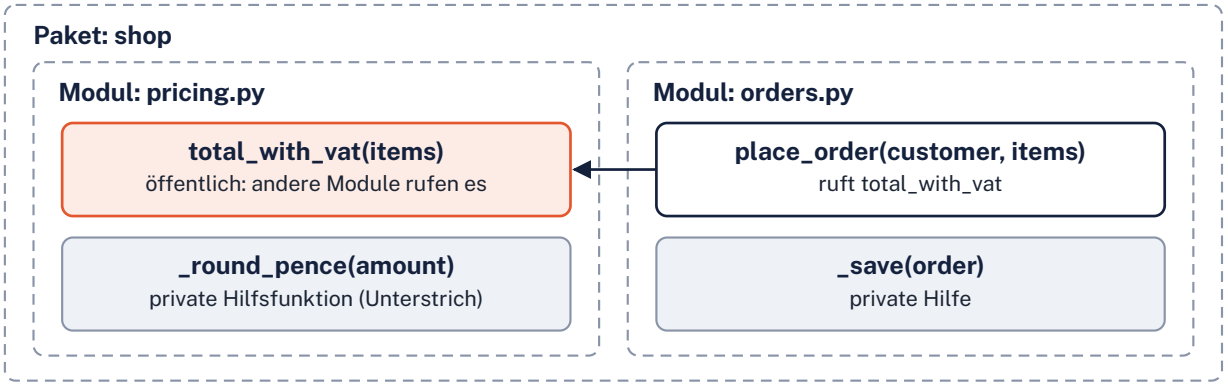
Eine Funktion, an mehreren Stellen wiederverwendet

Module und Pakete: zusammengehörigen Code gruppieren

Ein Modul fasst zusammengehörige Funktionen und Daten zu einer Einheit mit eigenem Namen zusammen. In Python ist ein Modul einfach eine `.py`-Datei; `pricing.py` könnte alles enthalten, was mit Preisen, Rabatten und Steuern zu tun hat. Eine andere Datei nutzt es, indem sie `import pricing` schreibt. Ein **Paket** ist ein Ordner zusammengehöriger Module; ein Shop könnte also ein Paket `shop` mit den Modulen `pricing`, `orders` und `customers` haben. Andere Sprachen verwenden andere Begriffe (Java hat Packages, C# hat Namespaces und Assemblies, JavaScript hat Module), aber die Idee ist dieselbe.

Ein Modul ist wie eine Abteilung in einer Organisation. Die Finanzabteilung hat einen öffentlichen Empfang, an dem andere Anliegen vorbringen können („Bitte stellen Sie diese Rechnung aus“), und interne Abläufe, die niemand außerhalb sehen muss. Ebenso trennen Module einen kleinen **öffentlichen** Teil, den andere Module nutzen sollen, von **privaten** Details. Python kennzeichnet private Namen per Konvention mit einem führenden Unterstrich, etwa `_round_pence`; Java und C# erzwingen die Privatheit mit Schlüsselwörtern wie `private`. Der Wert dieser Trennung liegt darin, dass sich der private Teil frei neu schreiben lässt, ohne bei anderen etwas kaputt zu machen, solange der öffentliche Teil seine Versprechen hält.

Funktionen, Module und ein Paket



Funktionen liegen in Modulen; verwandte Module bilden ein Paket; nur öffentliche Namen sind für außen gedacht

Das Modul orders nutzt nur die öffentliche Funktion des Moduls pricing, nie dessen private Hilfsfunktion.

Klassen und Objekte: Daten mit ihrem Verhalten

Viele Sprachen erlauben es, Daten mit den Funktionen zu bündeln, die mit ihnen arbeiten. Eine **Klasse** ist die Vorlage; ein **Objekt** ist ein einzelnes Exemplar, das daraus erzeugt wird. Eine Klasse `BankAccount` könnte festlegen, dass jedes Konto einen Inhaber und einen Kontostand

hat und Einzahlungen und Abhebungen annehmen kann. Das Konto jedes Kunden ist dann ein eigenes Objekt dieser Klasse. Funktionen, die zu einer Klasse gehören, heißen *Methoden*. (Siehe den Glossareintrag zu Klasse und Objekt.)

Die für die Architektur nützlichste Idee hierbei ist die Kapselung: Ein Objekt hält seine Daten privat und lässt anderen Code sie nur über seine Methoden ändern. Könnte jeder `account.balance` direkt setzen, würde nichts Code an anderer Stelle daran hindern, einen Kontostand negativ zu machen oder das Audit-Log zu umgehen. Mit Kapselung ist die Methode `withdraw` die einzige Tür, und sie kann die Regeln jedes Mal durchsetzen. Die Analogie ist ein Bankschalter: Kunden können nicht in den Tresorraum spazieren, aber sie können den Schalterbeamten fragen, der die Regeln prüft, bevor er Geld aushändigt.

Die objektorientierte Programmierung umfasst noch viel mehr (Vererbung, Polymorphie, Entwurfsmuster), und Fachleute sind uneins darüber, wie viel davon hilfreich ist: Manche Sprachen, etwa Go, verzichten bewusst auf Vererbung, und viele Teams bevorzugen einen eher funktionalen Stil. Kapselung ist jedoch in jedem Stil wertvoll.

```
class BankAccount:
    def __init__(self, owner):
        self.owner = owner
        self._balance = 0          # private by convention

    def deposit(self, pence):
        if pence ⩽ 0:
            raise ValueError("Deposit must be positive")
        self._balance += pence

    def withdraw(self, pence):
        if pence > self._balance:
            raise ValueError("Insufficient funds")
        self._balance -= pence

    def balance(self):
        return self._balance

acct = BankAccount("Ana")
acct.deposit(10000)
acct.withdraw(2500)
print(acct.balance())   # 7500
```

Kapselung: Der Kontostand kann sich nur über Methoden ändern, die die Regeln durchsetzen

Schnittstellen: Verträge zwischen Teilen

Eine Schnittstelle ist die vereinbarte Menge von Operationen, die ein Teil eines Systems anderen anbietet: ihre Namen, welche Eingaben sie erwarten, was sie zurückgeben und was schiefgehen kann. Sie ist ein Vertrag. Der Aufrufer verlässt sich auf den Vertrag, nicht darauf, wie die Arbeit erledigt wird. Eine Steckdose ist eine Schnittstelle: Jedes Gerät mit dem richtigen Stecker funktioniert, und dem Gerät ist es gleich, ob der Strom aus Wind, Gas oder Kernkraft stammt.

Im Code erlaubt eine Schnittstelle, eine Implementierung gegen eine andere auszutauschen. Angenommen, ein Shop wickelt Kartenzahlungen heute über einen Zahlungsanbieter ab, könnte aber im nächsten Jahr wechseln. Ruft der Checkout-Code überall direkt die eigenen Funktionen des Anbieters auf, bedeutet ein Wechsel, den Checkout neu zu schreiben. Definiert der Shop stattdessen eine eigene kleine Schnittstelle `PaymentGateway` mit einer Operation, `charge(amount, card_token)`, und schreibt dann einen Adapter pro Anbieter, hängt der Checkout nur von der Schnittstelle ab, und ein Anbieterwechsel bedeutet, einen neuen Adapter zu schreiben. Derselbe Kniff erleichtert das Testen: In Tests kann ein gefälschtes Gateway, das nie eine echte Bank kontaktiert, das echte ersetzen.

Schnittstellen sind die wichtigste Idee dieses Abschnitts für Architekten. Jede Grenze in einem System, von einer Funktionssignatur bis zu einer Web-API, die Tausende anderer Unternehmen nutzen, ist eine Schnittstelle. Gute Schnittstellen sind klein und stabil und verbergen Entscheidungen, die sich wahrscheinlich ändern; dieses Prinzip, Information Hiding genannt, formulierte David Parnas 1972, und es wird in Kapitel 3 behandelt.

```
from typing import Protocol

class PaymentGateway(Protocol):
    def charge(self, amount_pence: int, card_token: str) → bool: ...

class ProviderAGateway:
    def charge(self, amount_pence, card_token):
        ...  # call Provider A's web API
        return True

class FakeGateway:                      # used in tests
    def charge(self, amount_pence, card_token):
        return amount_pence < 100_000   # pretend large payments fail

def checkout(basket_total, card_token, gateway: PaymentGateway):
    return "paid" if gateway.charge(basket_total, card_token) else "declined"
```

Eine Schnittstelle (hier ein Python-Protocol) mit zwei austauschbaren Implementierungen

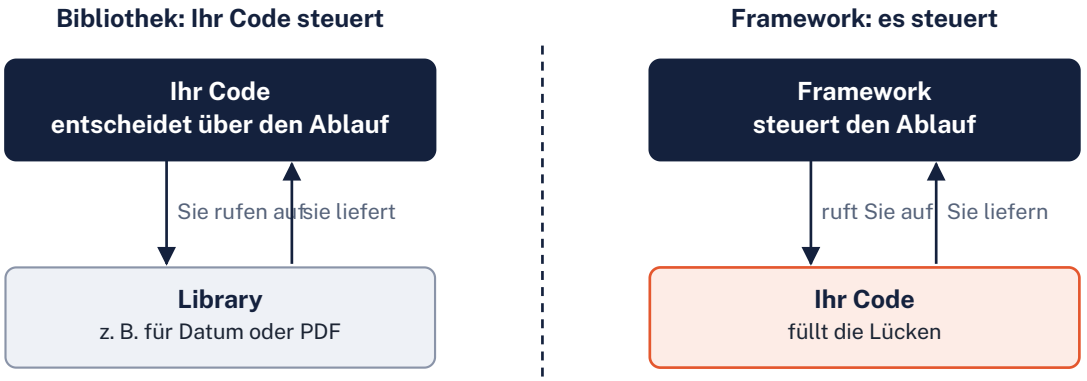
Bibliotheken und Frameworks: Code, den andere geschrieben haben

Kaum jemand schreibt alles von Grund auf selbst. Eine Bibliothek ist wiederverwendbarer Code, den jemand anderes geschrieben hat und den Ihr Code bei Bedarf aufruft: eine Bibliothek für Datumsangaben und Zeitzonen, für das Erzeugen von PDFs, für den Zugriff auf eine Datenbank. Sie behalten die Kontrolle; die Bibliothek ist ein Werkzeug, das Sie zur Hand nehmen und wieder weglegen.

Ein Framework ist etwas anderes. Es liefert das Grundgerüst einer Anwendung und entscheidet, wann was geschieht; Sie füllen die Lücken mit Ihrem eigenen Code, und das Framework ruft diesen Code im richtigen Moment auf. Ein Web-Framework wie Django (Python), Spring (Java), Ruby on Rails oder Express (JavaScript) empfängt jede eingehende Webanfrage, ermittelt, welche Ihrer Funktionen sie bearbeiten soll, ruft diese Funktion auf und macht aus ihrem Ergebnis eine Webseite oder Daten. Die Kurzformel für den Unterschied: *Eine Bibliothek rufen Sie auf; ein Framework ruft Ihren Code auf*.

Diese Umkehrung heißt Inversion of Control. Martin Fowler schreibt die Verwendung des Begriffs im Jahr 1988 Ralph Johnson und Brian Foote zu, und die Idee wird oft als Hollywood-Prinzip zusammengefasst: „Rufen Sie uns nicht an, wir rufen Sie an“. Ein Franchise-Restaurant ist eine treffende Analogie: Der Franchisegeber stellt das Gebäude, das Bestellsystem, die Öffnungszeiten und die Regeln; der Franchisenehmer liefert das Kochen in den Lücken, die der Franchisegeber vorsieht.

Wer ruft wen auf?



Die Richtung der Steuerung ist der eigentliche Unterschied zwischen einer Bibliothek und einem Framework.

Bibliothek oder Framework

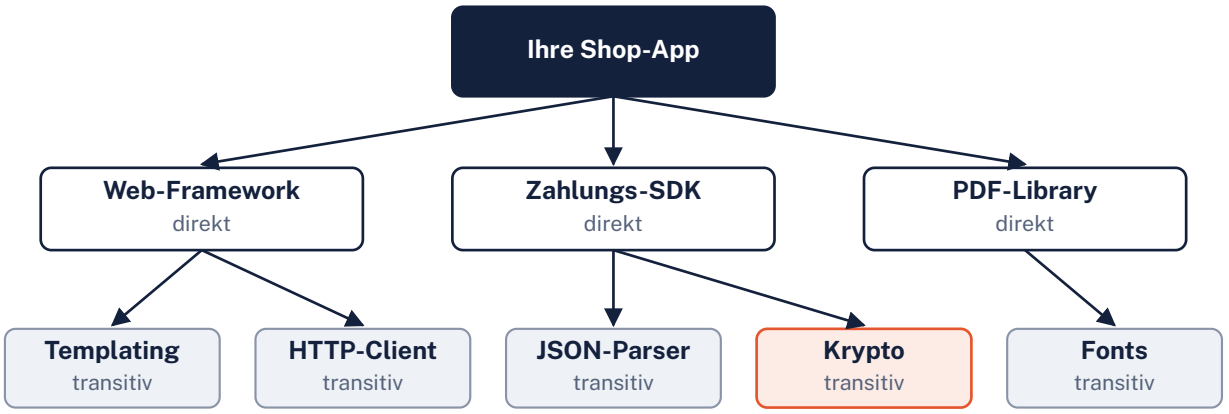
	Bibliothek	Framework
Wer steuert den Ablauf	Ihr Code	Das Framework (Inversion of Control)
Beispiele	Eine Datumsbibliothek, ein PDF-Generator, ein JSON-Parser	Django, Spring, Ruby on Rails, Angular
Wie leicht später zu ersetzen	Meist mäßig: die Stellen ändern, die sie aufrufen	Meist schwer: Die Struktur des Frameworks prägt die ganze Anwendung
Hauptnutzen	Erspart, eine Fähigkeit selbst zu schreiben	Erspart den Entwurf der gesamten Struktur; einheitliche Konventionen im ganzen Team
Hauptrisiko	Eine weitere Abhängigkeit, die gepflegt werden muss	Lock-in an seine Arbeitsweise und seinen Upgrade-Zyklus

Abhängigkeiten und Paketmanager

Jede Bibliothek und jedes Framework, das Ihr Code benötigt, ist eine Abhängigkeit. Moderne Projekte haben viele davon und nutzen daher einen Paketmanager, um sie zu beschaffen: `pip` für Python (aus dem Python Package Index, PyPI) und `npm` für JavaScript (aus der npm-Registry) sind die bekanntesten, daneben gibt es Entsprechungen wie Maven für Java und NuGet für .NET. Ein Projekt listet auf, was es braucht, oft mit Versionsbereichen, und der Paketmanager lädt diese Pakete herunter und, ganz entscheidend, auch die Pakete, von denen *sie* abhängen. Diese indirekten, *transitiven* Abhängigkeiten übertreffen die direkten oft um ein Vielfaches.

Versionen werden üblicherweise nach dem Schema MAJOR.MINOR.PATCH nummeriert, gemäß einer Konvention namens semantische Versionierung: Eine Änderung der Major-Nummer signalisiert eine inkompatible Änderung, eine Minor-Nummer neue Funktionen, die bestehende Nutzer nicht beeinträchtigen sollten, und eine Patch-Nummer Fehlerbehebungen. Projekte führen außerdem ein *Lockfile* (etwa `package-lock.json` für npm), das die genaue Version jedes installierten Pakets festhält, damit der Rechner jedes Entwicklers und der Produktions-Build denselben Code verwenden.

Direkte und transitive Abhängigkeiten



Sie wählten drei Pakete; Sie hängen von acht ab. Ein Fehler in einem davon (z. B. crypto) ist ein Fehler Ihrer App.

Der größte Teil des Codes, den Ihre Anwendung ausführt, geht auf die Entscheidungen anderer zurück.

Abhängigkeiten ersparen enormen Aufwand, aber jede ist Code, den Sie nicht geschrieben haben und der mit allen Berechtigungen Ihres eigenen Codes läuft. Zwei bekannte Vorfälle zeigen die Risiken. Im März 2016 entfernte ein Entwickler im Zuge eines Streits seine Pakete aus npm, darunter `left-pad`, eine winzige Funktion von etwa elf Zeilen zum Auffüllen von Text; Tausende Projekte, die direkt oder indirekt davon abhingen, ließen sich plötzlich nicht mehr bauen, bis npm das Paket wiederherstellte. Im Dezember 2021 wurde in Log4j 2, einer weit verbreiteten Java-Logging-Bibliothek, eine schwere Sicherheitslücke mit dem Spitznamen Log4Shell bekannt; Organisationen in aller Welt verbrachten Wochen damit, jedes System zu finden, das sie enthielt, oft mehrere Ebenen tief in ihren Abhängigkeiten vergraben.

Checkliste: bevor Sie eine Abhängigkeit hinzufügen

Fragen, die ein sorgfältiges Team stellt, bevor es sich auf das Paket eines anderen verlässt.

- [ ] **Brauchen wir es?**
  Ein paar Zeilen eigener Code können auf Dauer günstiger sein als ein Paket, das verfolgt und aktualisiert werden muss.
- [ ] **Wird es gepflegt?**
  Achten Sie auf aktuelle Releases, offene Sicherheitsprobleme und darauf, wie viele Personen es pflegen. Ein verwaistes Paket wird zu Ihrem Problem.
- [ ] **Was bringt es mit?**
  Prüfen Sie, wie viele transitive Abhängigkeiten es hinzufügt; jede ist weiterer Code, dem man vertrauen muss.
- [ ] **Ist die Lizenz kompatibel?**
  Open-Source-Lizenzen unterscheiden sich; manche bringen Pflichten mit sich, wenn Sie Ihre Software weitergeben.
- [ ] **Sind die Versionen fixiert und geprüft?**
  Verwenden Sie ein Lockfile und ein automatisiertes Werkzeug, das warnt, wenn eine bekannte Schwachstelle eine von Ihnen genutzte Version betrifft.
- [ ] **Könnten wir es ersetzen?**
  Kapseln Sie wichtige Abhängigkeiten hinter einer eigenen Schnittstelle, sodass ein Anbieterwechsel nur eine Stelle berührt.

Warum Struktur wichtig ist: Änderungen und Teams

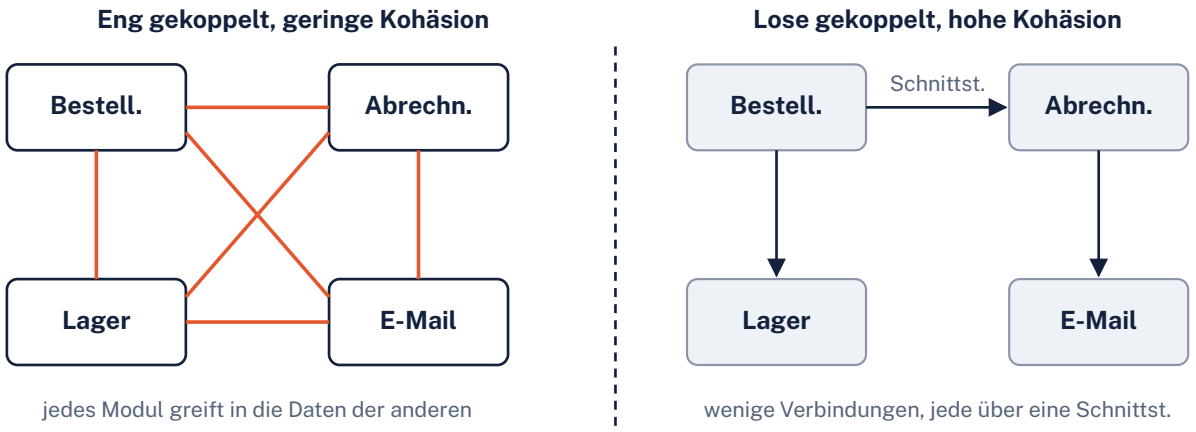
Code wird weit häufiger gelesen und geändert, als er anfangs geschrieben wird. Der größte Teil der Lebenszeitkosten von Software entfällt auf Änderungen: Fehler beheben, Features hinzufügen, sich an neue Gesetze und neue Partner anpassen. Die Struktur entscheidet, wie teuer das ist. In einem gut organisierten System berührt eine Änderung an der Mehrwertsteuerberechnung ein Modul. In einem schlecht organisierten berührt sie vierzig Dateien, die fünf Teams gehören, und irgendetwas, das nichts damit zu tun hat, geht kaputt.

Zwei Ideen, erstmals beschrieben von Larry Constantine und Kollegen in der Structured-Design-Bewegung der späten 1960er- und der 1970er-Jahre, erfassen, was „gut organisiert“ bedeutet. Kopplung beschreibt, wie stark ein Teil von den Details eines anderen abhängt: In einem eng gekoppelten System erzwingt die Änderung eines Teils Änderungen an anderen.

Kohäsion beschreibt, wie gut der Inhalt eines Teils zusammengehört: Ein Modul mit hoher Kohäsion erledigt eine klare Aufgabe. Das Ziel ist **lose Kopplung und hohe Kohäsion**: Teile, die jeweils eine Aufgabe gut erledigen und über kleine, stabile Schnittstellen miteinander sprechen.

Struktur ist auch für Menschen wichtig. Müssen zwei Teams für den Großteil ihrer Arbeit dasselbe Modul bearbeiten, werden sie ständig aufeinander warten und sich gegenseitig etwas kaputt machen. Besitzt jedes Team Module mit hoher Kohäsion und klaren Schnittstellen, kann es unabhängig arbeiten und releasen. Melvin Conway beobachtete 1968, dass Systeme dazu neigen, die Kommunikationsstruktur der Organisationen abzubilden, die sie bauen; diese Beobachtung ist als Gesetz von Conway bekannt und wird in Kapitel 3 untersucht.

Kopplung auf einen Blick



Dieselben vier Module, auf zwei Arten verbunden: Jede zusätzliche Verbindung ist ein weiterer Weg, auf dem sich eine Änderung ausbreiten kann.

ACHTUNG

Häufige Fehler

- **Kopieren und Einfügen statt einer Funktion.** Duplizierte Logik läuft auseinander, sodass dieselbe Regel an verschiedenen Stellen unterschiedliche Ergebnisse liefert.
- **Alles öffentlich.** Wenn jedes Detail eines Moduls von außen genutzt werden kann, wird jedes Detail zu etwas, wovon anderer Code abhängt, und nichts lässt sich mehr gefahrlos ändern.
- **Framework-Code überall.** Wenn Geschäftsregeln mit Framework-Aufrufen verwoben sind, bedeutet ein Upgrade oder Austausch des Frameworks, das Geschäft neu zu schreiben.
- **Ungeprüfte Abhängigkeiten.** Ein Paket für eine triviale Aufgabe hinzuzufügen bringt seinen ganzen Abhängigkeitsbaum mit, seine Bugs und seine Sicherheitsrisiken.

Warum das für die Architektur wichtig ist

Architektur ist weitgehend dieselbe Tätigkeit wie in diesem Abschnitt, nur in größerem Maßstab und mit höherem Einsatz: entscheiden, was die Hauptteile eines Systems sind, wofür jedes zuständig ist, welche Schnittstellen sie verbinden und in welche Richtung die Abhängigkeiten zeigen. Ein Modul innerhalb eines Programms und ein separat deployter Service jenseits eines Netzwerks sind beide Teile mit Schnittstellen; die Fragen nach Kopplung, Kohäsion und Information Hiding gelten für beide.

Kapitel 3 macht diese Prinzipien präzise, beginnend mit Modularität und Kopplung und Kohäsion, und Kapitel 8 greift Abhängigkeiten als Sicherheitsthema wieder auf, in Sicherheit der Software-Lieferkette. Als Nächstes verlässt der Code den einzelnen Computer: Der folgende Abschnitt erklärt, wie Computer miteinander kommunizieren.

PRÜFEN SIE IHR VERSTÄNDNIS

▼ **Warum ist es besser, eine Funktion `total_with_vat` von drei Stellen aus aufzurufen, als die Berechnung dreimal zu schreiben?**

Die Regel steht dann an einer Stelle: Ändert sich der Mehrwertsteuersatz, korrigiert eine Änderung jeden Aufrufer, und die Kopien können nicht auseinanderlaufen.

▼ **Woran hindert die Kapselung anderen Code bei einem Bankkonto-Objekt?**

Sie verhindert, dass anderer Code den Kontostand direkt ändert, sodass jede Änderung über Methoden wie withdraw laufen muss, die die Regeln durchsetzen.

▼ **Ein Web-Framework ruft Ihre Funktion auf, sobald eine Anfrage für `/orders` eintrifft. Wie nennt man diese Anordnung?**

Inversion of Control: Das Framework, nicht Ihr Code, steuert den Ablauf und ruft Ihren Code bei Bedarf auf.

▼ **Wozu dient ein Lockfile wie `package-lock.json`?**

Es hält die genaue Version jedes installierten Pakets fest, auch der indirekten, sodass jeder Entwicklerrechner und jeder Produktions-Build genau denselben Code verwenden.

QUELLEN

- Python tutorial: modules and packages: https://docs.python.org/3/tutorial/modules.html
- Python tutorial: classes: https://docs.python.org/3/tutorial/classes.html
- Martin Fowler: Inversion of Control: https://martinfowler.com/bliki/InversionOfControl.html

- Semantic Versioning 2.0.0: https://semver.org/
- npm documentation: package-lock.json: https://docs.npmjs.com/cli/v10/configuring-npm/package-lock-json

Netzwerke, Internet und Web: IP, DNS, HTTP und das Client-Server-Modell

DIE IDEE IN KÜRZE

Daten über das Internet zu senden ist wie Post zu verschicken. Ihre Nachricht wird in Päckchen (Pakete) aufgeteilt, jedes mit Absender- und Empfängeradresse (IP-Adressen) und einer Zimmernummer (einem Port), und Sortierzentren (Router) reichen sie weiter. Namen wie shop.example werden in einem riesigen gemeinsamen Adressbuch (DNS) nachgeschlagen, um die echte Adresse zu finden. Die Briefe selbst folgen einem festen Format (HTTP): eine Anfrage wie „Bitte schicken Sie mir Produkt 42“ und eine Antwort mit einem Code, der sagt, wie es gelaufen ist. HTTPS steckt den Brief in einen versiegelten, manipulationssicheren Umschlag.

WAS SIE WISSEN MÜSSEN

- Daten durchqueren Netzwerke in Paketen, die Router anhand der IP-Adresse des Ziels Hop für Hop weiterleiten; eine Portnummer kennzeichnet das Programm auf diesem Rechner.
- TCP bietet nach einem Handshake eine zuverlässige, geordnete Verbindung; UDP sendet einzelne Pakete ohne Garantien, aber mit weniger Verzögerung.
- DNS übersetzt Namen in IP-Adressen über eine Hierarchie aus Root-, Top-Level-Domain- und autoritativen Servern, mit starkem Caching, das über TTLs gesteuert wird.
- HTTP ist ein zustandsloses Anfrage-Antwort-Protokoll: eine Methode (GET, POST, PUT, DELETE), ein Pfad, Header und ein Body, beantwortet mit einem Statuscode (2xx, 3xx, 4xx, 5xx).
- HTTPS ist HTTP über TLS; TLS verschlüsselt den Datenverkehr, erkennt Manipulationen und nutzt von vertrauenswürdigen Stellen signierte Zertifikate, um die Identität des Servers nachzuweisen.
- Im Client-Server-Modell starten Clients Anfragen, und Server halten gemeinsame Daten und Regeln; Server dürfen Clients nie blind vertrauen.
- Latenz (Verzögerung) und Bandbreite (Kapazität) sind verschiedene Dinge; die Latenz ist durch die Lichtgeschwindigkeit begrenzt, deshalb zählen Entfernung und Roundtrips.

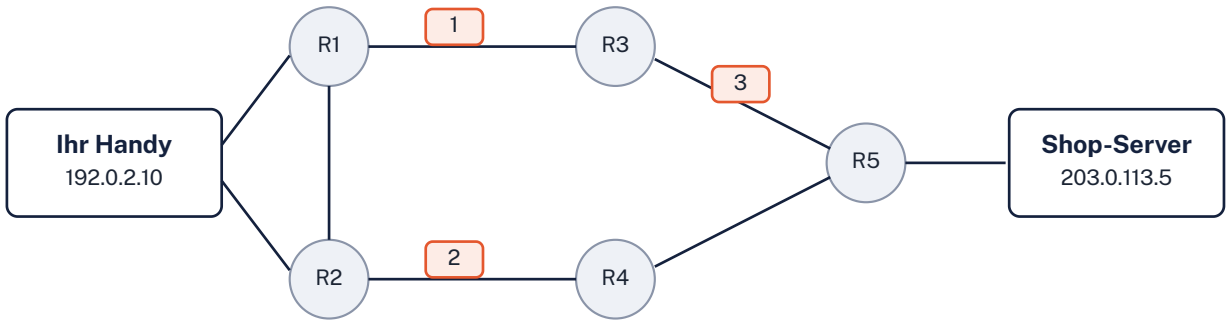
Netzwerke und Pakete

Kaum eine moderne Software läuft für sich allein. Eine Essensliefer-App auf Ihrem Smartphone kommuniziert mit den Servern des Unternehmens, die wiederum mit den Tablets der Restaurants, einem Zahlungsanbieter, einem Kartendienst und einem Benachrichtigungsdienst kommunizieren. Dieser Abschnitt erklärt, wie diese Kommunikation funktioniert, von unten nach oben.

Ein **Netzwerk** ist eine Gruppe von Computern, die so verbunden sind, dass sie Daten austauschen können, per Kabel, WLAN, Mobilfunk oder Glasfaser. Das **Internet** ist ein Netz aus Netzen: Millionen Netzwerke unterschiedlicher Eigentümer (Breitbandanschlüsse zu Hause, Mobilfunkbetreiber, Universitäten, Cloud-Anbieter), die vereinbart haben, den Datenverkehr der jeweils anderen nach gemeinsamen Regeln, sogenannten *Protokollen*, zu befördern.

Daten reisen nicht als ein durchgehender Strom. Sie werden in Pakete zerlegt: Stücke von typischerweise bis zu etwa 1500 Bytes, jedes mit Herkunft und Ziel beschriftet, wie ein Päckchen mit Absender- und Empfängeradresse. Geräte namens **Router** reichen Pakete von Netzwerk zu Netzwerk weiter und wählen dabei jeweils einen guten nächsten Hop, ähnlich wie Sortierzentren Päckchen in Richtung ihres Ziels weitergeben. Verschiedene Pakete derselben Nachricht können unterschiedliche Wege nehmen, in falscher Reihenfolge ankommen oder gelegentlich verloren gehen, und der empfangende Computer setzt sie wieder zusammen. Die Aufteilung in Pakete lässt Millionen Gespräche dieselben Kabel fair teilen und den Datenverkehr eine unterbrochene Verbindung umgehen.

Pakete finden ihren eigenen Weg



Eine Nachricht wird in nummerierte Pakete zerlegt, die einzeln über Router reisen und am Ziel zusammenfinden

Kein einzelnes Kabel verbindet Ihr Smartphone mit dem Server; Pakete springen zwischen Routern, die vielen verschiedenen Netzwerken gehören.

Adressen: IP-Adressen und Ports

Damit ein Paket ankommt, braucht das Ziel eine Adresse. Jedes Gerät im Internet wird über eine IP-Adresse erreicht (IP steht für Internet Protocol). Die ältere Version, IPv4, schreibt Adressen als vier Zahlen von 0 bis 255, etwa `192.0.2.10`; sie erlaubt etwa 4,3 Milliarden Adressen, was nicht für jedes Smartphone, jeden Laptop und jeden Sensor der Welt reicht. Das neuere IPv6 verwendet viel längere Adressen in hexadezimaler Schreibweise (Basis 16, mit den Ziffern 0 bis 9 und den Buchstaben a bis f), etwa `2001:db8::1`, und bietet Platz für praktisch unbegrenzt viele Geräte. Beide sind nebeneinander im Einsatz. (Die Beispieladressen in diesem Abschnitt stammen aus Bereichen, die für Dokumentationszwecke reserviert sind, und gehören daher zu keinem echten Rechner.)

Eine IP-Adresse bringt ein Paket zum richtigen Computer, aber ein Computer führt viele Programme gleichzeitig aus. Eine Portnummer von 0 bis 65535 gibt an, welches Programm es empfangen soll. Ist die IP-Adresse die Straßenadresse eines Bürogebäudes, dann ist der Port die Zimmernummer. Bekannte Dienste haben Standardports: 443 für HTTPS, 80 für einfaches HTTP, 53 für DNS, 5432 für die Datenbank PostgreSQL. Ein Serverprogramm *lauscht* auf einem Port und wartet auf Verbindungen; wenn Ihr Browser eine Website kontaktiert, verbindet er sich mit Port 443 unter der IP-Adresse der Website.

TCP und UDP: zwei Arten der Zustellung

Auf IP setzen zwei wichtige Transportprotokolle auf, die bestimmen, wie viel Sorgfalt auf die Zustellung verwendet wird. TCP (Transmission Control Protocol) bietet zwei Programmen einen zuverlässigen, geordneten Bytestrom. Bevor Daten fließen, bauen beide Seiten mit einem dreistufigen *Handshake* eine Verbindung auf. Jedes Paket ist nummeriert; der Empfänger bestätigt, was angekommen ist; Fehlendes wird erneut gesendet; alles wird dem Programm in der richtigen Reihenfolge übergeben. TCP drosselt außerdem das Tempo, wenn das Netzwerk überlastet ist. Es ist wie ein Einschreiben mit Sendungsverfolgung: verlässlich, aber mit etwas Mehraufwand.

UDP (User Datagram Protocol) sendet einfach einzelne Pakete, Datagramme genannt, ohne Handshake, ohne Bestätigungen, ohne erneutes Senden und ohne Reihenfolge. Es ist wie das Verschicken von Postkarten: billig und schnell, aber manche kommen vielleicht nicht an. Genau das ist richtig, wenn ein verspätetes Paket ohnehin nutzlos ist. In einem Videoanruf ist ein Tonfragment, das eine halbe Sekunde zu spät ankommt, schlimmer als eines, das nie ankommt; deshalb nutzen Anrufe, Livestreams und Onlinespiele UDP und kommen mit kleinen Lücken zurecht. Auch DNS-Abfragen, die aus winzigen Fragen und Antworten bestehen, nutzen traditionell UDP.

TCP und UDP im Vergleich

	TCP	UDP
Verbindungsaufbau	Ja: ein Handshake, bevor Daten fließen	Keiner: einfach senden
Zustellung	Garantiert: Verlorene Pakete werden erneut gesendet	Best Effort: Verluste werden nicht behoben
Reihenfolge	Bleibt erhalten	Nicht garantiert
Mehraufwand und Verzögerung	Höher	Niedriger
Typische Einsatzzwecke	Webseiten und APIs (HTTP/1.1 und HTTP/2), E-Mail, Dateiübertragung, Datenbankverbindungen	Sprach- und Videoanrufe, Livestreaming, Onlinespiele, DNS

VERTIEFUNG

Vertiefung: QUIC und HTTP/3

Die neueste Version des Web-Protokolls, HTTP/3 (2022 als RFC 9114 veröffentlicht), läuft nicht über TCP. Sie läuft über QUIC (RFC 9000, 2021), ein auf UDP aufsetzendes Transportprotokoll, das die zuverlässige Zustellung selbst neu implementiert und den Verbindungsaufbau mit dem Aufbau der Verschlüsselung kombiniert, was Roundtrips spart. Das zeigt, dass die Grenze zwischen „zuverlässigem TCP“ und „unzuverlässigem UDP“ davon handelt, was die Transportschicht garantiert, und dass sich Zuverlässigkeit auch auf einer höheren Schicht herstellen lässt, wenn das der Anwendung besser dient.

DNS: das Adressbuch des Internets

Menschen merken sich Namen wie `shop.example`; Netzwerke leiten nach IP-Adressen weiter. Das Domain Name System (DNS) übersetzt das eine in das andere. Es ist ein weltweites, verteiltes Verzeichnis: Kein einzelner Computer kennt alle Namen. Stattdessen ist die Zuständigkeit nach den Punkten im Namen aufgeteilt, von rechts nach links gelesen. Die **Root**-Server wissen, wer für jede Top-Level-Domain wie `.com` oder `.uk` zuständig ist; diese Top-Level-Domain-Server wissen, welche **autoritativen** Nameserver für jede darunter registrierte Domain zuständig sind; und die autoritativen Server halten die eigentlichen Einträge, etwa die IP-Adresse von `www.shop.example`. Die Root-Zone wird von 13 benannten Server-Identitäten bedient, die 12 unabhängige Organisationen über weit mehr als tausend Serverinstanzen in aller Welt betreiben.

Ihr Gerät geht diese Kette normalerweise nicht selbst ab. Es fragt einen **rekursiven Resolver**, meist betrieben von Ihrem Internetanbieter oder einem öffentlichen Dienst, der die Laufarbeit erledigt und Antworten für eine vom Domaininhaber festgelegte Dauer speichert (cacht), die TTL (Time to Live). Weil Antworten auf vielen Ebenen gecacht werden, sind die meisten Abfragen in wenigen Millisekunden beantwortet, ganz ohne die Root-Server zu erreichen. Dasselbe Caching erklärt, warum eine Änderung an einem DNS-Eintrag Minuten oder Stunden brauchen kann, bis sie überall sichtbar ist.

DNS-Auflösung Schritt für Schritt (nichts im Cache)

1. **Den Resolver fragen**
   Ihr Gerät fragt seinen konfigurierten rekursiven Resolver: Wie lautet die IP-Adresse von `www.shop.example`?

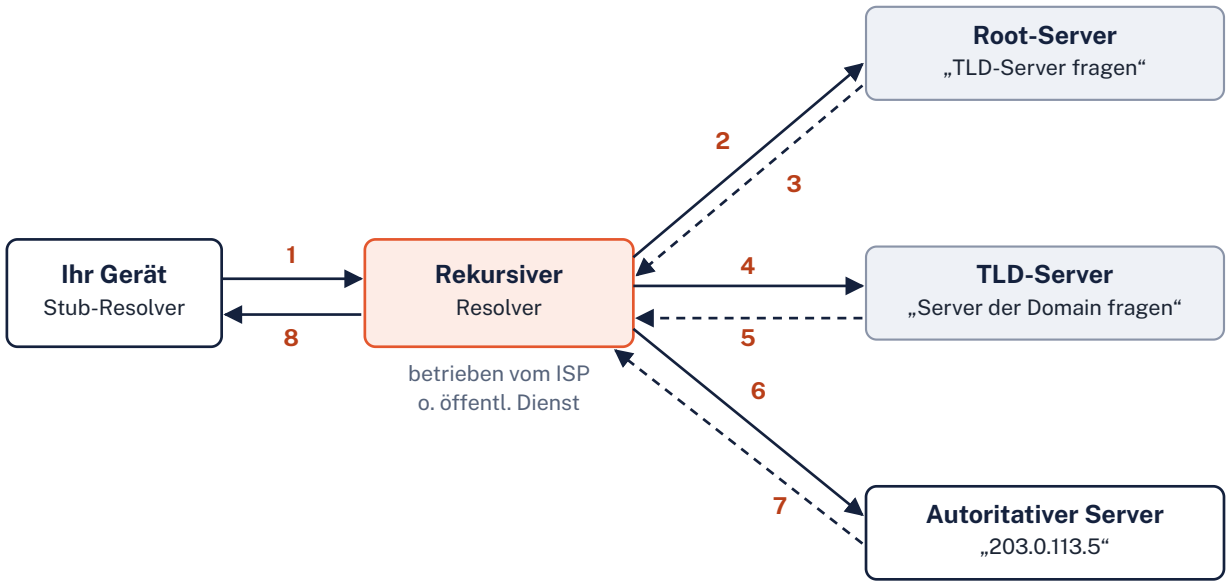
2. **Einen Root-Server fragen**
   Der Resolver fragt einen Root-Server, der die Antwort nicht kennt, aber mit den für die Top-Level-Domain zuständigen Servern antwortet.

3. **Den Top-Level-Domain-Server fragen**
   Der Resolver fragt einen dieser Server, der mit den autoritativen Nameservern für `shop.example` antwortet.

4. **Den autoritativen Server fragen**
   Der Resolver fragt den autoritativen Server, der mit dem Eintrag antwortet: zum Beispiel `203.0.113.5`, gültig für 300 Sekunden.

5. **Antworten und merken**
   Der Resolver gibt die Adresse an Ihr Gerät zurück, und beide behalten sie im Cache, bis die TTL abläuft, sodass die nächste Abfrage fast sofort erfolgt.

Wer beantwortet eine DNS-Frage?



Durchgezogene Pfeile sind Fragen, gestrichelte Pfeile Antworten; durch Caching überspringen die meisten realen Abfragen die Schritte 2 bis 7.

HTTP: Anfragen und Antworten

Sobald ein Programm ein anderes erreichen kann, brauchen beide eine gemeinsame Sprache für ihre Anliegen. Im Web ist diese Sprache HTTP (Hypertext Transfer Protocol). Es folgt einem einfachen Muster: Der Client sendet eine **Anfrage** (Request); der Server schickt genau eine **Antwort** (Response) zurück. Jede Anfrage nennt eine **Methode** (was zu tun ist), einen **Pfad** (woran), einige **Header** (Zusatzinformationen als Name-Wert-Paare) und manchmal einen **Body** (gesendete Daten). Jede Antwort enthält einen Statuscode, Header und meist einen Body: eine Webseite, ein Bild oder Daten in einem Format wie JSON (ein einfaches Textformat für strukturierte Daten, erklärt im Abschnitt über APIs).

Die gängigsten Methoden sind `GET` (etwas abrufen, ohne es zu ändern), `POST` (etwas Neues übermitteln, etwa eine Bestellung aufgeben), `PUT` (etwas unter einer bekannten Adresse ersetzen, etwa eine Lieferadresse aktualisieren) und `DELETE` (etwas entfernen). GET, PUT und DELETE sind per Definition *idempotent*: Dieselbe Anfrage zweimal zu senden hat dieselbe Wirkung wie einmal, weshalb sie nach einem Netzwerkfehler gefahrlos wiederholt werden können. POST ist es nicht; deshalb kann eine unbedachte Wiederholung eine Bestellung doppelt aufgeben. Header transportieren etwa den Inhaltstyp (`Content-Type: application/json`), die bevorzugte Sprache des Nutzers, Cookies (kleine Datenstücke, die eine Website vom Browser speichern und bei späteren Anfragen zurücksenden lässt) und Anmeldedaten sowie Caching-Anweisungen.

HTTP ist *zustandslos*: Jede Anfrage wird für sich verstanden, und der Server muss sich nicht an frühere erinnern. Angemeldete Sitzungen werden darauf aufgebaut, meist indem der Client bei jeder Anfrage ein Cookie oder Token mitschickt. Die Zustandslosigkeit ist ein Grund, warum das Web so gut skaliert: Jeder von vielen identischen Servern kann jede Anfrage bearbeiten.

```
POST /api/orders HTTP/1.1
Host: shop.example
Content-Type: application/json
Authorization: Bearer eyJhbGciOi...

{"product_id": 42, "quantity": 2}

HTTP/1.1 201 Created
Content-Type: application/json
Location: /api/orders/9917

{"order_id": 9917, "status": "received", "total_pence": 3000}
```

Ein realistischer HTTP-Austausch: eine Bestellung aufgeben, dann die Antwort des Servers

Statuscodes nach Klassen

Klasse	Bedeutung	Typische Beispiele
2xx	Erfolg	200 OK; 201 Created (etwas Neues wurde angelegt); 204 No Content
3xx	Umleitung: woanders nachsehen	301 Moved Permanently; 302 Found (vorübergehend verschoben); 304 Not Modified (Ihre zwischengespeicherte Kopie ist noch gültig)
4xx	Client-Fehler: Die Anfrage war falsch	400 Bad Request; 401 Unauthorized (nicht angemeldet); 403 Forbidden (angemeldet, aber nicht berechtigt); 404 Not Found; 429 Too Many Requests
5xx	Server-Fehler: Die Anfrage war in Ordnung, der Server hat versagt	500 Internal Server Error; 502 Bad Gateway; 503 Service Unavailable; 504 Gateway Timeout

Die Trennung zwischen 4xx und 5xx ist in der Praxis wichtig. Ein 4xx sagt: „Nicht einfach wiederholen; die Anfrage korrigieren.“ Ein 5xx sagt: „Das Problem liegt bei uns“; ein späterer erneuter Versuch kann klappen, und das Betriebsteam sollte alarmiert werden, wenn die 5xx-Raten steigen.

HTTPS und TLS: Vertraulichkeit und Identität

Einfaches HTTP sendet alles als lesbaren Text, sodass jeder auf dem Weg, etwa jemand im selben Café-WLAN, es lesen oder verändern könnte. HTTPS ist HTTP, das durch einen geschützten Kanal gesendet wird, den **TLS** herstellt (Transport Layer Security; die aktuelle Version, TLS 1.3, wurde 2018 als RFC 8446 veröffentlicht). TLS bietet drei Dinge: *Vertraulichkeit* (Lauscher sehen nur verschlüsselte Daten), *Integrität* (Manipulationen werden erkannt) und *Authentifizierung* (Sie sprechen wirklich mit `shop.example`, nicht mit einem Betrüger).

Die Authentifizierung beruht auf **Zertifikaten**. Das Zertifikat einer Website ist ein digitales Dokument, das sinngemäß sagt: „Dieser öffentliche Schlüssel gehört zu `shop.example`“, signiert von einer **Zertifizierungsstelle** (CA), der Ihr Browser oder Smartphone bereits vertraut. Es funktioniert wie ein Reisepass: ausgestellt von einer Stelle, der beide Seiten vertrauen, mit dem Inhaber abgeglichen und nur bis zum Ablaufdatum gültig. Während des TLS-Handshakes legt der Server sein Zertifikat vor, der Client prüft Signatur, Namen und Daten, und beide Seiten vereinbaren frische geheime Schlüssel, um den Rest der Kommunikation zu verschlüsseln. Schlägt eine Prüfung fehl, zeigt der Browser eine Warnung.

TLS schützt Daten *während der Übertragung* zwischen zwei Punkten. Es schützt keine Daten, sobald sie gespeichert sind, und es beweist nicht, dass die Website ehrlich ist, nur dass sie die ist, die sie zu sein behauptet. Heute ist HTTPS für den gesamten Webverkehr die Norm, nicht nur für Zahlungen, und Zertifikate werden oft kostenlos ausgestellt und automatisch erneuert.

Das Client-Server-Modell

Die meisten Systeme in diesem Kurs folgen dem Client-Server-Modell. Ein **Client** (ein Browser, eine mobile App, ein anderes Programm) beginnt die Kommunikation, indem er eine Anfrage sendet. Ein **Server** wartet auf Anfragen und beantwortet sie. Der Server hält, was geteilt und vertrauenswürdig sein muss: den Produktkatalog, die Bestellungen aller Kunden, die Regeln für Preise und Erstattungen. Der Client kümmert sich um den Bildschirm und die Eingaben des Nutzers.

In einem Restaurant sind die Gäste die Clients und die Küche ist der Server: Die Gäste geben Bestellungen auf; die Küche verfügt über die Zutaten und das Können und bedient viele Gäste gleichzeitig. Eine Folge davon ist, dass der Server dem Client nie blind vertrauen darf. Eine Smartphone-App kann von ihrem Nutzer verändert werden; deshalb müssen Preise, Berechtigungen und Lagerbestände auf dem Server geprüft werden, wie sorgfältig die App sie auch selbst prüft.

Ein Server ist oft zugleich ein Client: Der Webserver des Shops ist ein Client seiner Datenbank und des Zahlungsanbieters. Reale Systeme sind Ketten und Geflechte solcher Beziehungen, wie das Sequenzdiagramm unten zeigt.

Latenz und Bandbreite

Zwei Größen beschreiben, wie schnell sich ein Netzwerk anfühlt, und sie werden oft verwechselt. Bandbreite ist die Datenmenge, die eine Verbindung pro Sekunde übertragen kann, etwa 100 Megabit pro Sekunde: die Breite des Rohrs. Latenz ist die Zeit, die ein Datenstück von einem Ende zum anderen braucht, meist gemessen als **Round-Trip-Time** (RTT) in Millisekunden: die Länge des Rohrs. Ein Autobahnvergleich hilft: Zusätzliche Fahrspuren (Bandbreite) lassen mehr Autos pro Stunde durch, verkürzen aber die Fahrt keines einzelnen Autos (Latenz).

Die Latenz hat eine Untergrenze, die die Physik setzt. Licht in Glasfasern bewegt sich mit rund 200.000 km pro Sekunde, etwa zwei Dritteln seiner Geschwindigkeit im Vakuum. London und Sydney liegen auf geradem Weg über den Globus etwa 17.000 km auseinander; selbst eine perfekte Glasfaser auf dieser Linie bräuchte also etwa 85 ms für eine Richtung und 170 ms für einen Roundtrip. Echte Kabel folgen nicht dem kürzesten Weg, und Router fügen Verzögerung hinzu, sodass gemessene Roundtrips typischerweise bei etwa 250 ms liegen. Kein Upgrade kann das beseitigen. Braucht das Laden einer Seite aus Sydney zehn aufeinanderfolgende Roundtrips zu einem Server in London, wartet der Nutzer zweieinhalb Sekunden, wie schnell sein Breitbandanschluss auch ist.

AUSGEARBEITETES BEISPIEL

Ausgearbeitetes Beispiel: ein Server in London, ein Kunde in Sydney

Ein Shop hostet alles in London. Ein Kunde in Sydney öffnet eine Produktseite über eine neue HTTPS-Verbindung. Nehmen Sie einen Roundtrip von 250 ms an und lassen Sie die Verarbeitungszeit auf dem Server außer Acht.

DNS-Abfrage (nicht im Cache)	Der Resolver braucht eventuell mehrere Abfragen; sagen wir 1 Roundtrip zu nahen Servern: etwa 50 ms
TCP-Handshake	1 Roundtrip nach London: 250 ms
TLS-1.3-Handshake	1 Roundtrip: 250 ms
Die HTML-Seite anfordern	1 Roundtrip: 250 ms
CSS und Skripte anfordern, die die Seite braucht	Mindestens 1 weiterer Roundtrip: 250 ms
Gesamt, bevor etwas Nützliches erscheint	Etwa 1050 ms, wobei die Bandbreite fast keine Rolle spielt
Derselbe Kunde, Inhalte aus Sydney ausgeliefert	Roundtrip von vielleicht 10 ms: Dieselben fünf Schritte dauern rund 50 ms

Bei kleinen Seiten dominiert die Latenz, nicht die Bandbreite. Den Breitbandanschluss des Kunden von 50 auf 500 Megabit pro Sekunde aufzurüsten würde am Ergebnis kaum etwas ändern.

Architekten verringern die Latenz, indem sie Roundtrips einsparen (Verbindungen wiederverwenden, Anfragen bündeln) und Inhalte näher an die Nutzer bringen, etwa mit Content Delivery Networks, behandelt in Server und die Cloud.

ENTSCHEIDUNGSHILFE

Entscheidungshilfe: Liegt ein langsames Netzwerkerlebnis an der Latenz oder an der Bandbreite?

Eine grobe erste Diagnose, wenn Nutzer klagen, dass etwas über das Netzwerk langsam ist.

Q1 **Sind auch kleine Anfragen (ein kurzer API-Aufruf, eine winzige Seite) langsam?**

Ja Wahrscheinliche Ursache ist die Latenz. Prüfen Sie die Entfernung zum Server und wie viele aufeinanderfolgende Roundtrips die Seite oder App macht.

Nein Weiter zur nächsten Frage.

Q2 **Sind nur große Übertragungen (Videos, große Downloads, umfangreiche Berichte) langsam?**

Ja Wahrscheinlich begrenzt die Bandbreite, auf der Verbindung des Nutzers oder der des Servers. Erwägen Sie Komprimierung, kleinere Dateien oder ein Content Delivery Network.

Nein Weiter zur nächsten Frage.

Q3 **Ist die Verzögerung von jedem Standort aus gleich?**

Ja Vielleicht ist das Netzwerk gar nicht das Problem: Sehen Sie sich Serververarbeitung, Datenbankabfragen und Algorithmen an (siehe den Abschnitt über Daten und Algorithmen).

Nein Standortabhängige Verzögerung deutet auf Latenz und Routing hin: Bedienen Sie Nutzer von näher an ihrem Standort aus.

Was passiert, wenn Sie eine Webadresse eingeben

All diese Teile wirken jedes Mal zusammen, wenn Sie eine Website besuchen. Hier ist der Ablauf, wenn Sie `https://shop.example/products/42` eingeben und die Eingabetaste drücken, vereinfacht, aber zutreffend.

Von der Eingabetaste zur Seite

1. **Die Adresse zerlegen**
   Der Browser zerlegt die URL in das Schema (`https` , also Port 443 und TLS), den Hostnamen (`shop.example`) und den Pfad (`/products/42`).

2. **Die IP-Adresse finden**
   Der Browser prüft seinen eigenen Cache, dann den des Betriebssystems, und fragt dann den rekursiven Resolver, der wie oben beschrieben das DNS befragt und etwa `203.0.113.5` zurückgibt.

3. **Eine TCP-Verbindung öffnen**
   Browser und Server tauschen die drei Handshake-Nachrichten aus, um eine zuverlässige Verbindung zu Port 443 aufzubauen.

4. **TLS aufbauen**
   Der Server legt sein Zertifikat vor; der Browser prüft, ob es für `shop.example` gültig und von einer vertrauenswürdigen Stelle signiert ist; beide Seiten vereinbaren Schlüssel für die Verschlüsselung.

5. **Die HTTP-Anfrage senden**
   Der Browser sendet `GET /products/42` mit Headern wie den akzeptierten Sprachen und etwaigen Cookies für die Website.

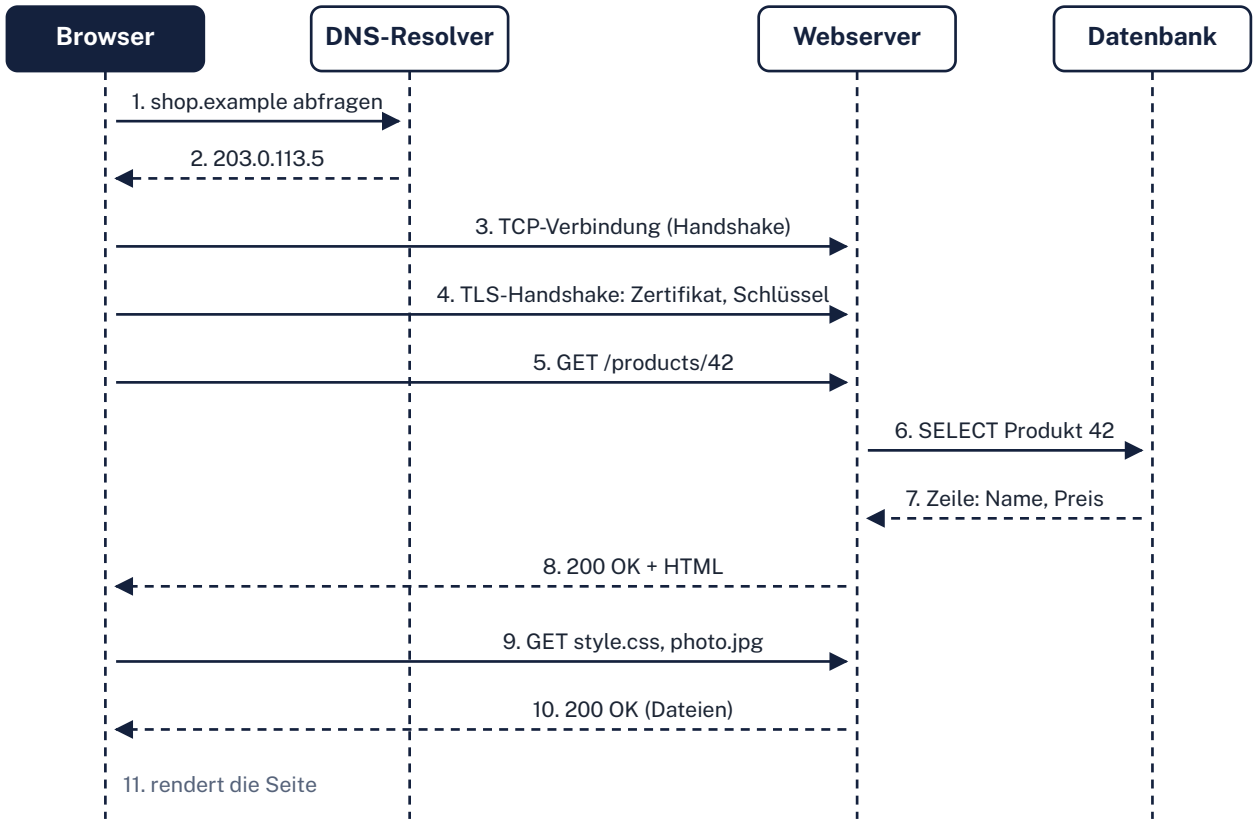
6. **Der Server erledigt seine Arbeit**
   Die Serversoftware empfängt die Anfrage und führt den Code des Shops aus, der eine Datenbank nach Produkt 42 abfragt und eine HTML-Seite erzeugt.

7. **Die Antwort empfangen**
   Der Server antwortet mit `200 OK`, mit dem HTML im Body und Headern wie `Content-Type: text/html` und Caching-Regeln.

8. **Den Rest abrufen**
   Das HTML verweist auf Stylesheets, Skripte und Bilder; der Browser fordert auch diese an, oft viele gleichzeitig über dieselbe Verbindung.

9. **Rendern**
   Der Browser ordnet die Seite an und zeichnet sie auf den Bildschirm; Skripte können danach weitere Anfragen stellen, etwa um Bewertungen abzurufen.

Sequenzdiagramm: eine Produktseite laden



Die Zeit läuft nach unten; durchgezogene Pfeile sind Anfragen, gestrichelte Pfeile Antworten.

ACHTUNG

Verbreitete Irrtümer

- **„Das Internet ist schnell, also spielt Entfernung keine Rolle."** Die Bandbreite ist enorm gewachsen; die Lichtgeschwindigkeit nicht. Jeder Roundtrip zu einem entfernten Server kostet echte Zeit.
- **„HTTPS bedeutet, dass die Website sicher ist."** Es bedeutet, dass die Verbindung vertraulich ist und die Website die ist, die sie zu sein behauptet. Auch eine Betrugsseite kann ein gültiges Zertifikat haben.
- **„Ein 404 bedeutet, dass der Server ausgefallen ist."** Ein 404 ist der korrekt arbeitende Server, der meldet, dass das Angefragte nicht existiert. Ein ausgefallener Server antwortet meist gar nicht oder liefert einen 5xx von etwas, das vor ihm sitzt.
- **„DNS-Änderungen wirken sofort."** Gecachte Antworten leben, bis ihre TTL abläuft, daher verbreiten sich Änderungen nach und nach.

Warum das für die Architektur wichtig ist

Sobald ein System auf mehr als einen Computer verteilt ist, wird jede Interaktion zwischen seinen Teilen zu einem Netzwerkaufruf, mit Latenz, der Möglichkeit eines Ausfalls und dem Bedarf an Sicherheit. Ein Funktionsaufruf innerhalb eines Programms dauert Nanosekunden und geht nie verloren; ein Aufruf über ein Netzwerk dauert Millisekunden, manchmal weit länger, und kehrt manchmal nie zurück. Viele der schwierigsten Probleme der Architektur folgen aus diesem Unterschied; deshalb beginnt Kapitel 6 mit den Fallacies of Distributed Computing, etwa dem Glauben, die Latenz sei null.

Die hier behandelten Protokolle tauchen auch als Bausteine wieder auf: HTTP liegt den Web-APIs im nächsten Teil dieses Kapitels und den Kommunikationsstilen in Kapitel 6 zugrunde, während TLS und Zertifikate in sichere Kommunikation und Zero Trust wiederkehren. Server nahe an den Nutzern zu platzieren ist Teil der Multi-Region- und Edge-Architektur.

PRÜFEN SIE IHR VERSTÄNDNIS

▼ Warum braucht ein Paket neben einer IP-Adresse auch eine Portnummer?

Die IP-Adresse bringt es zum richtigen Computer, aber dieser Computer führt viele Programme aus; der Port gibt an, welches Programm es empfangen soll, etwa der Webserver, der auf 443 lauscht.

▼ Was teilt ein 3xx-Statuscode dem Client mit?

Dass das Angefragte woanders liegt oder unverändert ist: Zum Beispiel bedeutet 301, dass es dauerhaft verschoben wurde, und 304, dass die zwischengespeicherte Kopie des Clients noch gültig ist.

▼ Was ist der Unterschied zwischen einer 401- und einer 503-Antwort?

401 ist ein Client-Fehler: Der Anfrage fehlen gültige Anmeldedaten. 503 ist ein Server-Fehler: Der Dienst kann die Anfrage vorübergehend nicht bearbeiten.

▼ Warum kann kein Netzwerk-Upgrade einen Roundtrip zwischen London und Sydney auf 10 ms senken?

Licht legt in Glasfasern etwa 200.000 km pro Sekunde zurück, daher dauert der Roundtrip von rund 34.000 km mindestens etwa 170 ms.

QUELLEN

- MDN: An overview of HTTP: https://developer.mozilla.org/en-US/docs/Web/HTTP/Guides/Overview
- MDN: HTTP response status codes: https://developer.mozilla.org/en-US/docs/Web/HTTP/Reference/Status
- RFC 9110: HTTP Semantics: https://www.rfc-editor.org/rfc/rfc9110
- RFC 8446: The Transport Layer Security (TLS) Protocol Version 1.3: https://www.rfc-editor.org/rfc/rfc8446
- Root Server Technical Operations Association: https://root-servers.org/

ABSCHNITT 0.5 VON 0.8 · EINSTEIGER · 18 MIN. LESEZEIT

Dateien und Datenbanken: relationale und nicht relationale Speicherung, Abfragen und Transaktionen

DIE IDEE IN KÜRZE

Eine Datenbank ist wie eine Bibliothek mit einer sorgfältigen Bibliothekarin. Statt dass jedes Programm Papiere in seine eigene Schublade stopft, kommen alle Daten an einen Ort, der einen Katalog (einen Index) führt, prüft, dass jeder Datensatz den Regeln folgt, dafür sorgt, dass nicht zwei Personen gleichzeitig dasselbe ändern, und nie eine Änderung verliert, deren Speicherung er zugesagt hat. Die meisten Datenbanken speichern Daten in Tabellen, ähnlich wie Tabellenkalkulationen mit strengen Regeln, und man stellt ihnen Fragen in einer Sprache namens SQL.

WAS SIE WISSEN MÜSSEN

- Dateien eignen sich für ganze Dokumente wie Bilder und Logs; Datenbanken eignen sich für strukturierte Daten, die viele Nutzer gleichzeitig lesen und ändern, weil das DBMS Nebenläufigkeit, Regeln, schnelles Nachschlagen und die Wiederherstellung nach Abstürzen übernimmt.
- Relationale Datenbanken speichern Daten in Tabellen aus Zeilen und Spalten; ein Primärschlüssel identifiziert jede Zeile, und ein Fremdschlüssel verknüpft eine Zeile mit einer Zeile in einer anderen Tabelle.
- SQL ist deklarativ: Sie geben das gewünschte Ergebnis an, etwa ein gefiltertes SELECT oder einen JOIN über mehrere Tabellen, und der Query Planner der Datenbank entscheidet, wie es ermittelt wird.
- Mit einem Index findet die Datenbank Zeilen in O(log n) Schritten, statt die ganze Tabelle zu durchsuchen, um den Preis von zusätzlichem Speicherplatz und langsameren Schreibvorgängen.
- Eine Transaktion lässt mehrere Änderungen gemeinsam gelingen oder scheitern; ACID (Atomarität, Konsistenz, Isolation, Dauerhaftigkeit) beschreibt ihre Garantien, etwa bei einer Banküberweisung, die nie nur halb stattfinden kann.
- NoSQL-Familien (Dokument, Key-Value, Wide-Column und Graph) passen zu bestimmten Datenformen und Zugriffsmustern; viele Systeme nutzen eine relationale Datenbank plus eine oder mehrere davon.
- Backups sind erst bewiesen, wenn man sie wiederhergestellt hat, Replikate schützen nicht vor Fehlern, und Managed Databases übernehmen den Betrieb, aber nicht den Entwurf von Schema und Abfragen.

Warum Programme einen Ort für ihre Daten brauchen

Ein laufendes Programm hält seine Arbeitsdaten im Arbeitsspeicher (RAM), der schnell ist, aber alles vergisst, sobald das Programm endet oder die Maschine neu startet (siehe Was Software ist). Alles, was einen Neustart überstehen muss, etwa ein Kundenkonto, eine Bestellung oder ein Krankenhaustermin, muss auf einen Speicher geschrieben werden, der seinen Inhalt ohne Strom behält: eine Festplatte oder SSD. Daten, die auf diese Weise erhalten bleiben, nennt man *persistent*, und die Entscheidung, wie man sie persistiert, gehört zu den ersten und langlebigsten Entscheidungen in jedem System.

Die einfachste Art, Daten zu persistieren, ist eine Datei: eine benannte Folge von Bytes, die das Betriebssystem auf der Festplatte ablegt. Dateien eignen sich hervorragend für alles, was von Natur aus ein ganzes Dokument ist: Bilder, PDFs, Videos, Konfiguration, Logs, exportierte Berichte. Systeme legen sie oft in einem *Object Storage* in der Cloud ab (einem Dienst, der Dateien unter einem Namen speichert) und speichern anderswo nur einen Verweis.

Mühsam werden Dateien, wenn viele Nutzer gleichzeitig kleine Teile strukturierter Daten lesen und ändern. Stellen Sie sich einen Shop vor, der alle Bestellungen in einer einzigen Tabellenkalkulationsdatei führt. Zwei Kunden schließen im selben Moment den Bezahlvorgang ab: Wessen Änderung gewinnt? Der Server stürzt mitten beim Speichern ab: Ist die Datei nun halb alt, halb neu? Ein Supportmitarbeiter will die Bestellungen eines Kunden sehen: Muss das Programm die ganze Datei lesen? Jemand tippt eine Kundennummer ein, die es nicht gibt: Wer hält ihn auf? Jedes Programm, das die Datei nutzt, müsste jedes dieser Probleme lösen, und zwar identisch.

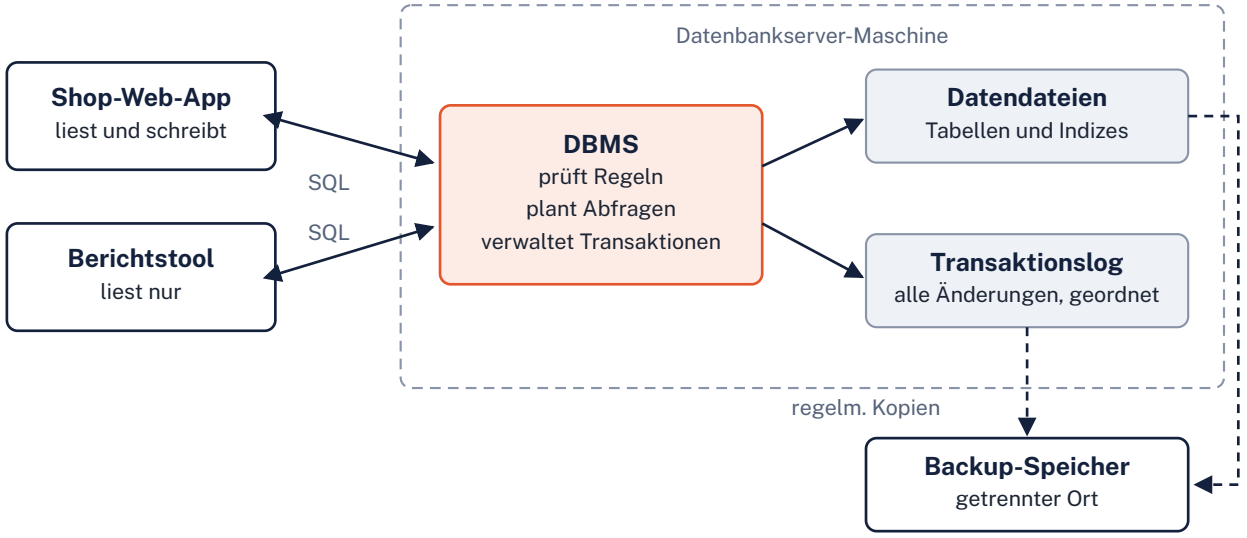
Was eine Datenbank ist

Eine Datenbank ist eine organisierte Sammlung von Daten, und das Programm, das sie verwaltet, ist ein Datenbankmanagementsystem (DBMS), etwa PostgreSQL, MySQL, Microsoft SQL Server, Oracle Database oder SQLite. Oft sagt man für beides „die Datenbank“. Das DBMS löst die Probleme mit Dateien ein für alle Mal, für alle: Es lässt viele Nutzer gleichzeitig arbeiten, ohne dass sie sich gegenseitig in die Quere kommen, findet Daten schnell, weist Daten zurück, die gegen die Regeln verstoßen, und kehrt nach einem Absturz in einen korrekten Zustand zurück.

Die Analogie ist eine Bibliothek. Ein Stapel Papiere auf einem Schreibtisch funktioniert, solange Sie ihn allein nutzen. Eine Bibliothek hat einen Katalog, Regeln dafür, wo was steht, eine Aufzeichnung, wer was ausgeliehen hat, und eine Bibliothekarin, die nicht zulässt, dass zwei Personen dasselbe Exemplar nehmen. Anwendungen fassen die Regale nicht an; sie fragen die Bibliothekarin.

In den meisten Geschäftssystemen läuft das DBMS als eigener Serverprozess, oft auf einer eigenen Maschine, und Anwendungen sprechen über das Netzwerk mit ihm (siehe Wie Computer kommunizieren). SQLite ist eine bekannte Ausnahme: Es ist eine Bibliothek, die innerhalb der Anwendung läuft und die ganze Datenbank in einer einzigen Datei speichert, weshalb man es in Smartphones, Browsern und kleinen Werkzeugen findet.

Wo die Daten tatsächlich liegen



Anwendungen schreiben nie direkt in die Dateien der Datenbank: Sie senden Anfragen an das DBMS, das Datendateien und ein Log schreibt, und ein separater Prozess kopiert beides in einen Backup-Speicher.

Relationale Datenbanken: Tabellen, Schlüssel und Beziehungen

Die am weitesten verbreitete Art von Datenbank ist die relationale Datenbank, die auf einem Modell beruht, das Edgar F. Codd von IBM 1970 veröffentlichte. Die Daten liegen in **Tabellen**. Jede Tabelle beschreibt eine Art von Ding (Kunden, Bestellungen, Produkte). Jede **Zeile** ist eine Instanz dieses Dings (ein Kunde), und jede **Spalte** ist ein Fakt darüber (Name, E-Mail-Adresse, Beitrittsdatum), mit einem deklarierten Typ wie Ganzzahl, Text, Datum oder Dezimalzahl.

Jede Tabelle hat einen Primärschlüssel: eine Spalte (oder Kombination von Spalten), deren Wert für jede Zeile eindeutig und nie leer ist, etwa `customer_id`. Ein Fremdschlüssel ist eine Spalte in einer Tabelle, die den Primärschlüssel einer Zeile in einer anderen Tabelle enthält. Eine Bestellung hat eine Spalte `customer_id`, die angibt, welcher Kunde sie aufgegeben hat. Wird sie als Fremdschlüssel deklariert, weist das DBMS eine Bestellung für Kunde 999 zurück, wenn es keinen Kunden 999 gibt: Das nennt man *referenzielle Integrität*.

Die Gesamtheit aus Tabellen, Spalten, Typen, Schlüsseln und Regeln ist das Schema. Das Schema ist ein Vertrag: Jedes Programm, das die Datenbank nutzt, kann sich darauf verlassen, dass eine Bestellung immer einen gültigen Kunden und einen Preis hat, der eine Zahl ist. Ein Schema später zu ändern, wenn Millionen von Zeilen und mehrere Programme davon abhängen, ist möglich, erfordert aber Sorgfalt; deshalb hat die Schemaevolution in Kapitel 7 einen eigenen Abschnitt.

Relationale Entwürfe versuchen, jeden Fakt nur einmal zu speichern. Die E-Mail-Adresse eines Kunden steht in einer Zeile der Kundentabelle und wird nicht in jede Bestellung kopiert. Das nennt man *Normalisierung*. Eine Aktualisierung erfolgt so an einer Stelle und kann keine widersprüchlichen Kopien hinterlassen. Der Preis für diese Ordnung ist, dass man zur Beantwortung einer Frage oft Tabellen kombinieren muss, und genau dafür gibt es Joins.

Fragen stellen mit SQL

SQL (Structured Query Language, ausgesprochen „S-Q-L“ oder „Sequel“) ist die Sprache, mit der man mit relationalen Datenbanken arbeitet. Sie wurde in den 1970er-Jahren bei IBM entwickelt und Ende der 1980er-Jahre zu einem ANSI- und ISO-Standard; jedes große relationale DBMS spricht sie, jeweils mit kleinen Dialektunterschieden. SQL ist *deklarativ*: Sie beschreiben das gewünschte Ergebnis, nicht die Schritte, um es zu finden. Der *Query Planner* des DBMS wählt die Schritte, zum Beispiel ob eine ganze Tabelle gelesen oder ein Index genutzt wird.

Ein Join kombiniert Zeilen aus zwei oder mehr Tabellen anhand übereinstimmender Spalten, meist ein Fremdschlüssel gegen den Primärschlüssel, auf den er verweist. Die zweite Abfrage unten listet jede Position der Bestellungen eines Kunden zusammen mit dem Produktnamen auf und greift dabei auf drei Tabellen gleichzeitig zu.

```
-- One customer's orders since 1 October, newest first
SELECT order_id, status, placed_at
FROM orders
WHERE customer_id = 42
  AND placed_at ⩾ '2026-10-01'
ORDER BY placed_at DESC;

-- Each order line with the product's name (a join)
SELECT o.order_id, p.name, i.quantity, i.unit_price
FROM orders AS o
JOIN order_items AS i ON i.order_id = o.order_id
JOIN products AS p ON p.product_id = i.product_id
WHERE o.customer_id = 42;
```

Zwei Abfragen auf das Shop-Schema, das weiter unten in diesem Abschnitt gezeigt wird: ein gefiltertes SELECT und ein JOIN über drei Tabellen.

`INSERT`, `UPDATE` und `DELETE` ändern Daten, und `CREATE TABLE` definiert das Schema. Anwendungscode sendet SQL oft über einen *objektrelationalen Mapper* (ORM), eine Bibliothek, die Zeilen in Objekte umwandelt. ORMs sparen Tipparbeit, verbergen aber das tatsächlich gesendete SQL; ein klassisches Performance-Problem ist ein ORM, das unbemerkt eine Abfrage pro Zeile statt eines einzigen Joins absetzt.

Indizes: das Register des Buchs, nicht das ganze Buch

Ohne Hilfe kann die Datenbank alle Bestellungen von Kunde 42 nur finden, indem sie jede Zeile der Bestelltabelle liest und prüft: ein Full Table Scan, der in der Notation aus Daten und Algorithmen O(n) ist. Ein Datenbankindex ist eine zusätzliche, sortierte Datenstruktur, die Werte einer Spalte den Zeilen zuordnet, die sie enthalten, so wie das Register hinten in einem Buch Wörter den Seitenzahlen zuordnet. Die meisten relationalen Datenbanken verwenden standardmäßig einen *B-Baum*, einen breiten, flachen, stets balancierten Baum, der einen Wert in O(log n) Schritten findet.

Primärschlüssel werden automatisch indiziert. Weitere Indizes legt man gezielt für Spalten an, nach denen Abfragen filtern, verknüpfen oder sortieren: `CREATE INDEX idx_orders_customer ON orders (customer_id);`. Indizes sind nicht kostenlos. Jedes Einfügen, Aktualisieren oder Löschen muss auch jeden Index der Tabelle aktualisieren, und jeder Index belegt Festplattenplatz und Arbeitsspeicher. Eine Tabelle mit fünfzehn Indizes liest schnell und schreibt langsam. Die Wahl der Indizes ist eine Abwägung, die sich an den Abfragen orientiert, die das System tatsächlich ausführt, und damit ein frühes Beispiel für die Abwägung zwischen Platz und Zeit, die sich durch die gesamte Architektur zieht.

AUSGEARBEITETES BEISPIEL

Ausgearbeitetes Beispiel: die Bestellungen eines Kunden finden

Ein Shop hat 10 Millionen Zeilen in seiner Bestelltabelle. Die Kundenservice-Ansicht fragt alle Bestellungen eines Kunden ab, etwa 20 Zeilen.

Ohne Index	Full Table Scan: Die Datenbank liest und prüft alle 10.000.000 Zeilen, um die 20 passenden zu finden.
Mit einem B-Baum-Index auf customer_id	Jede Indexseite enthält Hunderte von Schlüsseln, daher ist der Baum nur drei oder vier Ebenen tief. Die Datenbank liest eine Handvoll Indexseiten und holt dann die 20 passenden Zeilen.
Wachstum	Wächst die Tabelle auf 100 Millionen Zeilen, leistet der Scan die zehnfache Arbeit; die Indexsuche kommt höchstens um eine Ebene dazu.
Kosten an anderer Stelle	Jede neue Bestellung schreibt nun in die Tabelle und in den Index. Der Festplattenbedarf wächst um die Größe des Index.

Das ist O(n) gegenüber O(log n) aus Abschnitt 0.2: oft der Unterschied zwischen einer Antwort in Sekunden und einer in Millisekunden. Viele Beschwerden über langsame Abfragen in jungen Systemen behebt der richtige Index, nicht größere Hardware.

Transaktionen und ACID: die Banküberweisung

Um 100 Pfund von Alices Konto auf Bobs Konto zu überweisen, braucht es zwei Änderungen: 100 bei Alice abziehen, 100 bei Bob gutschreiben. Stürzt der Server nach der ersten Änderung und vor der zweiten ab, sind 100 Pfund verschwunden. Laufen zwei Überweisungen von Alice gleichzeitig, sehen womöglich beide einen Kontostand von 150 und beide gelingen, sodass ihr Konto überzogen ist. Eine Transaktion fasst mehrere Operationen zu einer Arbeitseinheit zusammen, die entweder vollständig abgeschlossen wird (*Commit*) oder gar keine Wirkung hat (*Rollback*).

Die Garantien einer Transaktion werden als ACID zusammengefasst, ein Akronym, das Theo Härder und Andreas Reuter 1983 prägten, aufbauend auf früheren Arbeiten von Jim Gray:

Eigenschaft	Was sie verspricht	Bei der Banküberweisung
Atomarität	Alle Änderungen der Transaktion finden statt, oder keine.	Alice wird nie belastet, ohne dass Bob die Gutschrift erhält.
Konsistenz	Eine Transaktion überführt die Datenbank von einem gültigen Zustand in einen anderen; Regeln wie Schlüssel und Prüfungen bleiben nie verletzt.	Eine Regel, dass Kontostände nicht unter null fallen dürfen, weist eine Überweisung zurück, die Alice sich nicht leisten kann.
Isolation	Gleichzeitig laufende Transaktionen sehen die halb fertige Arbeit der anderen nicht; das Ergebnis ist so, als liefen sie nacheinander (in einem Maß, das die Isolationsebene festlegt).	Zwei gleichzeitige Überweisungen von Alice können sich nicht beide auf denselben alten Kontostand stützen.
Dauerhaftigkeit	Sobald die Datenbank einen Commit bestätigt, übersteht die Änderung einen Absturz oder Stromausfall.	Wenn die App „Überweisung abgeschlossen“ meldet, ist das Geld wirklich bewegt, weil die Änderung bereits im Log auf der Festplatte steht.

```
import sqlite3

conn = sqlite3.connect("bank.db")
# The accounts table was created with CHECK (balance ⩾ 0)

def transfer(from_id, to_id, amount):
    try:
        with conn:  # one transaction: commit on success, rollback on error
            debit = conn.execute(
                "UPDATE accounts SET balance = balance - ? WHERE id = ?",
                (amount, from_id))
            credit = conn.execute(
                "UPDATE accounts SET balance = balance + ? WHERE id = ?",
                (amount, to_id))
            if debit.rowcount ≠ 1 or credit.rowcount ≠ 1:
                raise ValueError("No such account")  # forces a rollback
        return "Transfer complete"
    except (sqlite3.Error, ValueError):
        return "Transfer refused: no account was changed"
```

Eine Überweisung mit dem eingebauten Python-Modul sqlite3. Der with-Block führt einen Commit aus, wenn beide Aktualisierungen gelingen, und ein Rollback, wenn eine davon scheitert oder ein Konto nicht existiert.

Beachten Sie die Prüfung von `rowcount` (der Anzahl der Zeilen, die jede Aktualisierung geändert hat): Eine Aktualisierung, die auf kein Konto passt, ist für die Datenbank kein Fehler, daher muss der Code sie selbst als Fehler behandeln. Zwei weitere Details sind über dieses

Beispiel hinaus wichtig. Die Platzhalter `?` übergeben Werte getrennt vom SQL-Text; das ist die Standardabwehr gegen *SQL-Injection*, einen Angriff, bei dem Benutzereingaben als Code in eine Abfrage eingeschleust werden (Kapitel 8 kommt darauf zurück). Und das Versprechen der Dauerhaftigkeit wird durch ein *Write-Ahead-Log* eingehalten: Das DBMS schreibt jede Änderung in ein nur erweiterbares Log auf der Festplatte, bevor es den Commit bestätigt, sodass es nach einem Absturz Arbeit wiederholen oder rückgängig machen kann, um einen korrekten Zustand zu erreichen.

Isolation ist die subtilste Eigenschaft. Vollständige Isolation (*serialisierbar* genannt) ist sicher, begrenzt aber, wie viel Arbeit parallel laufen kann; daher bieten Datenbanken schwächere *Isolationsebenen*, die etwas Schutz gegen Geschwindigkeit tauschen, und viele nutzen standardmäßig eine schwächere Ebene. Welche Anomalien jede Ebene zulässt, behandelt Transaktionen, ACID und Isolationsebenen. Vorerst ist der Kerngedanke, dass eine einzelne Datenbank innerhalb einer Transaktion starke, gut verstandene Garantien gibt und dass diese Garantien viel schwerer zu erreichen sind, sobald Daten über mehrere Datenbanken oder Services verteilt sind.

NoSQL: andere Formen von Daten

Relationale Datenbanken sind nicht die einzige Option. NoSQL ist ein Sammelbegriff, populär geworden durch ein Meetup 2009 in San Francisco, für Datenbanken, die nicht das relationale Modell verwenden. Sie entstanden, um sehr große, über viele Maschinen verteilte Datenmengen, flexible Datenformen und Zugriffsmuster zu bewältigen, die Tabellen nur umständlich bedienen. Der Name sagt, was sie nicht sind, und umfasst daher recht unterschiedliche Familien.

Die vier wichtigsten NoSQL-Familien

Klasse	Wie sie Daten speichert	Gut geeignet für	Beispiele
Dokumentendatenbank	Jeder Datensatz ist ein in sich geschlossenes Dokument, meist JSON-ähnlich, das Listen und Unterobjekte verschachteln kann. Dokumente in einer Collection müssen nicht genau dieselben Felder haben.	Daten, die als Ganzes gelesen und geschrieben werden: ein Produktkatalog mit unterschiedlichen Attributen, ein Benutzerprofil, Inhalte für eine Webseite.	MongoDB, Couchbase, Firestore
Key-Value-Store	Ein riesiges Wörterbuch: Ein eindeutiger Schlüssel verweist auf einen Wert, in den die Datenbank nicht hineinschaut. Nachgeschlagen wird nur über den Schlüssel.	Sehr schnelle einfache Abfragen: Sessions, Warenkörbe, Caches, Feature-Flags.	Redis, Amazon DynamoDB, etcd
Wide-Column-Store	Zeilen, die durch einen Schlüssel identifiziert werden, jede mit vielen Spalten, gruppiert und über viele Maschinen verteilt (partitioniert). Tabellen werden um die Abfragen herum entworfen, die ausgeführt werden sollen.	Riesige Mengen an Schreibvorgängen, etwa Sensormesswerte, Ereignislogs und Nachrichtenverläufe, über viele Server und Rechenzentren hinweg.	Apache Cassandra, Apache HBase, Google Bigtable
Graphdatenbank	Knoten (Dinge) und Kanten (Beziehungen), beide mit Eigenschaften, wie in den Graphen aus Abschnitt 0.2.	Fragen nach Verbindungen: Freunde von Freunden, Betrugsringe, Routenplanung, Empfehlungen.	Neo4j, Amazon Neptune

Fachleute sind sich über den Standard uneinig. Ein Lager sagt, man solle mit einer relationalen Datenbank beginnen, sofern es keinen klaren Grund dagegen gibt: SQL ist weithin bekannt, Transaktionen und Joins sind mächtig, und moderne relationale Systeme speichern auch JSON-Dokumente gut. Das andere Lager sagt, man solle von Anfang an den Speicher wählen, der zum Zugriffsmuster passt, weil ein Dokumenten- oder Key-Value-Store für die richtige Last einfacher sein und günstiger skalieren kann. Beide sind sich einig, dass viele reale Systeme mehr als eine Art nutzen, jede dort, wo sie passt; das nennt man *polyglotte Persistenz*. Der Abschnitt Eine Datenbank wählen in Kapitel 7 bietet einen strukturierten Weg zur Entscheidung.

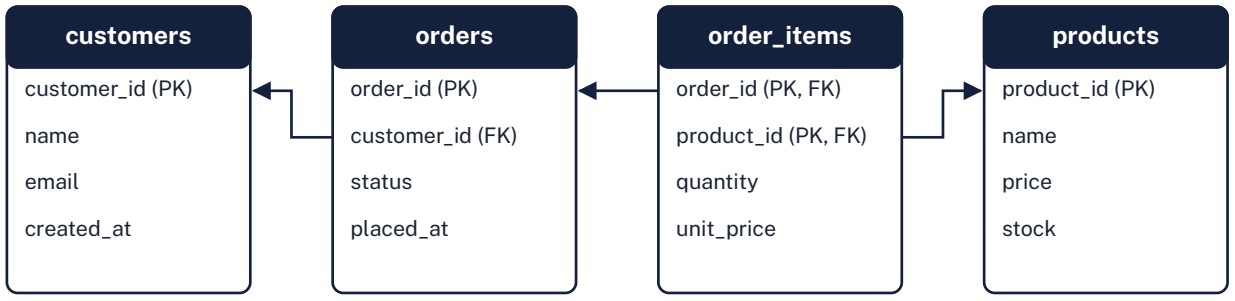
ACHTUNG

„Schemalos“ heißt nicht „kein Schema“

Eine Dokumentendatenbank akzeptiert heute ein Kundendokument mit `email` und morgen eines mit `emailAddress`. Die Daten haben trotzdem eine Form; sie wird jetzt von jedem Programm durchgesetzt, das sie liest, oder eben nicht, statt von der Datenbank. Teams, die bei NoSQL-Stores das Nachdenken über das Schema auslassen, entdecken es meist später wieder, in Form von Code voller Sonderfälle für alte Dokumente.

Ausgearbeitetes Beispiel: das Schema eines Onlineshops

Nehmen Sie einen kleinen Onlineshop. Er muss wissen, wer seine Kunden sind, was er verkauft und wer was bestellt hat. Ein erster Versuch könnte alles in eine Tabelle packen, mit einer Zeile pro Bestellung und den Spalten `product1`, `product2`, `product3`. Das scheitert, sobald jemand vier Dinge bestellt, und es kopiert Name und Adresse des Kunden in jede Bestellung.



Jeder Pfeil führt vom Fremdschlüssel (FK) zum Primärschlüssel (PK), auf den er verweist.

Ein normalisiertes Shop-Schema: Jeder Fakt wird einmal gespeichert, und Fremdschlüssel halten die Beziehungen fest.

AUSGEARBEITETES BEISPIEL

Ausgearbeitetes Beispiel: Warum das Schema vier Tabellen hat

Das Schema oben beantwortet die Fragen des Shops, ohne Fakten zu kopieren.

customers	Eine Zeile pro Person. Eine geänderte E-Mail-Adresse ändert eine Zeile.
orders	Eine Zeile pro Bezahlvorgang. `customer_id` ist ein Fremdschlüssel, daher gehört jede Bestellung zu einem echten Kunden. Ein Kunde hat viele Bestellungen: eine *1:n-Beziehung*.
products	Eine Zeile pro Artikel im Angebot, mit aktuellem Preis und Lagerbestand.
order_items	Eine Bestellung enthält viele Produkte, und ein Produkt kommt in vielen Bestellungen vor: eine *n:m-Beziehung*. Relationale Datenbanken bilden sie mit einer Verknüpfungstabelle ab, mit einer Zeile pro Produkt und Bestellung.
Bewusste Kopie: unit_price	Der Preis wird beim Aufgeben der Bestellung in `order_items` kopiert. Das ist kein Normalisierungsfehler: Die Bestellung muss festhalten, was der Kunde tatsächlich bezahlt hat, auch wenn sich der Katalogpreis nächste Woche ändert.
Eine Bestellung aufgeben	Das Einfügen der Bestellung, das Einfügen ihrer Positionen und das Verringern des Lagerbestands jedes Produkts geschehen in einer Transaktion, sodass ein Absturz nie eine Bestellung ohne Positionen oder einen nicht verringerten Bestand hinterlassen kann.

Guter Schemaentwurf beginnt bei den Fragen, die das Unternehmen stellt, und den Fakten, die es nie verlieren darf, nicht bei den Bildschirmen der ersten Version der App.

Derselbe Shop könnte in einer Dokumentendatenbank jede Bestellung als ein Dokument speichern, mit den Positionen darin verschachtelt; das passt gut, um eine ganze Bestellung auf einmal zu lesen, erschwert aber Fragen wie „Gesamtumsatz pro Produkt“.

Backups, Wiederherstellung und Managed Databases

Festplatten fallen aus, Menschen führen das falsche `DELETE` aus, Software hat Bugs, und Angreifer verschlüsseln Daten, um Lösegeld zu erpressen. Ein Backup ist eine Kopie der Daten, die getrennt aufbewahrt wird und aus der die Datenbank wiederhergestellt werden kann. Drei professionelle Gewohnheiten zählen. Ein Backup, das nie wiederhergestellt wurde, ist eine Hoffnung, kein Backup; deshalb proben Teams Wiederherstellungen und messen deren Dauer. Kopien werden an mehr als einem Ort aufbewahrt (eine verbreitete Faustregel lautet: drei Kopien, auf zwei Arten von Medien, eine davon extern). Und ein *Replikat*, eine zweite Datenbank, die die erste fortlaufend kopiert, schützt vor dem Ausfall einer Maschine, aber nicht vor

Fehlern: Ein falsches `DELETE` erreicht das Replikat innerhalb von Millisekunden. Viele Datenbanken können ein regelmäßiges Vollbackup mit dem gespeicherten Transaktionslog kombinieren, um auf jeden beliebigen Zeitpunkt wiederherzustellen; das nennt man *Point-in-Time-Recovery*.

Eine Datenbank gut zu betreiben (Patches einspielen, Tuning, Replikation, Backups und Wiederherstellung um drei Uhr morgens) ist anspruchsvolle, nie endende Arbeit; deshalb verkaufen Cloud-Anbieter eine Managed Database: Amazon RDS, Azure SQL Database und Google Cloud SQL sind Beispiele. Der Anbieter kümmert sich um die Maschinen, Software-Updates, automatische Backups und das Failover auf eine Standby-Kopie; beim Kunden bleiben weiterhin das Schema, die Abfragen, die Indizes, die Zugriffsrechte und die Kosten. Managed Services werden in Server und die Cloud allgemeiner behandelt.

Warum das für die Architektur wichtig ist

Daten überdauern meist den Code um sie herum: Anwendungen werden alle paar Jahre neu geschrieben, während Kundendaten und Krankengeschichten über Jahrzehnte aufbewahrt werden. Entscheidungen darüber, wo Daten liegen, welche Art von Datenbank sie hält, welches Programm für sie zuständig ist und welche Garantien sie brauchen, gehören daher zu den am teuersten rückgängig zu machenden. Mehrere große Architekturfragen erwachsen direkt aus diesem Abschnitt: ob jeder Service seine eigene Datenbank besitzen sollte (Datenhoheit), welche Datenbank zu welcher Aufgabe passt (Eine Datenbank wählen) und wie man Daten korrekt hält, wenn eine geschäftliche Aktion mehrere Datenbanken berührt und keine einzelne ACID-Transaktion mehr verfügbar ist (Sagas). Wenn ein Architekt fragt: „Wo liegt dafür die Source of Truth?“, dann ist dieser Abschnitt der Grund.

PRÜFEN SIE IHR VERSTÄNDNIS

▼ Warum ist eine Tabellenkalkulationsdatei ein schlechter Ort für die Bestellungen eines Onlineshops, sobald viele Kunden gleichzeitig einkaufen?

Gleichzeitige Schreiber können sich gegenseitig überschreiben, ein Absturz mitten beim Speichern kann die Datei beschädigen, um die Bestellungen eines Kunden zu finden, muss alles gelesen werden, und nichts setzt Regeln durch, etwa dass jede Bestellung einen echten Kunden hat. Ein DBMS löst alle vier Probleme.

▼ Was macht im Shop-Schema customer_id in der Tabelle orders zu einem Fremdschlüssel, und was bringt Ihnen das?

Die Spalte enthält den Primärschlüssel einer Zeile in der Tabelle customers und ist als Verweis darauf deklariert. Die Datenbank weist dann Bestellungen für Kunden zurück, die es nicht gibt.

▼ Warum nicht einfach jede Spalte jeder Tabelle indizieren?

Jeder Index muss bei jedem Einfügen, Aktualisieren und Löschen aktualisiert werden und belegt Platz. Zu viele Indizes machen Schreibvorgänge langsam und Speicher teuer; deshalb wählt man Indizes passend zu den tatsächlich ausgeführten Abfragen.

▼ Welche ACID-Eigenschaft ist verletzt, wenn ein Absturz Alice belastet, Bob aber keine Gutschrift erhalten hat?

Atomarität: Die beiden Änderungen hätten gemeinsam oder gar nicht stattfinden müssen.

▼ Ein Team sagt: „Wir haben ein Replikat in einem anderen Rechenzentrum, also brauchen wir keine Backups.“ Was hat es übersehen?

Ein Replikat kopiert Fehler und beschädigte Daten genauso getreu wie gute Daten, sodass ein versehentliches Löschen oder ein Ransomware-Angriff es binnen Augenblicken erreicht. Backups, idealerweise mit Point-in-Time-Recovery, schützen davor.

QUELLEN

- PostgreSQL documentation: Transactions (tutorial): https://www.postgresql.org/docs/current/tutorial-transactions.html
- PostgreSQL documentation: Indexes, introduction: https://www.postgresql.org/docs/current/indexes-intro.html
- PostgreSQL documentation: Joins between tables: https://www.postgresql.org/docs/current/tutorial-join.html
- Martin Fowler: NosqlDefinition: https://martinfowler.com/bliki/NosqlDefinition.html

ABSCHNITT 0.6 VON 0.8 · EINSTEIGER · 17 MIN. LESEZEIT

APIs, JSON und wie getrennte Systeme zusammenarbeiten

DIE IDEE IN KÜRZE

Eine API ist wie die Speisekarte eines Restaurants und sein Kellner. Die Speisekarte sagt Ihnen genau, was Sie bestellen können und was Sie bekommen; der Kellner bringt Ihre Bestellung in die Küche und das Gericht zurück. Sie betreten die Küche nie, und die Küche kann ändern, wie sie kocht, ohne dass Sie es merken, solange die Gerichte auf der Karte dieselben bleiben. Software nutzt APIs auf dieselbe Weise: Ein Programm bittet ein anderes in einem vereinbarten Format um etwas und erhält eine vereinbarte Antwort zurück.

WAS SIE WISSEN MÜSSEN

- Eine API ist ein Vertrag zwischen Softwareteilen: was angefragt werden kann, mit welchen Eingaben und was zurückkommt, während die Implementierung dahinter verborgen bleibt und sich frei ändern kann.
- In-Process-APIs sind Funktionsaufrufe innerhalb eines Programms; Web-APIs gehen über ein Netzwerk, sind daher langsamer und können auf eine Weise scheitern, wie es ein Funktionsaufruf nicht kann.
- REST-APIs stellen Ressourcen unter URLs bereit, wirken mit HTTP-Methoden (GET, POST, PUT, PATCH, DELETE) auf sie ein, melden Ergebnisse mit Statuscodes und tauschen meist JSON aus.
- API-Schlüssel identifizieren eine aufrufende Anwendung; Access Tokens sind kurzlebige Zugangsdaten, oft im Namen eines Nutzers; beide dürfen nur über HTTPS übertragen werden und gehören nicht in den Quellcode.
- Bei einem Webhook ruft der Anbieter die URL des Clients auf, wenn ein Ereignis eintritt; Empfänger müssen Signaturen prüfen und mit Duplikaten umgehen.
- Synchrone Aufrufe warten auf eine Antwort; asynchrone Nachrichten laufen über eine Queue, sodass Sender und Empfänger nicht gleichzeitig verfügbar sein müssen.
- Breaking Changes schaden Clients, daher werden APIs versioniert, und Rate Limits (HTTP 429) schützen Anbieter; jede Drittanbieter-API ist eine Abhängigkeit, die man isolieren und einplanen muss.

Was eine API ist

Eine API (Programmierschnittstelle) ist der definierte Weg, auf dem eine Software eine andere bittet, etwas zu tun. Sie legt fest, was angefragt werden kann, was mitgeliefert werden muss, was zurückkommt und was schiefgehen kann. Das entscheidende Wort ist *definiert*: Eine API ist

ein Vertrag. Der Anbieter verspricht, sie funktionsfähig zu halten; der Nutzer verspricht, nur das zu verwenden, was sie beschreibt.

Ein Restaurant ist die übliche Analogie. Die Speisekarte führt auf, was Sie bestellen können und was jedes Gericht enthält; der Kellner bringt Ihre Bestellung in die Küche und das Ergebnis zurück. Sie gehen nicht in die Küche, und Sie müssen nicht wissen, ob der Koch mit Gas oder Induktion kocht. Die Küche kann Ausstattung, Personal und Lieferanten nach Belieben wechseln, solange die Gerichte auf der Karte weiterhin wie beschrieben ankommen. Die Speisekarte ist die API, die Küche ist die Implementierung, und die Trennung zwischen beiden erlaubt es beiden Seiten, sich unabhängig zu ändern.

Das ist derselbe Gedanke wie bei den Schnittstellen in Code organisieren, angewandt auf jeder Ebene. Es ist auch der Grund, warum Architekten APIs so wichtig nehmen: An einer API trifft die Arbeit eines Teams auf die eines anderen, und sobald andere von ihr abhängen, ist sie teuer zu ändern.

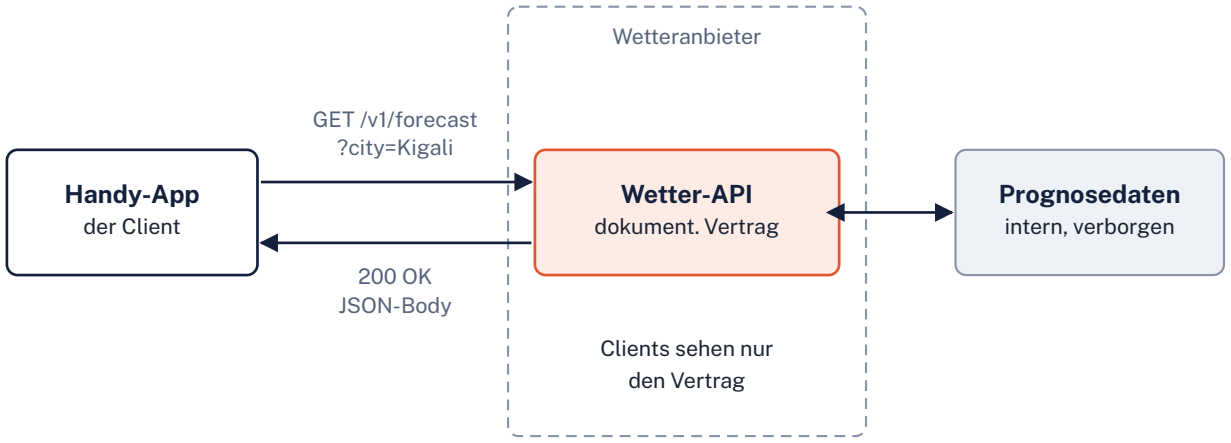
In-Process-APIs und Web-APIs

Manche APIs werden innerhalb eines laufenden Programms aufgerufen. Wenn Python-Code `json.loads(text)` aufruft, nutzt er die API des Moduls `json` der Standardbibliothek: ein Funktionsaufruf, der Nanosekunden dauert und nur scheitert, wenn die Eingabe fehlerhaft ist. Die öffentlichen Funktionen und Klassen einer Bibliothek sind ihre In-Process-API.

Eine Web-API wird über ein Netzwerk aufgerufen, meist mit HTTP (siehe Wie Computer kommunizieren). Aufrufer und Anbieter können auf verschiedenen Maschinen laufen, in verschiedenen Sprachen geschrieben sein und verschiedenen Organisationen gehören. Eine Mobile-Banking-App spricht mit der Web-API der Bank; die Server der Bank sprechen mit der API eines Kartennetzwerks; ein Shop spricht mit der API eines Zahlungsanbieters.

Der Unterschied ist nicht kosmetisch. Ein Netzwerkaufruf ist tausend- bis millionenfach langsamer als ein Funktionsaufruf, und er kann auf eine Weise scheitern, wie es ein Funktionsaufruf nicht kann: Das Netzwerk verliert die Anfrage, der Server ist überlastet, die Antwort kommt an, nachdem der Aufrufer aufgegeben hat, oder die Anfrage war erfolgreich, aber die Antwort ging verloren, sodass der Aufrufer nicht erkennen kann, ob die Zahlung durchging. Ein Großteil des Entwurfs verteilter Systeme, behandelt in Kapitel 6, dreht sich darum, mit diesen Ausfällen zu leben.

Ein Web-API-Aufruf



Der Client sieht nur den Vertrag der API; wie der Anbieter Prognosen speichert und berechnet, bleibt dahinter verborgen.

REST: Ressourcen, URLs und Verben

Die meisten Web-APIs folgen heute dem Stil namens REST (Representational State Transfer), den Roy Fielding in seiner Dissertation von 2000 beschrieb. Fielding definierte REST als eine Reihe von Randbedingungen für große vernetzte Systeme wie das Web; im Alltag meint „eine REST-API“ etwas Einfacheres: eine HTTP-API, die um **Ressourcen** herum organisiert ist.

Eine **Ressource** ist etwas, das die API bereitstellt: ein Kunde, eine Bestellung, eine Prognose. Jede hat eine URL, etwa `/orders/1042`. Auch eine Sammlung davon hat eine URL, etwa `/orders`. Der Client wirkt mit den Standard-HTTP-Methoden auf Ressourcen ein, liest den Statuscode des Ergebnisses und tauscht Daten im Body aus, meist als JSON. Dieselben wenigen Verben funktionieren daher für jede Ressource, was gut entworfene REST-APIs vorhersehbar macht.

Eine Ressource, die Standardverben

Anfrage	Bedeutung	Typischer Erfolgsstatus
`GET /orders?customer=42`	Bestellungen auflisten, gefiltert über einen Query-Parameter. Lesezugriffe ändern nie etwas.	200 OK
`GET /orders/1042`	Eine Bestellung abrufen.	200 OK (oder 404 Not Found, wenn es diese Bestellung nicht gibt)
`POST /orders`	Eine neue Bestellung aus dem JSON im Body der Anfrage anlegen.	201 Created, mit der URL der neuen Bestellung
`PUT /orders/1042`	Bestellung 1042 durch die gesendete Repräsentation ersetzen.	200 OK oder 204 No Content
`PATCH /orders/1042`	Nur einige Felder ändern, etwa die Lieferadresse.	200 OK
`DELETE /orders/1042`	Die Bestellung stornieren oder entfernen.	204 No Content

Statuscodes tragen eine Bedeutung, auf die ein Client-Programm reagieren kann. Ein **4xx**-Code besagt, dass der Client etwas falsch gemacht hat: 400 bei einer fehlerhaften Anfrage, 401, wenn Zugangsdaten fehlen oder ungültig sind, 403, wenn der Aufrufer bekannt, aber nicht berechtigt ist, 404, wenn die Ressource nicht existiert, 429, wenn er zu viele Anfragen gesendet hat. Ein **5xx**-Code besagt, dass der Server versagt hat: 500 bei einem unerwarteten Fehler, 503, wenn er vorübergehend nicht verfügbar ist. Die Unterscheidung ist wichtig, weil eine unveränderte Wiederholung einer 4xx-Anfrage erneut scheitert, während eine Wiederholung nach einem 503 etwas später gelingen kann.

Daten werden als JSON (JavaScript Object Notation) übertragen, einem Textformat, das als RFC 8259 standardisiert ist. Es hat nur wenige Bausteine: Objekte (Name-Wert-Paare in geschweiften Klammern), Arrays (Listen in eckigen Klammern), Zeichenketten in doppelten Anführungszeichen, Zahlen, `true`, `false` und `null`. Trotz des Namens kann jede gängige Sprache es lesen und schreiben, weshalb es zur gemeinsamen Sprache der Web-APIs wurde. Es gibt andere Stile für andere Bedürfnisse: Bei GraphQL gibt der Client genau an, welche Felder er haben will, und gRPC nutzt ein kompaktes Binärformat für schnelle Aufrufe zwischen Services; Kapitel 6 vergleicht sie.

AUSGEARBEITETES BEISPIEL

Ausgearbeitetes Beispiel: eine Wetter-API aufrufen

Eine Reise-App möchte die Wettervorhersage für morgen in Kigali. Der Anbieter (hier ein erfundenes api.example.com) dokumentiert einen *Endpoint* (eine URL, die eine bestimmte Art von Anfrage annimmt) und stellt der App einen API-Schlüssel aus.

1. Den Vertrag lesen	Die Dokumentation sagt: `GET /v1/forecast` mit den Query-Parametern `city` und `days`; den Schlüssel in einem `Authorization`-Header senden; Antworten sind JSON; Limit 60 Anfragen pro Minute.
2. Die Anfrage senden	Die App sendet die unten gezeigte HTTP-Anfrage über HTTPS, sodass Schlüssel und Daten bei der Übertragung verschlüsselt sind.
3. Den Status prüfen	200 bedeutet, dass der Body eine Vorhersage enthält. 401 bedeutet, dass der Schlüssel falsch ist; 429 heißt langsamer; 503 heißt später erneut versuchen.
4. Das JSON auswerten	Die App liest `days[0].max_c` und `days[0].rain_chance` und ignoriert Felder, die sie nicht braucht.
5. Fehler behandeln	Kommt innerhalb von, sagen wir, 3 Sekunden keine Antwort, zeigt die App die letzte zwischengespeicherte Vorhersage statt eines leeren Bildschirms.

Jeder Schritt hängt vom Vertrag ab, nicht davon, wie der Anbieter Vorhersagen berechnet. Der Anbieter kann morgen auf neue Server oder ein neues Wettermodell umsteigen, ohne dass sich die App ändert.

Schritt 5 ist der Schritt, den Einsteiger vergessen: Jeder Aufruf über ein Netzwerk kann langsam sein oder scheitern, daher muss der Aufrufer entscheiden, was dann geschieht.

```
GET /v1/forecast?city=Kigali&days=2 HTTP/1.1
Host: api.example.com
Authorization: Bearer sk_test_4f9a...
Accept: application/json

HTTP/1.1 200 OK
Content-Type: application/json

{
  "city": "Kigali",
  "units": "metric",
  "days": [
    {"date": "2026-10-23", "min_c": 16, "max_c": 27, "rain_chance": 0.6},
    {"date": "2026-10-24", "min_c": 16, "max_c": 26, "rain_chance": 0.4}
  ]
}
```

Anfrage und Antwort aus dem ausgearbeiteten Beispiel.

```
import os
import requests

API_KEY = os.environ["WEATHER_API_KEY"]   # never hard-code secrets

resp = requests.get(
    "https://api.example.com/v1/forecast",
    params={"city": "Kigali", "days": 2},
    headers={"Authorization": f"Bearer {API_KEY}"},
    timeout=3,                # give up after 3 seconds
)
resp.raise_for_status()       # raises an error for 4xx and 5xx
tomorrow = resp.json()["days"][0]
print(tomorrow["max_c"], "degrees, rain chance", tomorrow["rain_chance"])
```

Derselbe Aufruf aus Python mit der verbreiteten Bibliothek requests, einschließlich Timeout und Statusprüfung.

Wer ruft an? API-Schlüssel und Tokens

Die meisten APIs müssen wissen, wer aufruft, um abzurechnen, zu begrenzen und zu entscheiden, was der Aufrufer sehen darf. Zwei Begriffe sollte man von Anfang an trennen. *Authentifizierung* beantwortet „Wer sind Sie?“; *Autorisierung* beantwortet „Was dürfen Sie tun?“

Ein API-Schlüssel ist eine lange geheime Zeichenkette, die die aufrufende Anwendung identifiziert und mit jeder Anfrage gesendet wird, typischerweise in einem Header. Er ist einfach und identifiziert ein Programm, keine Person. Ein Access Token ist ein kurzlebiger

Berechtigungsnachweis, der nach einer Anmeldung ausgestellt wird, oft im Namen eines bestimmten Nutzers und mit bestimmten Berechtigungen („den Kalender dieses Nutzers lesen, aber nicht seine E-Mails“); er wird meist als `Authorization: Bearer <token>` gesendet. Tokens laufen ab, sodass ein gestohlenes nur begrenzte Zeit nützt. Die zugrunde liegenden Standardprotokolle, OAuth 2.0 und OpenID Connect, werden in Identität und Zugriff behandelt.

Was auch immer verwendet wird, die Regeln sind dieselben: Zugangsdaten nur über HTTPS senden, sie aus dem Quellcode heraushalten und aus mobilen Apps, aus denen jeder sie extrahieren kann, sie in einem Secrets Manager oder in Umgebungseinstellungen speichern und sie schnell widerrufen und ersetzen können.

Webhooks: Die API ruft Sie zurück

Manchmal muss der Client wissen, wann auf der Seite des Anbieters etwas geschieht: Eine Zahlung gelingt, ein Paket wird zugestellt, ein Dokument wird unterschrieben. Der Client könnte *pollen*, also alle paar Sekunden fragen „Ist es schon passiert?“, was Anfragen verschwendet und es trotzdem spät bemerkt. Ein Webhook kehrt die Richtung um: Der Client registriert eine eigene URL, und der Anbieter sendet ein HTTP-POST an diese URL, wenn das Ereignis eintritt. Den Begriff prägte Jeff Lindsay 2007.

Polling fragt wiederholt; ein Webhook wartet darauf, informiert zu werden, um den Preis neuer Pflichten für den Empfänger.

Webhooks bringen Pflichten mit sich. Der empfangende Endpoint steht im öffentlichen Internet, daher muss er prüfen, dass jeder Aufruf wirklich vom Anbieter stammt, meist durch Verifizieren einer kryptografischen Signatur in einem Header. Anbieter wiederholen den Versand, wenn der Empfänger nicht antwortet, sodass dasselbe Ereignis zweimal ankommen kann und der Empfänger Duplikate erkennen muss (die Verarbeitung eines Duplikats darf die Bestellung nicht zweimal versenden). Und Ereignisse können in falscher Reihenfolge ankommen.

Auf eine Antwort warten oder nicht: synchron und asynchron

Alles bisher war synchron: Der Aufrufer sendet eine Anfrage und wartet auf die Antwort, bevor er weitermacht, wie bei einem Telefonat. Das ist leicht nachzuvollziehen und die richtige Wahl, wenn der Aufrufer die Antwort wirklich sofort braucht, etwa „Wurde diese Kartenzahlung genehmigt?“. Die Schwäche: Der Aufrufer ist nur so verfügbar und so schnell wie alles, worauf er wartet.

Die Alternative ist asynchrones Messaging, eher wie ein Brief, den man in einen Briefkasten wirft. Der Sender legt eine Nachricht in eine Message Queue und macht weiter. Der Empfänger holt Nachrichten aus der Queue, wenn er bereit ist. Ist der Empfänger langsam oder eine Stunde lang ausgefallen, warten die Nachrichten, statt zu scheitern. Wenn der Shop eine Bestellung bestätigt, muss das Senden der Bestätigungs-E-Mail den Bildschirm des Kunden nicht aufhalten; eine Nachricht `OrderPlaced` kann in eine Queue gehen, die der E-Mail-Service einen Moment später abarbeitet.

Fällt der E-Mail-Service aus, warten Nachrichten in der Queue, statt verloren zu gehen.

Ein synchroner Aufruf bindet den Aufrufer an die Verfügbarkeit des Aufgerufenen; eine Queue entkoppelt sie zeitlich.

Asynchronität ist nicht kostenlos. Der Sender erfährt nicht sofort, ob die Arbeit gelungen ist, das System wird schwerer nachzuverfolgen und zu testen, und Nachrichten können mehr als einmal zugestellt werden. Die Entscheidung, welche Interaktionen synchron und welche asynchron sein sollen, ist eine der zentralen Entscheidungen in verteilten Systemen und wird ausführlich in Kommunikation und APIs und Queues und Backpressure behandelt.

ENTSCHEIDUNGSHILFE

Entscheidungshilfe: jetzt aufrufen oder eine Nachricht senden?

Eine erste Orientierung für eine Interaktion zwischen zwei Teilen eines Systems. Kapitel 6 verfeinert sie.

Q1 **Braucht der Aufrufer das Ergebnis, bevor er seinem eigenen Nutzer antworten kann (zum Beispiel, ob die Karte genehmigt wurde)?**

- **Ja** Nutzen Sie einen synchronen Aufruf, mit einem Timeout und einem Plan für den Fall, dass die andere Seite langsam oder ausgefallen ist.
- **Nein** Weiter zur nächsten Frage.

Q2 **Kann die Arbeit ein paar Sekunden oder Minuten später geschehen, ohne dass es jemand merkt (eine E-Mail senden, einen Bericht aktualisieren)?**

- **Ja** Senden Sie eine asynchrone Nachricht über eine Queue, damit ein langsamer oder ausgefallener Empfänger den Aufrufer nicht aufhält.
- **Nein** Weiter zur nächsten Frage.

Q3 **Ist die Arbeit langsam (Sekunden bis Stunden), etwa das Erzeugen eines großen Exports?**

- **Ja** Nehmen Sie die Anfrage sofort an (zum Beispiel mit 202 Accepted), erledigen Sie die Arbeit über eine Queue, und teilen Sie dem Client per Webhook oder über eine abfragbare Status-URL mit, wann sie fertig ist.
- **Nein** Ein einfacher synchroner Aufruf genügt meist; halten Sie ihn schnell und setzen Sie ein Timeout.

Mit den APIs anderer leben: SDKs, Versionen und Limits

Viele Anbieter veröffentlichen ein SDK (Software Development Kit): eine Bibliothek für jede verbreitete Sprache, die die HTTP-Aufrufe in gewöhnliche Funktionen verpackt, sodass ein Entwicklungsteam `client.refunds.create(payment_id)` schreibt, statt Anfragen von Hand zu bauen. SDKs sparen Zeit und kümmern sich um Details wie Wiederholungen und Signaturen, sind aber eine weitere Abhängigkeit, die aktuell gehalten werden muss.

Sobald eine API Nutzer hat, wird jede Änderung heikel. Ein neues optionales Feld in einer Antwort hinzuzufügen, ist meist unbedenklich. Ein Feld umzubenennen, zu entfernen, seinen Typ oder seine Bedeutung zu ändern oder eine optionale Eingabe zur Pflicht zu machen, ist ein Breaking Change: Jeder Client, der sich auf das alte Verhalten verlassen hat, bricht, oft in Produktion und oft bei einem Anbieter, der diese Clients weder sehen noch aktualisieren kann (denken Sie an mobile Apps, die Nutzer nie aktualisieren). Anbieter versionieren ihre APIs

daher, zum Beispiel mit `/v1/` und `/v2/` in der URL oder mit einem Versions-Header, lassen die alte Version für einen veröffentlichten Zeitraum weiterlaufen und kündigen *Deprecations* (Hinweise, dass eine Funktion zurückgezogen wird) lange im Voraus an. Hyrum Wrights Beobachtung, bekannt als Hyrums Gesetz, fügt eine Warnung hinzu: Bei genügend Nutzern hängt jemand von jedem beobachtbaren Verhalten einer API ab, auch von solchem, das die Dokumentation nie versprochen hat.

Anbieter schützen sich außerdem mit einem Rate Limit: einer Obergrenze wie 100 Anfragen pro Minute und Schlüssel. Anfragen über dem Limit werden mit Status 429 Too Many Requests abgelehnt, oft mit einem `Retry-After`-Header, der angibt, wann man es erneut versuchen kann. Clients sollten das respektieren, ihre Anfragen verteilen, Antworten zwischenspeichern, die sich selten ändern, und sich zurückhalten, wenn sie dazu aufgefordert werden.

Drittanbieter-APIs nutzen: selbst bauen oder einkaufen

Moderne Anwendungen werden ebenso zusammengesetzt wie geschrieben. Eine Essensliefer-App nutzt vielleicht die API eines Zahlungsanbieters für Kartenzahlungen, eine Karten-API für Adressen und Routen, eine E-Mail- oder SMS-API für Belege und einen Push-Benachrichtigungsdienst für „Ihr Fahrer ist in der Nähe“. Jede davon spart Monate an Spezialistenarbeit und bringt Fachwissen mit (Kartensicherheit, globale Kartendaten, E-Mail-Zustellbarkeit), mit dem nur wenige Teams mithalten könnten.

Jede bringt aber auch eine Abhängigkeit mit, die das Team nicht kontrolliert: Die Ausfälle des Anbieters werden zu Ihren Ausfällen, seine Preisänderungen zu Ihren Kosten, sein Umgang mit Daten wird Teil Ihrer Compliance, und ein Wechsel kann bedeuten, Code neu zu schreiben. Fachleute steuern das bewusst. Sie isolieren jeden Anbieter hinter einer kleinen eigenen Schnittstelle, sodass nur ein Teil des Codes die API des Anbieters kennt; sie setzen Timeouts und legen fest, was geschieht, wenn der Anbieter ausgefallen ist; sie beobachten die Statusseite und das Änderungsprotokoll des Anbieters; und sie halten fest, was ein Wechsel kosten würde.

Warum das für die Architektur wichtig ist

Architektur dreht sich weitgehend um die Teile eines Systems und die Verträge zwischen ihnen, und APIs sind diese Verträge in konkreter Form. Eine API, die sich leicht richtig nutzen und schwer kaputt machen lässt, lässt Teams parallel arbeiten und ihr Innenleben frei ändern; eine undichte oder instabile koppelt alle an alle. Die Wahl zwischen synchronen Aufrufen und asynchronen Nachrichten bestimmt, wie sich Ausfälle ausbreiten. Kapitel 3 erklärt die Prinzipien hinter guten Schnittstellen (Information Hiding, Abstraktion und Schnittstellen), und Kapitel 6 zeigt, wie Entscheidungen über APIs und Messaging verteilte Systeme gelingen oder scheitern lassen.

PRÜFEN SIE IHR VERSTÄNDNIS

▼ Welcher Teil der Restaurant-Analogie entspricht der API, welcher der Implementierung?

Die Speisekarte (und die Art, wie der Kellner Bestellungen aufnimmt) ist die API: die vereinbarte Liste dessen, was angefragt werden kann und was zurückkommt. Die Küche ist die Implementierung, die sich frei ändern kann, solange die Versprechen der Speisekarte eingehalten werden.

▼ Was ist der Unterschied zwischen einer 404- und einer 503-Antwort, und welche lohnt einen späteren erneuten Versuch?

404 Not Found besagt, dass die angefragte Ressource nicht existiert, ein Problem auf Client-Seite, das eine Wiederholung nicht behebt. 503 Service Unavailable besagt, dass der Server vorübergehend nicht antworten kann, daher kann eine Wiederholung nach einer Pause gelingen.

▼ Warum muss ein Webhook-Empfänger damit rechnen, dasselbe Ereignis zweimal zu erhalten?

Anbieter wiederholen den Versand, wenn sie nicht rechtzeitig eine Erfolgsantwort erhalten, selbst wenn die erste Zustellung tatsächlich verarbeitet wurde; Duplikate sind also normal. Der Empfänger muss ein bereits behandeltes Ereignis erkennen und nicht erneut darauf reagieren.

▼ Ist das Hinzufügen eines neuen optionalen Felds zu einer JSON-Antwort normalerweise ein Breaking Change?

Meist nicht, weil sich gut verhaltende Clients Felder ignorieren, die sie nicht kennen. Ein Feld zu entfernen oder umzubenennen oder seinen Typ zu ändern, ist dagegen ein Breaking Change.

QUELLEN

- Roy Fielding (2000), Architectural Styles and the Design of Network-based Software Architectures, Chapter 5: REST: https://ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm
- RFC 8259: The JavaScript Object Notation (JSON) Data Interchange Format: https://www.rfc-editor.org/rfc/rfc8259
- MDN Web Docs: 429 Too Many Requests: https://developer.mozilla.org/en-US/docs/Web/HTTP/Reference/Status/429
- Stripe documentation: Receive events in your webhook endpoint: https://docs.stripe.com/webhooks

- Hyrum's Law: https://www.hyrumslaw.com/

Server, Betriebssysteme, virtuelle Maschinen, Container und Cloud-Dienste

DIE IDEE IN KÜRZE

Ein Server ist einfach ein Computer, dessen Aufgabe es ist, Anfragen zu beantworten, untergebracht in einem großen Gebäude voller anderer Server, einem sogenannten Rechenzentrum. Statt eigene zu kaufen, mieten die meisten Organisationen heute Rechenleistung bei Cloud-Anbietern, so wie Haushalte Strom aus dem Netz beziehen, statt einen Generator zu betreiben. Sie können einen ganzen virtuellen Computer mieten, eine fertige Plattform für Ihren Code oder fertige Software, und zahlen nur für das, was Sie nutzen. Je mehr Sie fertig mieten, desto weniger müssen Sie sich selbst kümmern und desto weniger haben Sie unter Kontrolle.

WAS SIE WISSEN MÜSSEN

- Ein Server ist jeder Computer oder jedes Programm, das auf Anfragen wartet und sie beantwortet; die meisten Server im Internet laufen unter Linux und stehen in Rechenzentren, die für Stromversorgung, Kühlung und Sicherheit gebaut sind.
- Ein Hypervisor lässt eine physische Maschine mehrere virtuelle Maschinen ausführen, jede mit eigenem Gastbetriebssystem und von den anderen isoliert.
- Ein Container verpackt eine Anwendung mit ihren Bibliotheken und Einstellungen und läuft als isolierter Prozess, der den Kernel des Hosts mitbenutzt; er ist daher leichter und schneller gestartet als eine VM, aber weniger stark isoliert.
- Cloud-Computing (NIST SP 800-145) bedeutet bedarfsgesteuerten Zugriff in Selbstbedienung und mit nutzungsabhängiger Abrechnung auf gemeinsam genutzte, elastische Ressourcen über das Netzwerk.
- IaaS, PaaS, Serverless (FaaS) und SaaS geben schrittweise mehr vom Stack an den Anbieter ab und tauschen Kontrolle gegen weniger Betriebsaufwand; für ihre Daten und ihre Konfiguration bleiben die Kunden immer selbst verantwortlich.
- Regionen sind voneinander isolierte geografische Gebiete; Availability Zones sind getrennte Gruppen von Rechenzentren innerhalb einer Region; CDNs liefern statische Inhalte von Edge-Standorten in der Nähe der Nutzer aus.
- Vertikal skalieren (Scale-up) heißt eine größere Maschine; horizontal skalieren (Scale-out) heißt mehr Maschinen hinter einem Load Balancer, was weiter reicht und Ausfälle übersteht, aber nur, wenn die Software dafür ausgelegt ist.

Ein Server ist einfach ein Computer, der dient

Ein Server ist ein Computer, dessen Aufgabe es ist, auf Anfragen zu warten und sie zu beantworten: Webseiten, API-Aufrufe, Datenbankabfragen, Dateien. Das Wort beschreibt eine Rolle, keine besondere Art von Maschine. Ein Laptop, auf dem eine kleine Webanwendung läuft, ist ein Server, solange er Anfragen beantwortet; ein Rack-Server in einem Rechenzentrum ist ein Server, weil er nie etwas anderes tut. Dasselbe Wort steht auch für das Programm, das die Anfragen bedient (ein „Webserver“ wie nginx), daher kommt es auf den Zusammenhang an.

Server-Hardware hat dieselben Bestandteile wie jeder Computer (siehe Was Software ist): Prozessoren, Arbeitsspeicher, Massenspeicher und Netzwerkkarten. Der Unterschied liegt in der Gewichtung. Server haben meist viele Prozessorkerne, viel Arbeitsspeicher, redundante Netzteile und Festplatten, damit ein defektes Teil die Maschine nicht stoppt, eine Fernwartung, damit niemand im Raum sein muss, und überhaupt keinen Bildschirm und keine Tastatur. Sie sind dafür gebaut, jahrelang ohne Ausschalten zu laufen.

Die meisten Server im Internet laufen unter Linux oder einem anderen Unix-artigen Betriebssystem; Windows Server ist in Organisationen verbreitet, die auf Microsoft-Produkte setzen. Ein Server-Betriebssystem erledigt dieselben Aufgaben wie das auf Ihrem Laptop (Prozesse ausführen, Arbeitsspeicher und Dateien verwalten, mit dem Netzwerk kommunizieren), wird aber meist über eine Textkommandozeile per Fernverbindung administriert und ist so konfiguriert, dass es sich automatisch statt von Hand neu aufbauen lässt.

Rechenzentren: wo die Server wohnen

Server stehen in einem Rechenzentrum: einem Gebäude, das um ihre Bedürfnisse herum entworfen ist. Tausende Maschinen sitzen in Racks und werden mit Luft oder Flüssigkeit gekühlt, weil sie fast ihren gesamten Strom in Wärme umwandeln. Der Strom kommt über mehr als eine Zuleitung, abgesichert durch Batterien für die ersten Sekunden eines Ausfalls und danach durch Dieselgeneratoren. Netzwerkverbindungen kommen von mehreren Anbietern. Für die Türen braucht man Ausweise und oft auch Fingerabdrücke.

Eine Organisation kann ein eigenes Rechenzentrum betreiben oder Rack-Fläche in einem fremden mieten (*Colocation* genannt), und vor etwa 2006 war das fast die einzige Wahl. Server selbst zu betreiben (*On-Premises*) bedeutet, Maschinen Monate vor dem Bedarf zu kaufen, zu schätzen, wie viele, für sie zu bezahlen, ob sie ausgelastet sind oder nicht, und Leute zu beschäftigen, die nachts defekte Festplatten tauschen. Diese Kosten und diese Unbeweglichkeit sind das Problem, das die Cloud lösen wollte.

Virtuelle Maschinen: viele Computer in einem

Ein typischer Server ist größtenteils untätig, wenn er nur eine Anwendung ausführt. Eine virtuelle Maschine (VM) lässt eine physische Maschine so tun, als wäre sie mehrere. Jede VM hält sich für einen vollständigen Computer mit eigenen Prozessoren, eigenem Arbeitsspeicher, eigener Festplatte und Netzwerkkarte und führt ein eigenes Betriebssystem aus, den sogenannten *Gast*. Eine Softwareschicht namens Hypervisor teilt die echte Hardware unter den VMs auf und hält sie voneinander getrennt, sodass ein Absturz oder Angriff in einer die anderen nicht erreicht. Manche Hypervisoren laufen direkt auf der Hardware (VMware ESXi, Microsoft Hyper-V, KVM, das Teil von Linux ist); andere laufen als Anwendung auf einem normalen Betriebssystem (VirtualBox).

Die Virtualisierung hat die Wirtschaftlichkeit von Servern verändert. Eine Maschine lässt sich mit VMs verschiedener Teams oder Kunden füllen. Eine VM ist eine Menge von Dateien, daher kann sie kopiert, gesichert, auf eine andere Maschine verschoben und in Minuten angelegt werden, statt bestellt und geliefert zu werden. Beachten Sie, dass „virtuelle Maschine“ hier anders gemeint ist als bei den Sprach-VMs wie der Java Virtual Machine, die in Was Software ist erwähnt wird: Jene führen Programme aus, diese bilden ganze Computer nach.

Container: die Anwendung mit allem ausliefern, was sie braucht

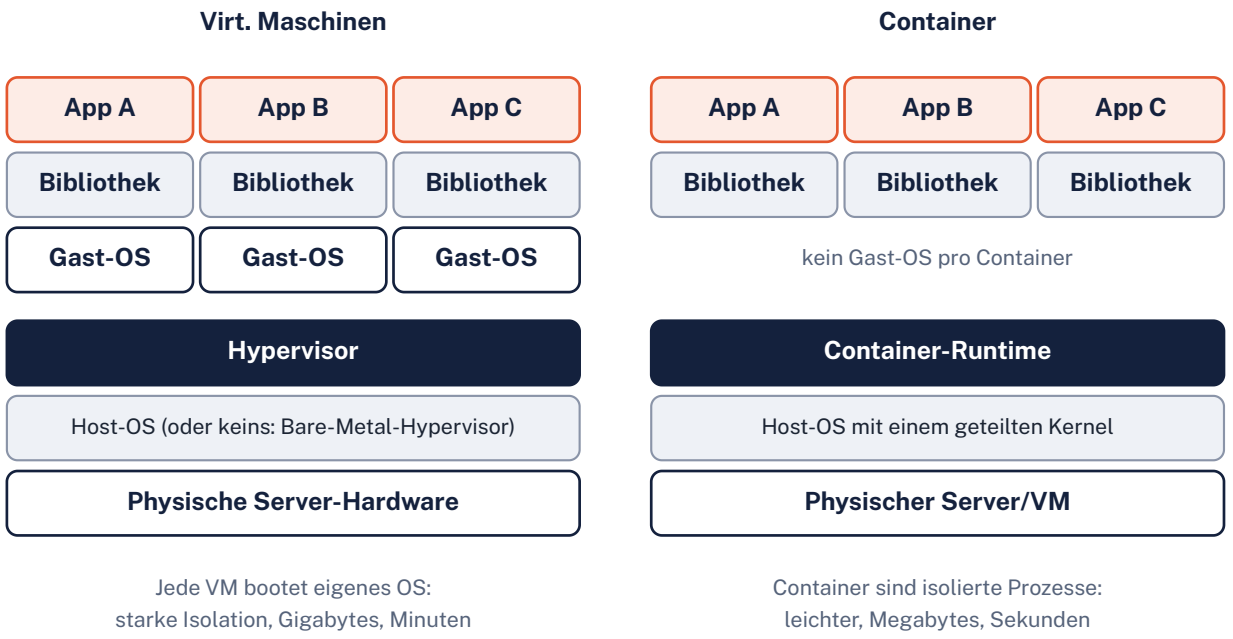
Jeder Entwickler kennt den Satz „Bei mir funktioniert es“. Das Programm hängt von einer bestimmten Sprachversion, bestimmten Bibliotheken und bestimmten Einstellungen ab, und auf dem Server sind sie leicht anders. Ein Container löst das, indem er eine Anwendung zusammen mit allem, was sie zum Laufen braucht (ihrer Laufzeitumgebung, ihren Bibliotheken, ihrer Konfiguration), in eine Standardeinheit verpackt, die auf einem Laptop, einem Testserver oder in der Cloud gleich läuft. Der Name kommt von Schiffscontainern: genormten Kisten, die jeder Kran, jedes Schiff und jeder Lkw handhaben kann, ohne zu wissen, was darin ist.

Das Paket ist ein Container-Image: eine schreibgeschützte, versionierte Datei, gebaut nach einem kurzen Rezept (bei Docker ein `Dockerfile`). Eine laufende Kopie eines Images ist ein Container. Docker, 2013 veröffentlicht, machte das für Entwicklungsteams im Alltag praktikabel, und sein Image-Format ist heute ein offener Standard, den die Open Container Initiative pflegt, sodass andere Werkzeuge dieselben Images bauen und ausführen können.

Der entscheidende Unterschied zu einer VM ist, was geteilt wird. Eine VM bildet Hardware nach und führt ein ganzes Gastbetriebssystem aus. Ein Container ist ein isolierter Prozess: Er teilt sich den *Kernel* des Host-Betriebssystems (den Kern des Betriebssystems, der Hardware, Arbeitsspeicher und Prozesse steuert) mit anderen Containern, und die Isolationsfunktionen des Kernels geben ihm eine eigene Sicht auf Dateien, Prozesse und Netzwerk. Container starten daher in Sekunden oder schneller und belegen Megabytes statt Gigabytes, sodass

Dutzende dort laufen können, wo nur wenige VMs Platz hätten. Der Preis ist eine schwächere Isolation, weil jeder Container von dem einen gemeinsamen Kernel abhängt; deshalb betreiben Cloud-Anbieter die Container ihrer Kunden häufig innerhalb von VMs und kombinieren so beides.

Virtuelle Maschinen und Container im Vergleich



VMs virtualisieren die Hardware, und jede trägt ein Betriebssystem; Container virtualisieren das Betriebssystem und teilen sich seinen Kernel.

```
# Start from an image that already has Python installed
FROM python:3.12-slim
WORKDIR /app
# Bake the libraries into the image
COPY requirements.txt .
RUN pip install -r requirements.txt
# Add the application code
COPY . .
EXPOSE 8000
CMD ["python", "-m", "uvicorn", "main:app", "--host", "0.0.0.0", "--port", "8000"]
```

Ein vollständiges Dockerfile für eine kleine Python-Web-API. Der Befehl docker build macht aus diesem Rezept ein Image.

Einen Container auszuführen ist einfach. Hunderte auf Dutzenden Maschinen zu betreiben, abgestürzte neu zu starten, bei steigendem Verkehr Kopien hinzuzufügen und neue Versionen ohne Ausfallzeit auszurollen ist es nicht. Diese Aufgabe übernimmt ein Orchestrator, von denen Kubernetes der bekannteste ist; er wird in Container und Orchestrierung behandelt.

Die Cloud: Rechenleistung minutenweise mieten

Cloud-Computing bedeutet, Rechenressourcen über das Internet bei Bedarf von einem Anbieter zu mieten, der riesige Rechenzentren betreibt: Amazon Web Services (AWS), Microsoft Azure und Google Cloud sind die größten, daneben gibt es viele weitere. Das US National Institute of Standards and Technology (NIST) veröffentlichte 2011 die meistzitierte Definition, die Special Publication 800-145. Sie nennt fünf wesentliche Merkmale: Selbstbedienung nach Bedarf (Sie legen Ressourcen selbst an, über eine Website oder eine API, ohne irgendwo anzurufen), breiter Netzwerkzugang, Ressourcenbündelung (viele Kunden teilen sich die Hardware des Anbieters), schnelle Elastizität (die Kapazität wächst und schrumpft rasch) und gemessene Dienstleistung (Sie zahlen für das, was Sie nutzen).

Die Analogie ist der Strom. Fabriken betrieben einst eigene Generatoren; heute hängen sie am Netz und zahlen pro Kilowattstunde. Mit der Cloud kann ein Start-up für einen Nachmittag hundert Server mieten, um einen Launch zu testen, und hört auf zu zahlen, sobald es sie zurückgibt. Das ist Pay-as-you-go-Abrechnung. Sie macht aus einer großen Anfangsinvestition laufende Kosten und belohnt Systeme, die Kapazität abschalten können, wenn sie nicht gebraucht wird. Sie schafft aber auch ein neues Risiko: Kosten, die unbemerkt wachsen, weil nichts jemanden daran hindert, Ressourcen anzulegen und sie zu vergessen.

IaaS, PaaS, SaaS und Serverless: Wie viel verwalten Sie selbst?

Cloud-Dienste unterscheiden sich vor allem darin, wie viel vom Stack der Anbieter für Sie verwaltet. NIST definierte drei Servicemodelle; ein viertes, Serverless, wurde populär, nachdem AWS 2014 seinen Dienst Lambda gestartet hatte. Eine beliebte Eselsbrücke ist Pizza: Sie können sie zu Hause machen, eine fertige zum Aufbacken kaufen, eine liefern lassen oder essen gehen. Jeder Schritt gibt mehr Arbeit an jemand anderen ab und nimmt Ihnen einige Wahlmöglichkeiten.

Cloud-Servicemodelle im Vergleich

Modell	Sie verwalten	Der Anbieter verwaltet	Beispiele	Pizza-Analogie
On-Premises	Alles: Gebäude, Hardware, OS, Laufzeitumgebung, Anwendung, Daten	Nichts	Eigene Server im eigenen Rechenzentrum	Zu Hause von Grund auf selbst gemacht
IaaS (Infrastructure as a Service)	Betriebssystem, Laufzeitumgebung, Anwendung, Daten	Hardware, Virtualisierung, Netzwerk, Gebäude	Amazon EC2, Azure Virtual Machines, Google Compute Engine	Zum Aufbacken: Sie backen sie im eigenen Ofen
PaaS (Platform as a Service)	Anwendungscode und Daten	Alles darunter, einschließlich OS-Patches und Skalierung	Heroku, Azure App Service, Google App Engine	Lieferung: Sie stellen Tisch und Getränke
Serverless / FaaS (Function as a Service)	Einzelne Funktionen, ausgelöst durch Ereignisse	Alle Server, Skalierung auf null im Leerlauf, Abrechnung pro Aufruf	AWS Lambda, Azure Functions, Google Cloud Run functions	Pizza stückweise, bezahlt pro Bissen
SaaS (Software as a Service)	Ihre Daten, Ihre Nutzer und deren Einstellungen	Die gesamte Anwendung	Microsoft 365, Gmail, Salesforce	Essen im Restaurant

„Serverless“ heißt nicht, dass es keine Server gibt; es heißt, dass Sie sie nie sehen oder verwalten. Sie laden eine Funktion hoch, legen fest, welche Ereignisse sie auslösen (eine HTTP-Anfrage, eine eintreffende Datei, eine Nachricht in einer Queue), und der Anbieter führt so viele Kopien aus wie nötig und rechnet pro Ausführung und pro Millisekunde ab. Das ist günstig bei sprunghaftem oder geringem Verkehr und erfordert fast keinen Betriebsaufwand. Die Kosten zeigen sich an anderer Stelle: Der erste Aufruf nach einer ruhigen Phase kann langsam sein, während der Anbieter die Funktion startet (ein *Kaltstart*), es gibt Grenzen dafür, wie lange eine Funktion laufen darf, und bei gleichmäßig hohem Verkehr können die Kosten die eines einfachen Servers übersteigen. Kapitel 6 behandelt Serverless-Architekturen ausführlich.

Zwischen diesen Modellen liegen die Managed Services: Datenbanken, Message Queues, Caches, Objektspeicher, Suchmaschinen und vieles mehr, die der Anbieter installiert, betreibt, patcht, sichert und skaliert. Eine verwaltete Datenbank zu nutzen, statt selbst eine auf einer VM zu installieren, verlagert viel Betriebsarbeit zum Anbieter. Die Abwägung, behandelt in FinOps und Lock-in, besteht darin, dass ein System umso schwerer umzuziehen ist, je mehr anbieterspezifische Dienste es nutzt.

ACHTUNG

Das Modell der geteilten Verantwortung

Der Umzug in die Cloud verlagert nicht die gesamte Verantwortung. Die Anbieter sagen klar, dass sie die Cloud selbst absichern (Gebäude, Hardware, die Virtualisierungsschicht) und der Kunde absichert, was er hineinlegt: seine Daten, seine Zugriffsrechte, seine Konfiguration und bei IaaS das Betriebssystem. Viele vielbeachtete Datenlecks in der Cloud waren kein Versagen der Anbieter, sondern vom Kunden falsch konfigurierte Speicher, die offen im Internet lagen.

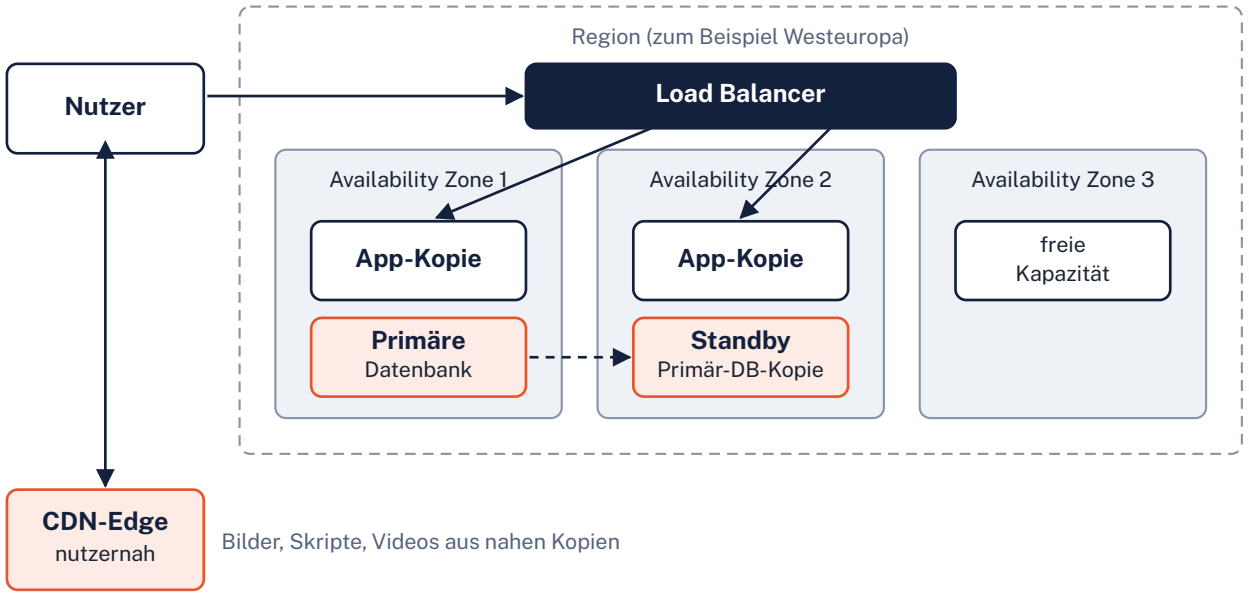
Regionen, Availability Zones und Content Delivery Networks

Cloud-Anbieter gruppieren ihre Rechenzentren nach Geografie. Eine Region ist ein geografisches Gebiet (etwa Irland, Frankfurt oder Virginia) mit einer eigenen, getrennten Gruppe von Rechenzentren, isoliert von anderen Regionen. Jede Region ist in mehrere Availability Zones unterteilt: physisch getrennte Gruppen aus einem oder mehreren Rechenzentren, jede mit unabhängiger Stromversorgung, Kühlung und Vernetzung, typischerweise Kilometer voneinander entfernt, aber nah genug für schnelle Verbindungen untereinander. Ein Brand, ein Hochwasser oder ein Stromausfall sollte höchstens eine Zone lahmlegen.

Das gibt Architekten eine Auswahl an Schutzstufen. Kopien einer Anwendung in zwei oder drei Zonen einer Region zu betreiben übersteht den Verlust eines Rechenzentrums bei kaum zusätzlicher Latenz; der Betrieb in zwei Regionen übersteht den Verlust einer ganzen Region, macht es aber viel schwieriger, die Daten synchron zu halten. Regionen sind auch für Recht und Latenz wichtig: Datenschutzvorschriften können verlangen, dass personenbezogene Daten in einem bestimmten Rechtsraum bleiben, und Nutzer werden aus einer nahen Region schneller bedient, weil kein Netzwerk die Lichtgeschwindigkeit schlagen kann (siehe Wie Computer kommunizieren).

Ein Content Delivery Network (CDN) treibt das weiter für Inhalte, die sich nicht pro Nutzer ändern, etwa Bilder, Stylesheets, Skripte und Videos. Es hält Kopien auf Servern an Hunderten Standorten, sogenannten Edge-Standorten, sodass ein Nutzer in Nairobi die Produktfotos des Shops aus einer nahen Stadt abruft statt aus einem Rechenzentrum in Europa. Die Seite lädt schneller, und die eigenen Server des Shops bearbeiten weit weniger Anfragen. Viele CDNs fangen auch Angriffsverkehr ab und können kleine Codestücke an der Edge ausführen.

Innerhalb einer Region



Kopien der Anwendung in zwei Availability Zones halten den Dienst am Laufen, wenn eine Zone ausfällt, während ein CDN statische Inhalte von Standorten nahe den Nutzern ausliefert.

Vertikal und horizontal skalieren

Wenn ein System mehr Arbeit bewältigen muss, gibt es zwei grundlegende Schritte. Vertikale Skalierung (Scale-up) bedeutet eine größere Maschine: mehr Kerne, mehr Arbeitsspeicher. Sie erfordert keine Änderung an der Software, was ihr großer Vorteil ist, aber es gibt eine größte Maschine, die man kaufen kann, nahe der Spitze wird es unverhältnismäßig teuer, und eine Maschine ist immer noch ein einzelner Ausfallpunkt. Horizontale Skalierung (Scale-out) bedeutet mehr Maschinen, die sich die Arbeit teilen, meist hinter einem Load Balancer, der die Anfragen auf sie verteilt und keine mehr an ausgefallene schickt. Sie reicht viel weiter und übersteht Maschinenausfälle, aber nur, wenn die Software dafür ausgelegt ist: Ein Server darf zum Beispiel die Session eines Nutzers nicht nur im eigenen Arbeitsspeicher halten oder hochgeladene Dateien nur auf der eigenen Festplatte speichern, wenn die nächste Anfrage des Nutzers bei einem anderen Server landen kann.

Die Cloud macht horizontale Skalierung attraktiv, weil eine zusätzliche Maschine nur einen API-Aufruf kostet und *Autoscaling* Kopien automatisch hinzufügen und entfernen kann, wenn sich die Last ändert. Datenbanken sind meist am schwersten horizontal zu skalieren, weshalb Kapitel 7 einen Abschnitt der Replikation und Partitionierung widmet.

AUSGEARBEITETES BEISPIEL

Ausgearbeitetes Beispiel: eine Ticketbuchungsseite hosten

Eine kleine Theaterkette verkauft Tickets online. Die meiste Zeit der Woche ist wenig los, aber wenn der Vorverkauf für eine beliebte Vorstellung beginnt, kommen innerhalb von Minuten zehnmal so viele Menschen.

Eine gemietete VM (IaaS)	Einfach und günstig. Das Team patcht das OS. Am Vorverkaufstag ist die einzelne Maschine überlastet, und wenn sie ausfällt, ist die Seite offline.
Größere VM	Vertikale Skalierung verschafft Reserven ohne Codeänderung, aber das Theater zahlt die ganze Woche für die große Maschine, um ein paar geschäftige Stunden abzudecken.
App auf einer PaaS, zwei Zonen, Autoscaling	Kopien der App laufen in zwei Availability Zones hinter einem Load Balancer. Am Vorverkaufstag fügt die Plattform automatisch Kopien hinzu und entfernt sie danach wieder. Voraussetzung ist, dass die App keine Session-Daten im lokalen Arbeitsspeicher hält.
Verwaltete Datenbank	Der Anbieter betreibt eine Primärdatenbank in einer Zone und ein Standby in einer anderen, mit automatischen Backups. Das Schema und die Indizes entwirft weiterhin das Team (siehe Daten speichern).
CDN für Bilder und Skripte	Saalpläne und Fotos der Vorstellungen werden von Edge-Standorten ausgeliefert, sodass sich die App-Server auf die Buchungen konzentrieren können.

Der endgültige Entwurf kostet in der Einrichtung mehr als eine VM, im Betrieb aber weniger als eine dauerhaft überdimensionierte Maschine, und er übersteht sowohl eine Lastspitze als auch den Verlust eines Rechenzentrums.

Jeder Schritt tauschte Kontrolle gegen Bequemlichkeit: Das Team entschied, welche Arbeit es behält und welche es an den Anbieter abgibt.

Warum das für die Architektur wichtig ist

Wo Software läuft, ist heute eine Architekturentscheidung und kein nachträglicher Gedanke. Die Wahl zwischen VMs, Containern, Plattformen und Funktionen prägt die Kosten, wie schnell das Team Releases ausliefern kann, wie das System skaliert und was passiert, wenn ein Rechenzentrum ausfällt. Regionen und Zonen machen aus Verfügbarkeitszielen konkrete Deployment-Pläne; Managed Services tauschen Betriebsarbeit gegen die Abhängigkeit von einem Anbieter. Kapitel 9 entwickelt Skalierungsstrategien, und Kapitel 10 behandelt Cloud-Service- und Bereitstellungsmodelle, Container und Infrastruktur in professioneller Tiefe.

PRÜFEN SIE IHR VERSTÄNDNIS

▼ Warum starten Container schneller und brauchen weniger Platz als virtuelle Maschinen?

Ein Container bootet kein eigenes Betriebssystem: Er ist ein isolierter Prozess, der den Kernel des Hosts mitbenutzt, und enthält nur die Anwendung und ihre Bibliotheken. Eine VM enthält ein vollständiges Gastbetriebssystem und bootet es.

▼ Ein Team mietet eine VM in der Cloud und installiert darauf seine eigene Datenbank. Wer ist dafür verantwortlich, das Betriebssystem zu patchen?

Das Team. Bei IaaS verwaltet der Anbieter die Hardware und die Virtualisierung, und der Kunde verwaltet das Betriebssystem und alles darüber.

▼ Vor welchem Ausfall schützt der Betrieb in zwei Availability Zones, und wovor schützt er nicht?

Er schützt vor dem Verlust eines Rechenzentrums oder einer Zone (Strom, Brand, Netzwerk). Er schützt nicht vor dem Verlust der ganzen Region und nicht vor Softwarefehlern und fehlerhaften Daten, die jede Kopie betreffen.

▼ Warum könnte sich eine Anwendung, die auf einem Server einwandfrei läuft, fehlerhaft verhalten, wenn sie hinter einem Load Balancer horizontal skaliert wird?

Sie hält womöglich Zustand wie Nutzer-Sessions im eigenen Arbeitsspeicher; schickt der Load Balancer die nächste Anfrage an eine andere Kopie, kennt diese Kopie den Nutzer nicht.

QUELLEN

- NIST SP 800-145: The NIST Definition of Cloud Computing (2011): https://csrc.nist.gov/pubs/sp/800/145/final
- Docker: What is a container?: https://www.docker.com/resources/what-container/
- AWS documentation: Regions and Availability Zones: https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using-regions-availability-zones.html
- Microsoft Learn: What are availability zones?: https://learn.microsoft.com/en-us/azure/reliability/availability-zones-overview
- MDN Web Docs: CDN (glossary): https://developer.mozilla.org/en-US/docs/Glossary/CDN

ABSCHNITT 0.8 VON 0.8 · EINSTEIGER · 20 MIN. LESEZEIT

Anforderungen, Versionskontrolle, Tests, Auslieferung und der Weg einer Anfrage

DIE IDEE IN KÜRZE

Software zu entwickeln ist Teamsport mit einer sorgfältigen Routine. Das Team schreibt in kurzen Stories auf, was die Nutzer brauchen, hält jede Änderung am Code in einer gemeinsamen Historie fest, damit nichts verloren geht, lässt jede Änderung von einer Kollegin oder einem Kollegen prüfen und führt automatische Tests aus, die wie ein Sicherheitsnetz wirken. Danach durchlaufen Änderungen Übungskopien des Systems, bevor sie das echte erreichen, das die Kunden nutzen. Am Ende des Abschnitts wird ein einziges Tippen auf eine Schaltfläche „Bestellen“ durch jeden Teil eines Essensliefersystems verfolgt.

WAS SIE WISSEN MÜSSEN

- Anforderungen beschreiben, was ein System tun muss und wie gut; User Stories (Als …, möchte ich …, damit …) mit Akzeptanzkriterien sind eine verbreitete, schlanke Form, funktionale Bedürfnisse festzuhalten.
- Git zeichnet jede Änderung als Commit auf; Branches halten laufende Arbeit getrennt, und Pull Requests bringen Code-Review und automatische Prüfungen vor einem Merge in main.
- Die Testpyramide empfiehlt viele schnelle Unit-Tests, weniger Integrationstests und wenige langsame End-to-End-Tests.
- Änderungen durchlaufen Entwicklungs-, Test- und Staging-Umgebungen vor der Produktion; je näher Staging an der Produktion ist, desto mehr Probleme fängt es ab.
- Continuous Integration prüft jede Änderung automatisch; Continuous Delivery hält Software jederzeit releasefähig; Continuous Deployment gibt jede Änderung, die alle Prüfungen besteht, automatisch frei.
- Wasserfall, Scrum und Kanban organisieren Arbeit unterschiedlich; DevOps ist eine Kultur der gemeinsamen Verantwortung für Entwicklung und Betrieb von Software.
- Eine einzige Anfrage in einem echten System durchläuft DNS, HTTPS, einen Load Balancer, Services, eine Datenbanktransaktion, Drittanbieter-APIs und Queues, und zu entscheiden, wie diese Teile angeordnet sind, ist Softwarearchitektur.

Von der Idee zur Anforderung

Software beginnt mit einem Bedürfnis: Ein Restaurant möchte Bestellungen online annehmen, ein Krankenhaus möchte, dass Patienten ihre Termine selbst buchen. Dieses Bedürfnis in etwas zu übersetzen, das ein Team bauen kann, ist die Arbeit mit Anforderungen. Manche Anforderungen beschreiben, was das System tun muss („ein Kunde kann per Karte bezahlen“); andere beschreiben, wie gut es das tun muss („der Bezahlvorgang ist für 95 % der Kunden innerhalb von zwei Sekunden abgeschlossen“). Kapitel 2 zeigt, dass die zweite Art, die Qualitätsmerkmale, die meisten Architekturentscheidungen treibt.

Viele Teams halten funktionale Bedürfnisse als User Stories fest: kurze Beschreibungen aus Sicht der Nutzer. Kent Beck führte sie Ende der 1990er-Jahre im Extreme Programming ein, und ein Team beim Londoner Unternehmen Connextra entwarf 2001 die Vorlage, die heute die meisten verwenden:

```
As a hungry customer,
I want to reorder a meal I have ordered before,
so that I can eat sooner without searching the menu again.

Acceptance criteria
- My last 10 orders appear on the home screen.
- Tapping "Reorder" fills the basket with the same items.
- Items that are no longer available are flagged, not silently dropped.
```

Eine User Story mit Akzeptanzkriterien. Die Kriterien sind das, was Tester und automatisierte Tests prüfen werden.

Eine Story ist bewusst klein und unvollständig. Ron Jeffries beschrieb sie als Karte, Gespräch und Bestätigung (Card, Conversation, Confirmation): Die Karte ist eine Erinnerung, die Details werden im Gespräch zwischen denen geklärt, die das Feature wollen, und denen, die es bauen, und die Akzeptanzkriterien bestätigen, wann es fertig ist. Fachleute sind uneins darüber, wie viel vorab aufgeschrieben werden sollte. Teams in regulierten Branchen wie Medizinprodukten und Luftfahrt müssen formale, nachverfolgbare Anforderungen führen; Produktteams bevorzugen oft schlankere Stories, die kurz vor Arbeitsbeginn verfeinert werden. Beides ist in seinem Kontext legitim.

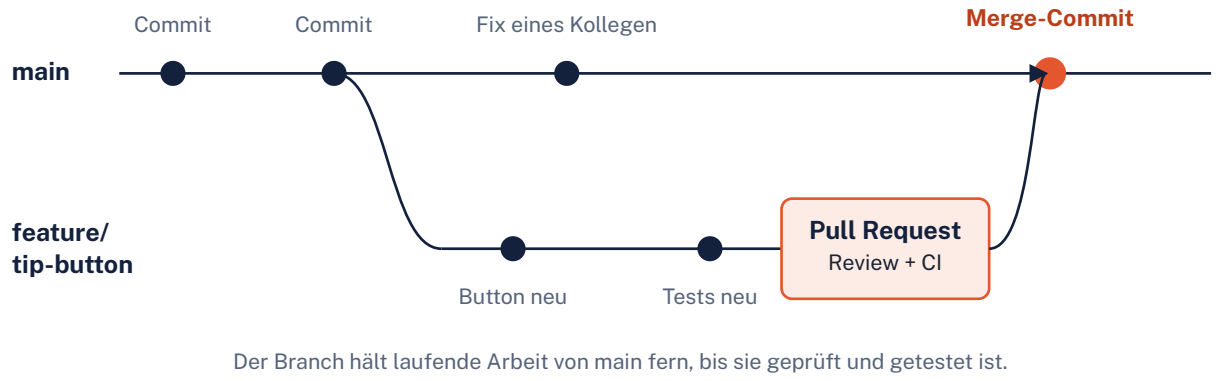
Versionskontrolle mit Git

Quellcode ist das wertvollste gemeinsame Dokument des Teams, und viele Menschen ändern ihn täglich. Versionskontrolle zeichnet jede Änderung auf, wer sie vorgenommen hat und warum, lässt Menschen parallel arbeiten, ohne sich gegenseitig zu überschreiben, und erlaubt

es, jeden früheren Stand wiederherzustellen. Fast jedes Team nutzt heute Git, das Linus Torvalds 2005 für die Entwicklung des Linux-Kernels geschaffen hat, meist zusammen mit einem Hosting-Dienst wie GitHub, GitLab oder Bitbucket.

Wenige Begriffe decken den größten Teil des Alltags ab. Ein Commit ist ein gespeicherter Schnappschuss einer Reihe von Änderungen mit einer erklärenden Nachricht („Add tip button to checkout“). Ein Branch ist ein eigener Arbeitsstrang: Eine Entwicklerin legt einen Branch von `main` an, macht dort Commits, ohne jemanden zu stören, und *mergt* den Branch nach Abschluss zurück, wobei die beiden Historien zusammengeführt werden. Haben zwei Personen dieselben Zeilen geändert, meldet Git einen *Merge-Konflikt*, den ein Mensch auflösen muss.

Auf Hosting-Diensten läuft das Mergen meist über einen Pull Request (bei GitLab Merge Request genannt): eine Webseite mit der vorgeschlagenen Änderung, auf der automatische Prüfungen laufen und Kolleginnen und Kollegen ein Code-Review durchführen, also die Änderung lesen, Fragen stellen und Verbesserungen vorschlagen, bevor sie sie freigeben. Reviews finden Fehler, aber ihr größerer Nutzen ist, Wissen zu verbreiten (mindestens zwei Personen verstehen jede Änderung) und das Design konsistent zu halten. Architekten beteiligen sich oft an Reviews von Änderungen, die Struktur oder Schnittstellen betreffen.



Der Branch hält laufende Arbeit von main fern, bis sie geprüft und getestet ist.

Branch, Commit, Review, Merge: der alltägliche Zyklus der Versionskontrolle.

```
git switch -c feature/tip-button     # create a branch and move onto it
# ... edit files ...
git add checkout.py test_checkout.py
git commit -m "Add tip button to checkout"
git push -u origin feature/tip-button  # upload; then open a pull request
# after review and green checks, the branch is merged into main
```

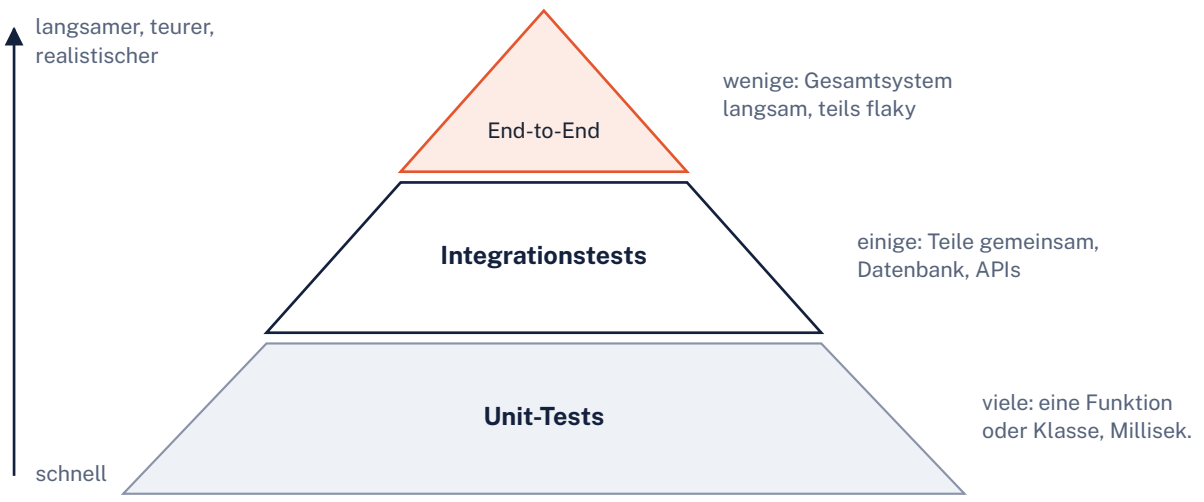
Der alltägliche Git-Zyklus auf der Kommandozeile.

Beim Branching gehen Teams unterschiedlich vor. Manche führen langlebige Branches pro Feature oder Release; andere praktizieren *Trunk-Based Development*, mergen kleine Änderungen mindestens täglich in `main` und verbergen unfertige Features hinter Schaltern, sogenannten Feature-Flags. Die Forschung des DORA-Programms bringt kurzlebige Branches mit besserer Delivery-Performance in Verbindung, aber die richtige Wahl hängt auch davon ab, wie ein Team Releases ausliefert.

Testen: die Pyramide

Jede Änderung riskiert, etwas kaputt zu machen, das vorher funktioniert hat; das nennt man eine *Regression*. Automatisierte Tests sind Programme, die andere Programme prüfen, daher lassen sie sich bei jeder Änderung in Minuten ausführen statt in Tagen durch Menschen. Es gibt sie in Schichten. Ein Unit-Test prüft ein kleines Stück, etwa eine Funktion, isoliert und läuft in Millisekunden. Ein Integrationstest prüft, ob Teile zusammenarbeiten: der Bestellcode mit einer echten Datenbank oder der Shop mit der Testumgebung eines Zahlungsanbieters. Ein End-to-End-Test steuert das ganze System so, wie es ein Nutzer täte, zum Beispiel indem er sich in der App von der Speisekarte bis zur Bestellbestätigung durchtippt.

Mike Cohns Testpyramide, veröffentlicht 2009 in seinem Buch *Succeeding with Agile*, schlägt die Proportionen vor: viele Unit-Tests, weniger Integrationstests und wenige End-to-End-Tests. Die höheren Schichten sind realistischer, aber langsamer, teurer in der Wartung und anfälliger für *Flaky*-Fehlschläge, die aus Gründen kommen und gehen, die nichts mit der Änderung zu tun haben. Manche Fachleute plädieren für eine andere Form, etwa mehr Integrationstests bei Systemen, die vor allem Bindeglied zwischen Services sind; einig sind sich alle im Grundsatz, die meisten Probleme mit schnellen, zuverlässigen, fokussierten Tests abzufangen.



Die Form ist ein Richtwert, kein Gesetz: Die meisten Prüfungen sollten schnell und fokussiert sein.

Viele schnelle Unit-Tests, weniger Integrationstests und wenige End-to-End-Tests geben schnelles Feedback, ohne an Realismus zu verlieren.

```
def delivery_fee(basket_total, distance_km):
    if basket_total ⩾ 25:
        return 0.0                      # free delivery on larger orders
    return round(1.50 + 0.40 * distance_km, 2)

def test_free_delivery_over_threshold():
    assert delivery_fee(30.00, 5) == 0.0

def test_fee_grows_with_distance():
    assert delivery_fee(10.00, 5) == 3.50
```

Zwei Unit-Tests, geschrieben mit pytest, für eine Funktion zur Liefergebühr. Sie laufen in Millisekunden und legen die Regeln fest.

Umgebungen: von der Entwicklung bis zur Produktion

Code durchläuft auf seinem Weg zu den Kunden mehrere Umgebungen, jede eine eigene Kopie des Systems mit eigenen Servern, Daten und Einstellungen. Ziel ist es, Probleme dort zu finden, wo sie billig und harmlos sind, bevor sie die Produktion erreichen, die Live-Umgebung, auf die echte Nutzer angewiesen sind.

Umgebung	Wer sie nutzt	Daten	Zweck
Entwicklung	Eine Entwicklerin oder ein Entwickler, auf dem Laptop oder in einem persönlichen Cloud-Bereich	Klein, erfunden	Code schnell schreiben und ausprobieren.
Test (oder Integration)	Das Team und die automatisierte Pipeline	Erfunden oder anonymisiert	Automatisierte und explorative Tests auf den zusammengeführten Änderungen ausführen.
Staging (oder Pre-Production)	Tester, Product Owner, manchmal ausgewählte Kunden	In Form und Umfang produktionsähnlich, nie rohe personenbezogene Daten ohne Schutzmaßnahmen	Das Release auf einer Kopie proben, die der Produktion so nah wie möglich kommt.
Produktion	Echte Kunden	Echt	Das Geschäft betreiben. Änderungen kommen nur über die Pipeline.

Je stärker Umgebungen von der Produktion abweichen, desto mehr Probleme rutschen durch: Eine Testdatenbank mit hundert Zeilen verrät nicht den fehlenden Index, durch den eine Abfrage bei zehn Millionen Zeilen vierzig Sekunden dauert. Umgebungen ähnlich zu halten ist leichter, wenn sie aus denselben Container-Images und derselben per Skript erzeugten Konfiguration gebaut werden, ein Thema von Infrastructure as Code.

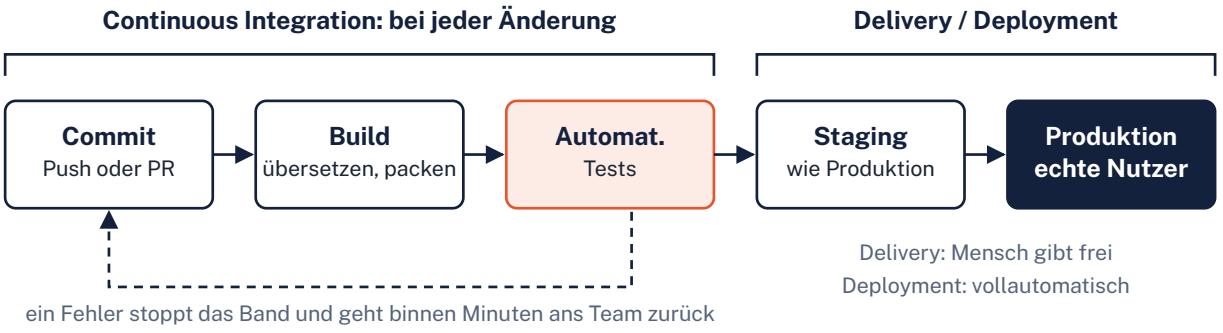
Continuous Integration, Delivery und Deployment

Continuous Integration (CI) ist die Praxis, die Arbeit aller häufig, mindestens täglich, in die gemeinsame Hauptlinie zu mergen, wobei ein automatischer Build- und Testlauf jede Änderung prüft. Sie ging in den 1990er-Jahren aus dem Extreme Programming hervor. Ihr Sinn ist, dass Integrationsprobleme innerhalb von Minuten gefunden werden, solange die Änderung der Autorin oder dem Autor noch frisch im Kopf ist, statt Wochen später in einer mühsamen „Integrationsphase“. Ein Team, das CI praktiziert, behandelt einen fehlgeschlagenen Build als Notfall: Ihn zu reparieren geht jeder neuen Arbeit vor.

Continuous Delivery (CD) erweitert CI so, dass die Software jederzeit releasefähig ist: Jede Änderung, die die Pipeline durchläuft, könnte auf Knopfdruck in Produktion gehen, und ein Release wird zu einer routinemäßigen Geschäftsentscheidung statt zu einem riskanten Ereignis. Den Ansatz beschrieben Jez Humble und David Farley 2010 in ihrem Buch *Continuous*

Delivery. Continuous Deployment geht einen Schritt weiter und gibt jede erfolgreiche Änderung automatisch frei, ohne menschliche Freigabe; Teams, die das tun, deployen unter Umständen viele Male am Tag.

Die Automatisierung, die eine Änderung vom Commit bis in die Produktion trägt, ist eine Pipeline. Weil jeder Schritt automatisiert und jedes Mal identisch ist, hängen Releases nicht mehr davon ab, dass sich eine Person um Mitternacht an eine Checkliste erinnert.



Continuous Integration prüft jede Änderung automatisch; Delivery und Deployment bringen den erfolgreichen Build in die Produktion.

Checkliste: bevor eine Änderung die Produktion erreicht

Was der Prozess eines erfahrenen Teams bei jeder Änderung prüft, größtenteils automatisch.

- [ ] **Die Änderung ist mit einer Story oder einem Issue verknüpft**
  Jeder kann später herausfinden, warum sie vorgenommen wurde.
- [ ] **Sie ist klein**
  Kleine Änderungen sind leichter zu prüfen, zu testen, zu deployen und notfalls rückgängig zu machen.
- [ ] **Jemand aus dem Team hat sie geprüft**
  Über einen Pull Request, mit beantworteten Fragen und erledigten Kommentaren.
- [ ] **Automatisierte Tests bestehen auf jeder Ebene**
  Unit- und Integrationstests in der CI und die End-to-End-Prüfungen, die den betroffenen Ablauf abdecken.
- [ ] **Sie lief in Staging**
  Derselbe Build, derselbe Konfigurationsmechanismus und dieselben Deployment-Schritte wie in der Produktion.
- [ ] **Es gibt einen Weg zurück**
  Ein Rollback oder ein Feature-Flag kann sie schnell abschalten, wenn etwas schiefgeht.
- [ ] **Jemand bemerkt es, wenn sie etwas kaputt macht**
  Monitoring und Alerts decken ab, was die Änderung berührt (Kapitel 9, Observability).

Arbeitsweisen: Wasserfall, Scrum und Kanban

Teams entscheiden auch, wie sie die Arbeit selbst organisieren. Bei einem sequenziellen Vorgehen, oft *Wasserfall* genannt, durchläuft das Projekt Phasen der Reihe nach: Anforderungen, Entwurf, Implementierung, Test, Deployment. Jede Phase endet, bevor die nächste beginnt. Der Name wird meist auf Winston Royces Aufsatz von 1970 zurückgeführt, obwohl Royce das Wort nicht verwendete und sogar davor warnte, dass ein einziger Durchlauf durch die Phasen riskant sei. Sequenzielle Pläne passen nach wie vor zu Arbeit, bei der die Anforderungen feststehen und Änderungen sehr teuer sind, etwa bei manchen regulierten oder hardwareabhängigen Projekten. Ihre Schwäche für die meiste Software ist, dass Nutzer bis spät nichts sehen, sodass Missverständnisse dann auftauchen, wenn sie am teuersten zu beheben sind.

Agile Ansätze, benannt im Agilen Manifest von 2001, liefern funktionierende Software in kurzen Zyklen und passen Pläne an, während sie dazulernen. Scrum ist das am weitesten verbreitete Framework: Ein kleines Team arbeitet in festen Zyklen namens Sprints von höchstens einem Monat, die jeweils mit einer Planung beginnen und mit einem Review der Arbeit und einer Retrospektive zur Zusammenarbeit des Teams enden. Der Scrum Guide 2020 definiert drei Verantwortlichkeiten: den Product Owner (entscheidet, was am wertvollsten ist), den Scrum Master (hilft dem Team, Scrum wirksam einzusetzen) und die Developers (erledigen die Arbeit). Kanban, für Software aus dem Produktionssystem von Toyota übernommen, hat keine festen Zyklen: Arbeitspakete fließen über ein sichtbares Board (zu erledigen, in Arbeit, Review, erledigt), und das Team begrenzt, wie viele davon gleichzeitig in Arbeit sein dürfen, damit Arbeit fertig wird, bevor neue begonnen wird.

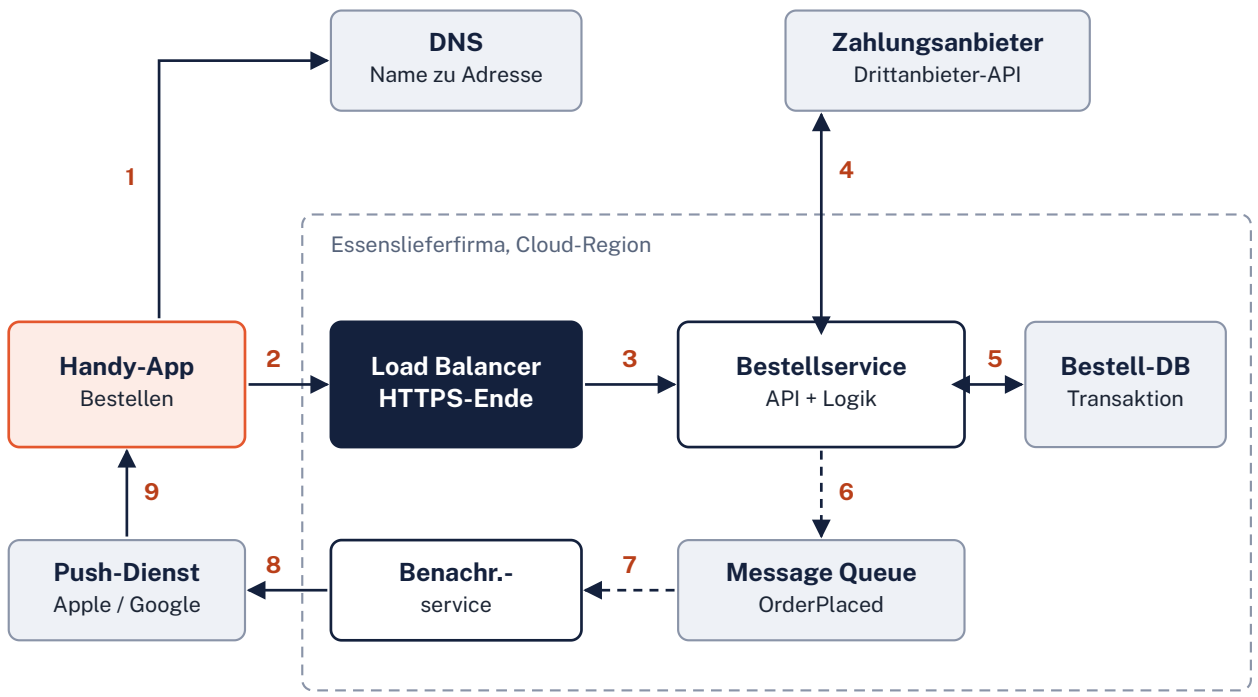
Keiner dieser Ansätze passt für jedes Team, und viele Organisationen mischen sie: Scrum für Produktteams, Kanban für Support und Betrieb, ein eher sequenzieller Plan rund um eine feste regulatorische Frist. Einen Prozess „agil“ zu nennen macht ihn nicht agil; entscheidend ist, ob funktionierende Software oft genug echte Nutzer erreicht, um von ihnen zu lernen.

DevOps: die Verantwortung eines Teams vom Code bis zum Kunden

Traditionell schrieben Entwickler Software, und ein separates Betriebsteam betrieb sie, mit gegensätzlichen Anreizen: Entwickler wurden für Veränderung belohnt, der Betrieb für Stabilität. DevOps ist die Bewegung, benannt auf der ersten DevOpsDays-Konferenz 2009 in Gent, die beide zusammenbringt. Sie ist vor allem eine Kultur der gemeinsamen Verantwortung („you build it, you run it“), gestützt durch Praktiken wie CI/CD, Infrastructure as Code, Monitoring und schuldzuweisungsfreie Auswertung von Vorfällen. Sie ist weder eine Berufsbezeichnung noch ein Werkzeug, auch wenn beides unter dem Namen verkauft wird. Das DORA-Forschungsprogramm ist vor allem dafür bekannt, Delivery-Performance mit vier Schlüsselmetriken zu messen: wie oft ein Team deployt, wie lange eine Änderung bis in die Produktion braucht, wie oft Änderungen fehlschlagen und wie schnell der Dienst nach einem Ausfall wiederhergestellt ist.

Ausgearbeitetes Beispiel: der Weg eines Tippens

Alles in diesem Kapitel kommt in einem einzigen Moment zusammen. Eine Kundin hat in einer Essensliefer-App einen Warenkorb gefüllt und tippt auf **Bestellen**. Binnen ein oder zwei Sekunden zeigt der Bildschirm „Bestellung bestätigt“, und zwanzig Minuten später vibriert das Telefon: „Ihr Fahrer ist unterwegs“. Hier ist, was Schritt für Schritt geschieht, mit dem Konzept hinter jedem Schritt und dem Ort, an dem es eingeführt wurde.



Ein Tippen durchläuft das Telefon, das Internet, die Cloud-Region des Unternehmens und zwei externe Anbieter; die Nummern entsprechen den Schritten unten.

Schritt für Schritt

1 **0. Die App baut die Anfrage**
Die App ist Software auf dem Telefon (Abschnitt 0.1). Sie holt den Warenkorb aus ihren Datenstrukturen im Arbeitsspeicher (Abschnitt 0.2) und macht daraus einen JSON-Body: Restaurant, Artikel, Lieferadresse und einen eindeutigen Bestellschlüssel, der auf dem Telefon erzeugt wird.

2 **1. DNS**
Die App braucht die Adresse von `api.fooddelivery.example`. Das Telefon fragt einen DNS-Resolver, der eine IP-Adresse zurückgibt, oft aus einem Cache (Abschnitt 0.4).

3 **2. HTTPS**
Die App öffnet eine TCP-Verbindung und eine TLS-Sitzung, prüft dabei das Zertifikat des Servers und sendet dann `POST /v1/orders` mit einem Access Token, das belegt, welcher Kunde angemeldet ist (Abschnitte 0.4 und 0.6).

4 **3. Load Balancer**
Die Anfrage erreicht einen Load Balancer in der Cloud-Region des Unternehmens. Er entschlüsselt HTTPS und leitet die Anfrage an eine von mehreren funktionsfähigen Kopien des Bestellservice weiter, die in Containern über mehrere Availability Zones verteilt laufen (Abschnitt 0.7).

5 **4. API und Zahlung**
Der Bestellservice ist über seine REST-API erreichbar. Sein in Module gegliederter Code (Abschnitt 0.3) validiert den Warenkorb und ruft dann die Web-API des Zahlungsanbieters auf, um die Karte zu autorisieren; er wartet synchron mit einem Timeout, weil der Kunde erst „bestätigt“ sehen darf, wenn die Zahlung genehmigt ist (Abschnitt 0.6). Er sendet den Bestellschlüssel als Idempotenzschlüssel mit, damit die Karte nur einmal belastet wird, falls der Aufruf wiederholt wird.

6 **5. Datenbanktransaktion**
In einer ACID-Transaktion fügt der Service die Bestellung und ihre Artikel ein und speichert die Zahlungsreferenz. Entweder wird alles gespeichert oder nichts; nach dem Commit übersteht es einen Absturz (Abschnitt 0.5). Ein Index auf der Kunden-ID sorgt dafür, dass die Bestellhistorie schnell lädt.

7 **6. Queue**
Der Service legt eine `OrderPlaced`-Nachricht in eine Message Queue und antwortet sofort mit `201 Created`. Die App zeigt „Bestellung bestätigt“. Alles Weitere geschieht asynchron (Abschnitt 0.6).

8 **7. Andere Services reagieren**
Der Tablet-Service für das Restaurant, der Service für die Kurierdisposition und der Benachrichtigungsservice lesen die Nachricht jeweils, sobald sie bereit sind. Ist einer kurz ausgefallen, warten seine Nachrichten in der Queue.

9 **8. Push-Benachrichtigung**
Der Benachrichtigungsservice reagiert auf `OrderPlaced` und später auf eine ähnliche Nachricht, wenn ein Fahrer zugewiesen ist, indem er den Push-Dienst von Apple oder Google aufruft, eine weitere Drittanbieter-API, die die Benachrichtigung auf das Telefon des Kunden zustellt.

10 **9. Und hinter allem**
Jedes Stück Code auf diesem Weg wurde nach einer User Story geschrieben, in einem Pull Request geprüft, auf mehreren Ebenen getestet und über eine CI/CD-Pipeline via Staging in die Produktion ausgeliefert (dieser Abschnitt).

AUSGEARBEITETES BEISPIEL

Ausgearbeitetes Beispiel: was der Weg lehrt

Fragen Sie bei jedem Schritt „Was wäre, wenn?“, und die Architekturfragen zeigen sich.

Was, wenn beim Zahlungsaufruf ein Timeout eintritt?	Wurde die Karte belastet oder nicht? Der Idempotenzschlüssel erlaubt dem Bestellservice, gefahrlos erneut zu fragen, ohne doppelt zu belasten.
Was, wenn der Datenbank-Commit nach der Zahlung fehlschlägt?	Zahlung und Datenbank sind getrennte Systeme, daher deckt keine einzelne ACID-Transaktion beide ab. Der Entwurf braucht einen Weg zur Erstattung oder Wiederholung, ein Problem, das Kapitel 6 mit Sagas löst.
Was, wenn der Benachrichtigungsservice ausgefallen ist?	Sichtbar geht nichts kaputt: Bestellungen werden weiter angenommen, und Nachrichten warten in der Queue. Genau deshalb ist dieser Schritt asynchron.
Was, wenn sich der Verkehr zur Abendessenszeit verdreifacht?	Der Load Balancer verteilt die Anfragen auf mehr Kopien des Bestellservice, die Autoscaling hinzufügt; die Datenbank ist nun der wahrscheinliche Engpass.
Was, wenn ein Release den Checkout kaputt macht?	Tests in der Pipeline sollten es abfangen; falls nicht, löst das Monitoring einen Alert aus, und das Team macht einen Rollback auf die vorige Version.

Keine dieser Fragen betrifft, wie man eine Funktion schreibt. Es geht darum, wie die Teile angeordnet sind und was jeder Teil den anderen verspricht.

Warum das für die Architektur wichtig ist

Der Weg eines Tippens nutzt ein Dutzend Komponenten, drei Organisationen sowie synchrone und asynchrone Kommunikation, und er musste trotzdem entworfen werden: Jemand hat entschieden, dass die Zahlung synchron ist und Benachrichtigungen nicht, dass Bestellungen in einer relationalen Datenbank liegen, dass der Service in Containern über zwei Zonen läuft, dass der Zahlungsanbieter hinter einem Modul abgeschottet ist. Zu entscheiden, wie diese Teile angeordnet sind, welche Qualitäten jede Anordnung erreicht und was sie kostet, und festzuhalten, warum, ist Softwarearchitektur. Auch die Art, wie Software entsteht, ist für die Architektur wichtig: Eine Architektur, die sich nicht in Teilen testen oder nicht ohne die Abstimmung von zehn Teams ausliefern lässt, bremst jede Änderung. Kapitel 1 beginnt mit Was Softwarearchitektur ist.

PRÜFEN SIE IHR VERSTÄNDNIS

▼ Was ist der Unterschied zwischen Continuous Delivery und Continuous Deployment?

Bei Continuous Delivery kann jede erfolgreiche Änderung auf Knopfdruck freigegeben werden, aber ein Mensch entscheidet, wann. Bei Continuous Deployment wird jede erfolgreiche Änderung automatisch freigegeben.

▼ Warum stellt die Testpyramide Unit-Tests an die Basis?

Sie sind schnell, billig und zuverlässig, sodass ein Team bei jeder Änderung Tausende ausführen und in Minuten Feedback bekommen kann. Langsamere, anfälligere End-to-End-Tests bleiben den wenigen Abläufen vorbehalten, die sie brauchen.

▼ Warum ist beim Weg eines Tippens der Zahlungsaufruf synchron, die Benachrichtigung aber asynchron?

Die App darf erst „bestätigt“ anzeigen, wenn die Zahlung genehmigt ist, daher wartet sie auf diese Antwort. Die Benachrichtigung kann etwas später gesendet werden, daher läuft sie über eine Queue, und ihr Ausfall kann das Bestellen nicht blockieren.

▼ Welche zwei Hauptvorteile hat ein Pull Request über das Finden von Bugs hinaus?

Er verbreitet Wissen, weil mindestens eine weitere Person jede Änderung versteht, und er hält das Design des Codes konsistent, weil Reviewer strukturelle Entscheidungen hinterfragen können.

QUELLEN

- Martin Fowler: Continuous Integration: https://martinfowler.com/articles/continuousIntegration.html
- Martin Fowler: Continuous Delivery: https://martinfowler.com/bliki/ContinuousDelivery.html
- Ham Vocke: The Practical Test Pyramid (martinfowler.com): https://martinfowler.com/articles/practical-test-pyramid.html
- The 2020 Scrum Guide: https://scrumguides.org/scrum-guide.html
- Pro Git (Scott Chacon and Ben Straub), free online book: https://git-scm.com/book/en/v2

Weiterführende Literatur

- *Code: The Hidden Language of Computer Hardware and Software (2nd edition)*, Charles Petzold (2022), Microsoft Press. Baut einen Computer in verständlichen Schritten aus Schaltern und Codes auf; die beste nichttechnische Erklärung dafür, wie Hardware Software ausführt.
  https://www.microsoftpressstore.com/store/code-the-hidden-language-of-computer-hardware-and-software-9780137909100
- *Grokking Algorithms (2nd edition)*, Aditya Y. Bhargava (2024), Manning. Eine illustrierte, behutsame Einführung in Datenstrukturen, Algorithmen und die O-Notation, ideal nach dem Abschnitt über Daten und Algorithmen.
  https://www.manning.com/books/grokking-algorithms-second-edition
- *The Python Tutorial*, Python Software Foundation, docs.python.org. Die offizielle, kostenlose Einführung in die Sprache der Codebeispiele dieses Kapitels, einschließlich Funktionen, Modulen und Klassen.
  https://docs.python.org/3/tutorial/index.html
- *Learn web development*, MDN Web Docs contributors, Mozilla. Kostenlose, sorgfältig gepflegte Leitfäden für Einsteiger dazu, wie das Web funktioniert, zu HTTP, HTML und JavaScript.
  https://developer.mozilla.org/en-US/docs/Learn_web_development
- *Computer Networking: A Top-Down Approach (9th edition)*, James F. Kurose and Keith W. Ross (2025), Pearson. Das Standardlehrbuch über Netzwerke, das beim Web beginnt und sich bis zu den Paketen hinunterarbeitet; selektiv lesen, sobald die Grundlagen klar sind.
- *Pro Git (2nd edition)*, Scott Chacon and Ben Straub (2014), Apress (free online at git-scm.com). Das kostenlose, offizielle Buch über Git: Commits, Branches, Merges und Zusammenarbeit, wie im letzten Abschnitt eingeführt.
  https://git-scm.com/book/en/v2
- *The NIST Definition of Cloud Computing (SP 800-145)*, Peter Mell and Timothy Grance (2011), National Institute of Standards and Technology. Eine kurze, viel zitierte Definition von Cloud-Computing und seinen Service- und Bereitstellungsmodellen.
  https://csrc.nist.gov/pubs/sp/800/145/final
- *Designing Data-Intensive Applications (2nd edition)*, Martin Kleppmann and Chris Riccomini (2026), O'Reilly Media. Der beste nächste Schritt, sobald dieses Kapitel sitzt: wie sich Datenbanken, Netzwerke und verteilte Systeme wirklich verhalten und warum das wichtig ist.

Glossar

Abhängigkeit
Jede Software, die Ihr Code zum Funktionieren braucht, etwa eine Bibliothek, ein Framework oder ein anderer Service. Wenn sie sich ändert oder ausfällt, kann auch Ihr Code ausfallen.

Access Token
Ein kurzlebiger Berechtigungsnachweis, der nach der Anmeldung eines Benutzers oder einer Anwendung ausgestellt und mit jeder API-Anfrage mitgesendet wird, um zu belegen, was der Aufrufer tun darf.

ACID
Atomarität, Konsistenz, Isolation, Dauerhaftigkeit: die Garantien einer Datenbanktransaktion, damit die Daten trotz Abstürzen und gleichzeitiger Benutzer korrekt bleiben.

Agile Softwareentwicklung
Eine Familie von Ansätzen der Softwareentwicklung, beschrieben im Agilen Manifest von 2001, die in kurzen Zyklen funktionierende Software liefern und Pläne anhand von Feedback anpassen.

Algorithmus
Ein präzises, schrittweises Verfahren zur Lösung eines Problems, etwa zum Sortieren einer Liste oder zum Finden des kürzesten Wegs, das immer zu einem Ende kommt.

Anforderung
Eine Aussage darüber, was ein System tun muss oder welche Qualität es haben muss, abgestimmt mit den Menschen, die es brauchen.

API (Programmierschnittstelle)
Eine festgelegte Menge von Operationen, Eingaben und Ausgaben, über die eine Software eine andere nutzt, ohne wissen zu müssen, wie diese intern funktioniert.

API-Schlüssel
Eine lange geheime Zeichenkette, die die aufrufende Anwendung gegenüber einer API identifiziert. Wer den Schlüssel besitzt, kann als diese Anwendung auftreten, daher muss er geheim gehalten werden.

API-Versionierung
Die Veröffentlichung von Änderungen an einer API unter einer neuen Versionskennung wie /v2/, während die alte Version eine Zeit lang weiter funktioniert, damit Clients in ihrem eigenen Tempo umsteigen können.

API-Vertrag
Das dokumentierte Versprechen einer API: welche Anfragen sie annimmt, was jede zurückgibt, welche Fehler auftreten können und was sich nicht ohne Vorankündigung ändert.

Asynchrones Messaging
Kommunikation, bei der der Sender eine Nachricht irgendwo ablegt, damit der Empfänger sie später verarbeitet, und weiterarbeitet, ohne auf eine Antwort zu warten.

Availability Zone
Eine von mehreren physisch getrennten Gruppen von Rechenzentren innerhalb einer Cloud-Region, jede mit eigener Stromversorgung, Kühlung und Netzwerkanbindung, nahe genug beieinander für schnelle Verbindungen.

Backup
Eine getrennt vom Original aufbewahrte Kopie von Daten, damit sie nach Verlust, Beschädigung oder Löschung wiederhergestellt werden können.

Bandbreite
Wie viele Daten eine Verbindung pro Sekunde übertragen kann, etwa 100 Megabit pro Sekunde. Hohe Bandbreite verringert die Latenz nicht.

Betriebssystem
Die Software, die die Hardware eines Computers verwaltet und zwischen Programmen aufteilt: CPU-Zeit, Arbeitsspeicher, Dateien, Netzwerk und Sicherheit. Beispiele sind Linux, Windows, macOS, Android und iOS.

Bibliothek
Wiederverwendbarer Code, den jemand anderes geschrieben hat und den Ihr Code bei Bedarf aufruft, etwa eine Bibliothek für Datumsangaben oder für den E-Mail-Versand.

Bit
Eine Binärziffer, 0 oder 1: die kleinste Informationseinheit, die ein Computer speichert.

Branch
Ein separater Entwicklungszweig in der Versionskontrolle, in dem an einer Änderung gearbeitet wird, ohne die Hauptlinie zu beeinflussen, bis sie zum Mergen bereit ist.

Breaking Change
Eine Änderung an einer API, die bestehenden, korrekten Client-Code nicht mehr funktionieren lässt, etwa das Umbenennen oder Entfernen eines Felds.

Bug
Ein Fehler in Software, durch den sie sich anders verhält als beabsichtigt, vom Absturz bis zu einer stillschweigend falschen Antwort.

Byte
Eine Gruppe von acht Bits, die 256 verschiedene Werte aufnehmen kann. Datei- und Speichergrößen werden in Bytes gezählt (Kilobyte, Megabyte, Gigabyte).

Bytecode
Kompakte Anweisungen für eine virtuelle Maschine statt für eine echte CPU. Java und Python kompilieren Quellcode zu Bytecode, den ihre Laufzeitumgebungen dann auf jeder unterstützten Maschine ausführen.

Cache
Eine Kopie von Daten an einem schneller erreichbaren Ort, damit wiederholte Anfragen langsame Arbeit vermeiden. Sie tauscht zusätzlichen Speicher und das Risiko veralteter Daten gegen Geschwindigkeit.

CI/CD-Pipeline
Die automatisierte Abfolge von Schritten (Build, Test, Deployment auf Staging, Deployment in Produktion), die jede Änderung auf dem Weg zu den Benutzern durchläuft.

Client-Server-Modell
Eine Anordnung, bei der Clients (etwa Browser oder mobile Apps) Anfragen senden und Server, die gemeinsame Daten und Logik halten, antworten.

Cloud-Computing
Das Mieten von Rechenressourcen (Server, Speicher, Datenbanken und andere Dienste) über das Internet nach Bedarf, wobei man für die tatsächliche Nutzung bezahlt.

Code-Review
Die Prüfung einer vorgeschlagenen Codeänderung durch Kollegen vor dem Mergen, um Mängel zu finden, Wissen zu teilen und das Design einheitlich zu halten.

Commit
Ein gespeicherter Schnappschuss von Änderungen in der Versionskontrolle, mit Autor, Zeitstempel und einer Nachricht, die die Änderung erklärt.

Compiler
Ein Programm, das Quellcode vor der Ausführung in eine andere Form übersetzt, etwa in Maschinencode oder Bytecode.

Container
Eine paketierte Anwendung mit ihren Bibliotheken und Einstellungen, ausgeführt als isolierter Prozess, der sich den Betriebssystemkern des Hosts teilt. Leichtgewichtiger und schneller startend als eine virtuelle Maschine.

Container-Image
Ein schreibgeschütztes, versioniertes Paket mit einer Anwendung und allem, was sie zum Laufen braucht, aus dem sich überall identische Container starten lassen.

Content Delivery Network (CDN)
Ein Netz von Servern an vielen Standorten, das Kopien von Inhalten wie Bildern, Skripten und Videos nahe bei den Benutzern vorhält, damit sie schneller laden und der Ursprungsserver weniger Arbeit hat.

Continuous Delivery
Software stets in einem releasefähigen Zustand halten, sodass jede Version, die die automatisierte Pipeline besteht, auf Knopfdruck in Produktion gebracht werden kann.

Continuous Deployment
Das automatische Ausliefern jeder Änderung, die die Pipeline besteht, in die Produktion, ohne manuellen Freigabeschritt.

Continuous Integration (CI)
Die Praxis, die Änderungen aller Entwicklerinnen und Entwickler häufig, mindestens täglich, in die gemeinsame Hauptlinie zu integrieren, wobei ein automatisierter Build und Tests jede Änderung prüfen.

CPU (Prozessor)
Die zentrale Verarbeitungseinheit: der Chip, der die Anweisungen eines Programms ausführt, einen einfachen Schritt nach dem anderen, milliardenfach pro Sekunde. Moderne CPUs haben mehrere Kerne, die parallel arbeiten.

Datei
Eine benannte Folge von Bytes, die auf einem Datenträger gespeichert und vom Betriebssystem verwaltet wird, etwa ein Foto, eine Tabelle oder ein Log.

Datenbank
Eine organisierte Sammlung von Daten, die so gespeichert ist, dass sie durchsucht, aktualisiert und konsistent gehalten werden kann, meist verwaltet von einem Datenbankmanagementsystem.

Datenbankindex
Eine zusätzliche Datenstruktur, meist ein B-Baum, mit der die Datenbank Zeilen anhand des Werts einer Spalte findet, ohne die ganze Tabelle zu lesen. Sie beschleunigt Lesezugriffe, verlangsamt aber Schreibzugriffe und belegt Platz.

Datenbankmanagementsystem (DBMS)
Die Software, die eine Datenbank speichert und den Zugriff darauf steuert: Sie beantwortet Abfragen, setzt Regeln durch, bedient viele Benutzer gleichzeitig und stellt den Zustand nach Abstürzen wieder her. PostgreSQL und MySQL sind Beispiele.

Datenstruktur
Eine Art, Daten im Speicher zu organisieren, etwa als Liste, Dictionary, Menge, Queue, Stack, Baum oder Graph, gewählt, damit die benötigten Operationen schnell sind.

Datentyp
Die Art eines Werts, etwa eine ganze Zahl, eine Dezimalzahl, Text oder wahr/falsch, die bestimmt, welche Operationen darauf sinnvoll sind.

DevOps
Eine Kultur und eine Reihe von Praktiken, bei denen Entwicklung und Betrieb als Einheit arbeiten und die Verantwortung für Bau, Release und Betrieb von Software teilen, unterstützt durch Automatisierung und Messung.

Dictionary (Map)
Eine Datenstruktur, die Werte unter eindeutigen Schlüsseln speichert, sodass ein Wert direkt über seinen Schlüssel gefunden werden kann, meist in annähernd konstanter Zeit. Auch Map oder Hashtabelle genannt.

DNS
Domain Name System: der Verzeichnisdienst des Internets, der Namen wie www.example.com in IP-Adressen umwandelt.

Docker
Ein weit verbreiteter, 2013 veröffentlichter Werkzeugsatz zum Bauen von Container-Images und Ausführen von Containern. Sein Image-Format wurde zu einem offenen Standard.

Dokumentendatenbank
Eine NoSQL-Datenbank, die jeden Datensatz als in sich geschlossenes Dokument speichert, meist JSON-ähnlich, das verschachtelte Daten enthalten kann und kein festes gemeinsames Schema braucht.

End-to-End-Test
Ein automatisierter Test, der das ganze System so bedient wie ein Benutzer, zum Beispiel durch Tippen durch eine App, um einen vollständigen Nutzungsablauf zu prüfen.

Framework
Wiederverwendbarer Code, der die Gesamtstruktur einer Anwendung vorgibt und Ihren Code zu den richtigen Zeitpunkten aufruft, etwa Django oder Spring.

Fremdschlüssel
Eine Spalte in einer Tabelle, die den Primärschlüssel einer Zeile in einer anderen Tabelle enthält und die beiden verknüpft. Die Datenbank kann Werte zurückweisen, die ins Leere zeigen.

Funktion
Ein benanntes, wiederverwendbares Stück Code, das Eingaben (Parameter) entgegennimmt, eine Arbeit verrichtet und meist eine Ausgabe zurückgibt.

Git
Das am weitesten verbreitete Versionskontrollsystem, 2005 von Linus Torvalds entwickelt. Jede Entwicklerin und jeder Entwickler hat eine vollständige Kopie der Projekthistorie.

Graphdatenbank
Eine Datenbank, die Elemente als Knoten und die Beziehungen zwischen ihnen als Kanten speichert, sodass das Verfolgen von Verbindungsketten schnell ist.

Hardware
Die physischen Teile eines Computers, die man anfassen könnte: Prozessor, Arbeitsspeicher, Datenspeicher, Netzwerkkarte, Bildschirm und so weiter.

Horizontale Skalierung (Scale-out)
Mehr Last bewältigen, indem weitere Maschinen hinzugefügt und die Arbeit auf sie verteilt wird, meist hinter einem Load Balancer.

HTTP
Hypertext Transfer Protocol: die Anfrage-Antwort-Sprache des Web, in der ein Client eine Methode wie GET an eine URL sendet und ein Server mit einem Statuscode und Inhalt antwortet.

HTTP-Statuscode
Eine dreistellige Zahl in jeder HTTP-Antwort, die angibt, wie die Anfrage verlaufen ist: 2xx Erfolg, 3xx Umleitung, 4xx ein Problem mit der Anfrage, 5xx ein Problem auf dem Server.

Hypervisor
Software, die virtuelle Maschinen erzeugt und ausführt und dabei Prozessoren, Arbeitsspeicher und Geräte eines physischen Computers unter ihnen aufteilt, während sie sie voneinander isoliert hält.

IaaS (Infrastructure as a Service)
Ein Cloud-Modell, bei dem Sie virtuelle Maschinen, Speicher und Netzwerke mieten und das Betriebssystem und alles darüber selbst verwalten.

Index
Eine zusätzliche Datenstruktur neben den Daten, mit der ein System Elemente anhand eines gewählten Felds findet, ohne alles zu durchsuchen, wie das Register hinten in einem Buch.

Integrationstest
Ein automatisierter Test, der prüft, ob mehrere Teile korrekt zusammenarbeiten, etwa Code mit einer echten Datenbank oder der API eines anderen Service.

Interpreter
Ein Programm, das Code liest und seine Anweisungen fortlaufend ausführt, statt das ganze Programm vorab zu übersetzen.

Inversion of Control
Die Anordnung, bei der ein Framework und nicht Ihr eigener Code den Programmablauf steuert und Ihren Code bei Bedarf aufruft.

IP-Adresse
Die numerische Adresse, die ein Gerät in einem Netzwerk identifiziert, damit Pakete an es zugestellt werden können, etwa 192.0.2.10 (IPv4) oder 2001:db8::1 (IPv6).

Join
Eine SQL-Operation, die Zeilen aus zwei oder mehr Tabellen über zusammengehörige Spalten verknüpft, etwa Bestellungen mit den Kunden, die sie aufgegeben haben.

JSON (JavaScript Object Notation)
Ein Textformat für strukturierte Daten aus Objekten (Name-Wert-Paare in geschweiften Klammern), Arrays (Listen in eckigen Klammern), Strings, Zahlen, true, false und null. Standardisiert in RFC 8259.

Kanban
Eine Methode, Arbeit als kontinuierlichen Fluss auf einem visuellen Board zu steuern und dabei zu begrenzen, wie viele Aufgaben gleichzeitig in Bearbeitung sein dürfen.

Kapselung
Die internen Daten und Details eines Objekts oder Moduls privat halten, sodass anderer Code es nur über die Operationen nutzen kann, die es anbieten will.

Key-Value-Store
Eine Datenbank, die Werte unter eindeutigen Schlüsseln speichert und nur über den Schlüssel abruft, wie ein sehr großes, sehr schnelles Dictionary.

Klasse und Objekt
Eine Klasse ist eine Vorlage, die Daten mit den Funktionen (Methoden) bündelt, die auf ihnen arbeiten; ein Objekt ist eine aus dieser Vorlage erzeugte Instanz, etwa das Konto eines bestimmten Kunden.

Kohäsion
Wie eng die Inhalte eines Teils eines Systems zusammengehören. Hohe Kohäsion bedeutet, dass ein Modul eine klare Aufgabe erfüllt.

Kopplung
Wie stark ein Teil eines Systems von den Details eines anderen abhängt. Lose Kopplung bedeutet, dass eine Änderung an einem Teil selten Änderungen an anderer Stelle erzwingt.

Latenz
Die Verzögerung, bis Daten ankommen oder eine Antwort zurückkommt, meist in Millisekunden gemessen. Entfernung, Netzwerk-Hops und langsame Verarbeitung tragen alle dazu bei.

Linux
Ein freier Open-Source-Betriebssystemkern, der zusammen mit weiterer Software als Betriebssystem auf den meisten Servern, Cloud-Maschinen und Containern sowie auf Android-Smartphones läuft.

Load Balancer
Eine Komponente, die eingehende Anfragen entgegennimmt und auf mehrere Server verteilt, wobei sie fehlerhafte Server überspringt.

Managed Database
Ein von einem Cloud-Anbieter betriebener Datenbankdienst, der Installation, Patches, Backups und Failover übernimmt, während der Kunde Schema, Abfragen und Zugriffe verwaltet.

Managed Service
Ein Cloud-Dienst, etwa eine Datenbank, eine Queue oder ein Speicherdienst, den der Anbieter installiert, betreibt, patcht und skaliert, sodass Kunden ihn nutzen, ohne ihn selbst zu betreiben.

Maschinencode
Als Zahlen codierte Anweisungen, die eine bestimmte CPU-Familie direkt ausführen kann. Maschinencode für eine Familie (zum Beispiel x86-64) läuft nicht auf einer anderen (zum Beispiel Arm).

Message Queue
Software, die Nachrichten von Sendern speichert, bis die Empfänger bereit sind, sie zu verarbeiten, sodass beide nicht gleichzeitig laufen oder gleich schnell sein müssen.

Modul
Eine Codeeinheit, oft eine einzelne Datei, die zusammengehörige Funktionen und Daten unter einem Namen bündelt und festlegt, was sie anderem Code zur Verfügung stellt. Ein Paket ist eine Gruppe von Modulen.

NoSQL
Ein Sammelbegriff für Datenbanken, die nicht das relationale Tabellenmodell verwenden, etwa Dokumenten-, Key-Value-, Wide-Column- und Graphdatenbanken.

O-Notation (Big-O)
Eine Art zu beschreiben, wie der Aufwand eines Algorithmus mit der Datenmenge wächst, zum Beispiel O(n) für lineares Wachstum oder O(1) für konstante Zeit.

PaaS (Platform as a Service)
Ein Cloud-Modell, bei dem Sie Ihren Anwendungscode oder Container liefern und der Anbieter die darunterliegende Plattform betreibt, patcht und skaliert.

Paket (Netzwerk)
Ein kleines Datenstück, das mit angehängten Adressinformationen über ein Netzwerk gesendet wird. Große Nachrichten werden in viele Pakete aufgeteilt und am anderen Ende wieder zusammengesetzt.

Paketmanager
Ein Werkzeug wie pip (Python) oder npm (JavaScript), das Bibliotheken und die Bibliotheken, von denen sie abhängen, in den Versionen herunterlädt, die ein Projekt vorgibt.

Pay-as-you-go
Ein Preismodell, bei dem Ihnen Ressourcen nur berechnet werden, solange Sie sie nutzen, zum Beispiel pro Sekunde Serverzeit oder pro gespeichertem Gigabyte.

Port
Eine Zahl von 0 bis 65535, die angibt, welches Programm auf einem Gerät Netzwerkverkehr empfangen soll, etwa 443 für HTTPS.

Primärschlüssel
Eine Spalte oder Gruppe von Spalten, deren Wert jede Zeile einer Tabelle eindeutig identifiziert, etwa customer_id.

Produktion
Die Live-Umgebung, auf die echte Nutzer und echte Daten angewiesen sind.

Programm
Eine Liste präziser Anweisungen, die ein Computer befolgt, um eine Aufgabe zu erledigen. Ein laufendes Programm heißt Prozess.

Prozess
Eine laufende Instanz eines Programms mit eigenem privatem Speicher und eigenem Anteil an CPU-Zeit, vom Betriebssystem erzeugt und verwaltet.

Pull Request
Eine Anfrage, einen Branch in die Hauptlinie zu mergen, auf einer Plattform wie GitHub, wo Kollegen die Änderung reviewen und automatische Prüfungen laufen, bevor sie angenommen wird.

Quellcode
Der für Menschen lesbare Text eines Programms, geschrieben in einer Programmiersprache wie Python oder Java, der anschließend kompiliert oder interpretiert wird, damit ein Computer ihn ausführen kann.

RAM (Arbeitsspeicher)
Random-Access Memory: schneller, flüchtiger Speicher, der die gerade verwendeten Programme und Daten enthält. Sein Inhalt geht verloren, wenn der Strom ausfällt oder das Programm endet.

Rate Limit
Eine Obergrenze dafür, wie viele Anfragen ein Client in einem Zeitraum stellen darf, etwa 100 pro Minute. Anfragen darüber werden abgewiesen, typischerweise mit dem HTTP-Status 429.

Rechenzentrum
Ein Gebäude, das dafür ausgelegt ist, viele Server mit zuverlässiger Stromversorgung, Kühlung, physischer Sicherheit und Netzwerkanbindung zu beherbergen.

Region
Ein geografisches Gebiet, etwa Westeuropa oder Ost-USA, in dem ein Cloud-Anbieter eine eigene Gruppe von Rechenzentren betreibt. Regionen sind voneinander isoliert.

Relationale Datenbank
Eine Datenbank, die Daten in Tabellen aus Zeilen und Spalten speichert und Tabellen über Schlüssel verknüpft. Sie wird mit SQL abgefragt.

REST (Representational State Transfer)
Ein Architekturstil für Web-APIs, im Jahr 2000 von Roy Fielding beschrieben, bei dem Clients mit Standard-HTTP-Methoden auf Ressourcen einwirken, die durch URLs identifiziert werden.

SaaS (Software as a Service)
Komplette Software, die über das Internet genutzt wird, meist über einen Browser oder eine App, wobei der Anbieter alles betreibt.

Schema
Die deklarierte Struktur einer Datenbank: ihre Tabellen, Spalten, Datentypen, Schlüssel und Regeln.

Schnittstelle
Die vereinbarte Menge von Operationen, die eine Software anderen anbietet, mit ihren Ein- und Ausgaben: ein Vertrag, der verbirgt, wie die Arbeit erledigt wird.

Scrum
Ein agiles Framework, in dem ein kleines Team in festen Zyklen von höchstens einem Monat, den Sprints, arbeitet, mit festgelegten Events, Verantwortlichkeiten und Artefakten.

SDK (Software Development Kit)
Eine Bibliothek, meist vom Eigentümer einer API bereitgestellt, die die HTTP-Aufrufe der API für eine bestimmte Programmiersprache in gewöhnliche Funktionen verpackt.

Server
Ein Computer oder ein darauf laufendes Programm, der auf Anfragen anderer Computer (Clients) wartet und sie beantwortet.

Serverless und FaaS (Function as a Service)
Ein Cloud-Modell, bei dem Sie einzelne Funktionen hochladen, die nur laufen, wenn ein Ereignis sie auslöst; der Anbieter kümmert sich um alle Server, und Sie bezahlen pro Ausführung.

Software
Programme zusammen mit den Daten und der Konfiguration, die sie benötigen; die Anweisungen, die der Hardware sagen, was sie tun soll.

SQL (Structured Query Language)
Die Standardsprache zum Definieren, Abfragen und Ändern von Daten in relationalen Datenbanken. Sie beschreiben das gewünschte Ergebnis, und die Datenbank ermittelt, wie sie es erhält.

Synchroner Aufruf
Eine Anfrage, bei der der Aufrufer auf die Antwort wartet, bevor er fortfährt, wie bei einem gewöhnlichen HTTP-API-Aufruf.

TCP
Transmission Control Protocol: liefert einen zuverlässigen, geordneten Datenstrom zwischen zwei Programmen und sendet verlorene Pakete erneut. Der meiste Webverkehr lief traditionell darüber.

Testpyramide
Eine Orientierungshilfe von Mike Cohn für das Verhältnis automatisierter Tests: viele schnelle Unit-Tests, weniger Integrationstests und nur wenige langsame End-to-End-Tests.

Thread
Eine Folge von Anweisungen, die innerhalb eines Prozesses ausgeführt wird. Threads desselben Prozesses teilen sich dessen Speicher, sodass sie gleichzeitig arbeiten können, sich aber abstimmen müssen.

TLS und HTTPS
Transport Layer Security verschlüsselt Daten zwischen zwei Programmen und weist über Zertifikate die Identität des Servers nach. HTTPS ist HTTP über TLS.

Transaktion
Eine Gruppe von Datenbankoperationen, die als eine Einheit behandelt wird: Entweder werden alle wirksam (Commit) oder keine (Rollback).

UDP
User Datagram Protocol: sendet einzelne Pakete ohne Garantie für Zustellung oder Reihenfolge. Schneller und einfacher als TCP; verwendet für Videoanrufe, Spiele und DNS.

Umgebung
Eine separate, vollständige Kopie des Systems und seiner Einstellungen für einen bestimmten Zweck, etwa Entwicklung, Test, Staging oder Produktion.

Unit-Test
Ein automatisierter Test, der ein kleines Stück Code, etwa eine Funktion, isoliert prüft und in Millisekunden läuft.

User Story
Eine kurze Beschreibung eines Features aus Sicht der Person, die es sich wünscht, oft formuliert als: Als [Rolle] möchte ich [Fähigkeit], damit [Nutzen].

UTF-8
Die vorherrschende Art, Text als Bytes zu codieren. Sie kann jedes Unicode-Zeichen darstellen und verwendet ein Byte für einfache lateinische Buchstaben und bis zu vier Bytes für andere.

Versionskontrolle
Ein System, das jede Änderung an einer Menge von Dateien aufzeichnet, wer sie vorgenommen hat und warum, sodass Menschen parallel arbeiten, Änderungen prüfen und zu jeder früheren Version zurückkehren können.

Vertikale Skalierung (Scale-up)
Mehr Last bewältigen, indem auf eine größere Maschine mit mehr Prozessoren oder Arbeitsspeicher gewechselt wird.

Virtuelle Maschine (VM)
Eine softwarebasierte Nachbildung eines vollständigen Computers mit eigenem Betriebssystem, die neben anderen VMs auf einer physischen Maschine unter einem Hypervisor läuft.

Web-API
Eine über ein Netzwerk angebotene API, meist über HTTP, damit Programme auf verschiedenen Maschinen, geschrieben in verschiedenen Sprachen, zusammenarbeiten können.

Webhook
Eine HTTP-Anfrage, die ein Anbieter an eine von seinem Kunden registrierte URL sendet, wenn ein Ereignis eintritt, etwa eine erfolgreiche Zahlung. Sie kehrt die übliche Richtung eines API-Aufrufs um.

Wide-Column-Store
Eine NoSQL-Datenbank, die Daten in über einen Schlüssel identifizierte Zeilen mit jeweils vielen Spalten gruppiert, verteilt auf viele Maschinen. Ausgelegt für sehr große Schreibvolumen.