Engineer

Microservices

Architektur aus kleinen, einzeln bereitstellbaren Diensten

Was es ist

Microservices bezeichnen einen Architekturstil, der eine Anwendung in kleine Dienste aufteilt. Jeder Dienst ist ein eigenes Programm, läuft als separater Prozess und kann ohne die anderen bereitgestellt werden. James Lewis und Martin Fowler, deren Artikel von 2014 dem Stil seine bekannteste Beschreibung gab, fassen ihn als Bau von Anwendungen aus „Gruppen unabhängig bereitstellbarer Dienste“ zusammen. Sie nennen, was solche Systeme gewöhnlich gemeinsam haben: Dienste, die über leichtgewichtige Mechanismen sprechen, oft eine HTTP-Ressourcen-API; Dienste, die um Geschäftsfähigkeiten herum aufgebaut sind; vollständig automatisierte Bereitstellung; ein Minimum an zentraler Verwaltung; und die Freiheit jedes Diensts, seine eigene Programmiersprache und seinen eigenen Datenspeicher zu verwenden. Sie ergänzen ausdrücklich, dass es keine formale Definition gibt, sondern nur Merkmale, die die meisten so bezeichneten Systeme aufweisen.

Das Wort ist jünger als die Idee. Eine Fußnote desselben Artikels hält fest, dass ein Workshop von Softwarearchitekten bei Venedig den Begriff im Mai 2011 diskutierte und dieselbe Gruppe sich im Mai 2012 auf „Microservices“ einigte. Adrian Cockcroft bei Netflix hatte den Ansatz als feinkörnige Form von SOA bezeichnet. Lewis und Fowler behaupten nicht, dass der Stil neu sei: Sie verfolgen seine Wurzeln mindestens bis zu den Prinzipien zurück, auf denen Unix aufgebaut wurde.

Die Alternative, mit der meist verglichen wird, ist der Monolith: eine serverseitige Anwendung, die als eine Einheit gebaut und bereitgestellt wird. Die gesamte Logik einer Anfrage läuft in einem Prozess. Mit etwas Sorgfalt lässt sich die ganze Anwendung auf dem Laptop eines Entwicklers ausführen und testen. Sie skaliert, indem weitere Kopien hinter einem Load Balancer laufen. Die Schwierigkeiten zeigen sich später. Eine kleine Änderung bedeutet, alles neu zu bauen und

bereitzustellen. Eine gute modulare Struktur ist über Jahre schwer zu erhalten, und der Teil mit höherem Kapazitätsbedarf lässt sich nur zusammen mit allem anderen skalieren.

Lewis und Fowler definieren eine Komponente als Softwareeinheit, die sich unabhängig ersetzen und aktualisieren lässt. Eine Bibliothek wird in ein Programm eingebunden und im Speicher aufgerufen. Ein Dienst ist eine Komponente außerhalb des Prozesses, erreichbar über das Netzwerk durch eine Webanfrage oder einen entfernten Prozeduraufruf. Dieser Unterschied ist für die Kopplung wichtig. In den meisten Sprachen halten nur Dokumentation und Disziplin anderen Code davon ab, in das Innere einer Bibliothek zu greifen. Ein Dienst lässt sich dagegen nur über ausdrückliche entfernte Aufrufe erreichen. Seine Schnittstelle ist deshalb viel schwerer zu umgehen. Ein Dienst kann trotzdem aus mehreren Prozessen bestehen, etwa einer Anwendung samt einer nur von ihr verwendeten Datenbank.

Das „micro“ sagt weniger, als es scheint. Lewis und Fowler weisen darauf hin, dass der Name die Aufmerksamkeit auf die Größe lenkt und Streit darüber erzeugt, was als micro zählt. Die größten ihnen berichteten Dienste richteten sich nach Amazons Regel für ein Zwei-Pizza-Team, also höchstens einem Dutzend Menschen. Am kleinen Ende sahen sie ein Team von einem halben Dutzend Menschen, das ein halbes Dutzend Dienste betreute.

Das Konzept für eine Entscheidung nutzen

Nutzen Sie „Was sind Microservices?“ für eine konkrete Entscheidung. Wir können helfen, das Konzept mit dem System in Verbindung zu bringen, das Sie entwickeln oder einkaufen. Der Teil, den Nutzer nie sehen: wo die Daten liegen, welche Regeln gelten und wie andere Systeme darauf zugreifen dürfen.

Unsere Leistungen entdecken Kontakt aufnehmen →

Die Annahmen hinter dem Begriff prüfen

Nutzen Sie die Erklärung und die verlinkten Arbeiten, um Fragen für Ihr Projekt zu formulieren. Wir können die Annahmen gemeinsam mit Ihnen prüfen, bevor ein Werkzeug ausgewählt wird. 6 ETL/ELT-Pipelines entworfen und betrieben — 6 ETL-Pipelines über drei Datenbankengines hinweg

6 ETL/ELT-Pipelines entworfen und betrieben Kontakt aufnehmen →

Warum es wichtig ist

Der erste Gewinn ist die Veröffentlichung nach eigenem Zeitplan. Ist eine Anwendung in Dienste geteilt, muss bei vielen Änderungen nur ein Dienst neu bereitgestellt werden. Ein Team kann deshalb ausliefern, ohne auf die anderen zu warten. Lewis und Fowler stellen klar, dass das nicht uneingeschränkt gilt: Eine Änderung an der Schnittstelle eines Diensts muss weiterhin mit den aufrufenden Diensten abgestimmt werden.

Der zweite Gewinn ist, dass die Software der Organisation folgen kann, statt gegen sie zu arbeiten. Melvin Conway argumentierte 1968 in Datamation, dass Organisationen, die Systeme entwerfen, auf Entwürfe beschränkt sind, die ihre eigenen Kommunikationsstrukturen kopieren. Lewis und Fowler wenden das auf die übliche Aufteilung einer großen Anwendung nach technischen Schichten an: ein Team für die Oberfläche, eines für serverseitige Logik und eines für die Datenbank. Schon eine einfache Änderung wird dann zu einem Projekt über mehrere Teams hinweg, und Logik landet dort, wo ein Team gerade Zugriff hat. Der Microservice-Stil dreht das um. Jeder Dienst deckt eine Geschäftsfähigkeit, etwa Bestellungen, Bestand oder Zahlungen, von der Oberfläche bis zum Speicher ab. Er gehört einem funktionsübergreifenden Team mit den Fähigkeiten für den gesamten Ausschnitt. Dieses Team sollte den Dienst sein ganzes Leben lang verantworten, statt ihn am Projektende abzugeben. Amazons Fassung der Regel lautet: Wer einen Dienst baut, betreibt ihn auch in Produktion.

Der dritte Gewinn ist, dass die Logik in den Diensten bleibt. Lewis und Fowler fassen die Vorliebe als intelligente Endpunkte und einfache Leitungen zusammen. Dienste sprechen über HTTP-Anfragen an Ressourcen-APIs oder einen leichtgewichtigen Nachrichtenbus, der nur Nachrichten weiterleitet. Die Entscheidungen fallen in den Diensten an beiden Enden. Das ist eine bewusste Abkehr vom Enterprise Service Bus, der mitten im Netzwerk statt an dessen Enden weiterleitet, transformiert und Geschäftsregeln anwendet.

Daten werden nach denselben Grenzen aufgeteilt. Jeder Dienst verwaltet seine eigene Datenbank. Wenn es ihrer Arbeit dient, können verschiedene Dienste unterschiedliche Datenbanksysteme verwenden. Eric Evans' domänengetriebener Entwurf liefert die Begründung: Ein großes System enthält mehrere Modelle. Jedes braucht einen ausdrücklich abgegrenzten Kontext, bestimmt durch Teamorganisation, die von ihm geregelten Anwendungsteile und physische Dinge wie Codebasen und Datenbankschemas. Vertrieb und Support können einen Kunden dann unterschiedlich modellieren, ohne beides in einer Tabelle vereinbaren zu müssen.

Die Kosten gehören zur Definition, und Lewis und Fowler nennen sie offen. Ein entfernter Aufruf kostet viel mehr als einer im Speicher, daher müssen Schnittstellen zwischen Diensten gröber geschnitten sein. Code über eine Dienstgrenze zu verschieben ist schwierig: Schnittstellenänderungen müssen abgestimmt, Schichten von Rückwärtskompatibilität ergänzt werden, und das Testen wird komplizierter. Verteilte Transaktionen sind schwer zu implementieren. Dienste stimmen sich deshalb ohne sie ab, akzeptieren möglicherweise erst später erreichte Konsistenz und reparieren Fehler mit

ausgleichenden Operationen. Jeder Aufruf kann scheitern. Passen die Dienste nicht sauber zusammen, ist die Komplexität nicht verschwunden. Sie liegt nun in den schwerer sichtbaren Verbindungen zwischen ihnen. Lewis und Fowler nennen ein Argument, das sie vernünftig finden: zuerst einen Monolithen mit sauberen Modulen bauen und ihn erst aufteilen, wenn er Probleme verursacht. Sie ergänzen eine Einschränkung: Eine innerhalb eines Prozesses gut funktionierende Schnittstelle ist selten eine gute Schnittstelle zwischen Diensten.

Wie es in der Praxis aussieht

Die Notiz Die letzte Eintrittskarte genau einmal verkaufen greift erst zu einem Dienst, wenn eine Datenbank nicht mehr reicht. Dann nennt sie die Regel, die den Entwurf zusammenhält: Jeder Artikel hat genau einen Eigentümer seines Bestands, einen Bestandsdienst. Nichts anderes entscheidet. Dieser Dienst wird nach Mandant und Artikel in Shards aufgeteilt; die größten Mandanten erhalten eigene Zellen. Nur die Bestandsreservierung muss synchron sein. Zahlung, Bestellung und Beleg sind separate Systeme, und keine Transaktion spannt sich über sie. Die Notiz verwirft Zwei-Phasen-Commit, weil es blockiert, solange irgendein Teilnehmer langsam ist, und Zahlungsanbieter daran nicht teilnehmen. Stattdessen verwendet sie eine Saga: eine Folge lokaler Schritte mit jeweils einem zugehörigen Schritt, der den betreffenden Vorgang rückgängig macht.

Jeder Schritt schreibt seine Nachricht in derselben Transaktion wie die angekündigte Änderung in eine Outbox-Tabelle. In PostgreSQL kann eine datenändernde Anweisung innerhalb von `WITH` ihr Ergebnis an ein Insert weitergeben. Der Bestandsdienst kann deshalb mit einer einzigen Anweisung eine Einheit reservieren und die Nachricht aufzeichnen:

```
-- inventory service, its own database
WITH held AS (
    UPDATE inventory
    SET available = available - 1
    WHERE tenant_id = 't-7' AND item_id = 'item-881'
      AND available >= 1
    RETURNING item_id
)
INSERT INTO outbox (order_id, item_id, event, units)
SELECT 'A-17', item_id, 'StockReserved', 1 FROM held;
-- sold out: no row matches, so no message is written
```

Change Data Capture veröffentlicht danach die Outbox-Zeilen. Jeder Verbraucher ist idempotent, mit der Bestellung als Schlüssel, weil Nachrichten mindestens einmal ankommen. In dieser Skizze greift kein anderer Dienst auf die Tabelle `inventory` zu: Die anderen erfahren den Bestand aus diesen Nachrichten oder durch eine Anfrage an den Bestandsdienst.

Dieselbe Notiz zeigt auch den Platz der Aufteilung in der Arbeitsreihenfolge. In ihrer Liste der ersten Schritte kommt ein eigener Dienst mit eigenem Verantwortlichen für den Bestand an vierter Stelle: nach dem Messen, nach dem Umbau des Schreibens in eine einzige bedingte Anweisung und nach der Lastabweisung durch Warteraum, Zugangstor und zwischengespeicherte Ausverkauft-Antwort. Die günstigeren Korrekturen kommen zuerst.

Der Portfoliofall Eine maritime Plattform in klaren Schichten hält den Architekturbeitrag der Gründerin bei NextMariner fest und zeigt die andere Seite der Entscheidung. Es war ein junges Produkt mit noch stark veränderlichen Anforderungen. Eine geänderte Geschäftsregel sollte nicht bedeuten, zugleich Frontend, API und Datenbank anzufassen. Gewählt wurden drei Schichten mit je einer Aufgabe: ein Next.js-Frontend für die Darstellung, eine bewusst dünne Go-API für Weiterleitung, Validierung, Sicherheitsfilterung und Orchestrierung sowie sämtliche Geschäftslogik in PostgreSQL-Funktionen. Die Grenzen wurden aufgeschrieben und in Reviews durchgesetzt. Lewis und Fowler merken an, dass modulare Grenzen innerhalb einer Anwendung viel Disziplin brauchen und die ausdrücklichere Trennung von Diensten Teamgrenzen leichter klar hält. Diese Plattform hielt die Disziplin ein, ohne ihre Geschäftslogik in separate Dienste aufzuteilen.

Wo es schiefgeht

Ketten synchroner Aufrufe multiplizieren die Ausfallzeit. Wenn der erste Dienst zur Bearbeitung einer Anfrage einen zweiten aufruft, der wiederum einen dritten aufruft, gelingt sie nur, solange alle drei verfügbar sind. Sie wartet auf jeden nacheinander. Lewis und Fowler nennen das den multiplikativen Effekt von Ausfallzeiten und zitieren die Regel des Guardian, pro Nutzeranfrage nur einen synchronen Aufruf zuzulassen. Halten Sie den synchronen Pfad so kurz wie in der Notiz: Nur die Reservierung wartet auf eine Antwort, alles danach ist eine Nachricht. Geben Sie jedem verbleibenden Aufruf ein Timeout und einen Circuit Breaker. Lewis und Fowler weisen für diese Muster auf das Buch Release It! hin. Ein ausfallender Dienst wird dann abgetrennt, statt auf ihn zu warten.

Dienste, die sich immer gemeinsam ändern. Eine Aufteilung an falschen Grenzen erzeugt Dienste, die sich nicht einzeln veröffentlichen lassen: Jedes Feature berührt drei von ihnen, und sie werden im Gleichschritt ausgeliefert. Lewis und Fowler geben den Test: Werden zwei Dienste ständig zusammen geändert, sollten sie zusammengelegt werden. Ein verwandter Fehler macht aus Methodenaufrufen eins zu eins entfernte Aufrufe. Die Dienste plaudern dann über das Netzwerk und sind langsam. Ziehen Sie Grenzen um Geschäftsfähigkeiten und abgegrenzte Kontexte, entwerfen Sie gröber geschnittene Schnittstellen und gestalten Sie jeden Dienst so, dass er Änderungen der aufgerufenen Dienste verträgt. Ein bereitstellender Dienst kann sich dann weiterentwickeln, ohne die aufrufenden Dienste zu beeinträchtigen. Lewis und Fowler behandeln Versionierung als letztes Mittel.

Zwei Dienste teilen eine Datenbank. Liest und schreibt der Bestelldienst direkt in den Bestandstabellen, wird das Schema zu einer Schnittstelle ohne Verantwortlichen. Keines der Teams kann es allein ändern, und der Bestand erhält einen zweiten Schreiber, den die Prüfungen des Bestandsdiensts nie sehen. Geben Sie jedem Datenelement einen Eigentümer, wie die Notiz dem Bestand einen gibt. Lassen Sie andere Dienste diesen Eigentümer fragen oder seine Nachrichten lesen. Erstreckt sich eine Änderung über Dienste, verwenden Sie eine Saga mit ausgleichenden Schritten statt einer verteilten Transaktion.

Niemand kann eine Anfrage verfolgen. Eine Anfrage durch mehrere Dienste hinterlässt in jedem Log ein Fragment ihrer Geschichte. Lewis und Fowler geben Echtzeitüberwachung einen zentralen Platz: technische Zahlen wie Datenbankanfragen pro Sekunde neben Geschäftszahlen wie Bestellungen pro Minute. Sie warnen auch, dass Dienste, die auf gegenseitige Ereignisse reagieren, emergentes Verhalten erzeugen, das die Überwachung schnell erkennen muss. Verfolgen Sie jede Anfrage. Im OpenTelemetry-Modell ist ein Trace der Pfad einer Anfrage durch das System, aufgebaut aus Spans mit gemeinsamer Trace-ID und Verweis auf den übergeordneten Span. Kontextweitergabe ermöglicht es, Spans verschiedener Dienste zu einem Trace zusammenzufügen. Geben Sie den Trace-Kontext bei jedem Aufruf und mit jeder Nachricht weiter, auch über die Outbox, sonst endet der Trace am ersten asynchronen Übergang.

Weiterführende Lektüre

- Microservices — James Lewis und Martin Fowler, 2014: die neun Merkmale, die Namensgeschichte, die Infokästen zu Größe und synchronen Aufrufen sowie die Gründe für Vorsicht.

- Wie erfinden Komitees? — Melvin Conways Beitrag von 1968 in Datamation, nach dem die Struktur eines Systems die Kommunikationsstruktur der entwerfenden Organisation kopiert.
- Referenz zum domänengetriebenen Entwurf — Eric Evans' Zusammenfassung der Muster einschließlich abgegrenztem Kontext und Karten der Beziehungen zwischen Kontexten.
- OpenTelemetry: Traces — Spans, Trace-IDs, übergeordnete Spans und Kontextweitergabe, die eine Anfrage über Dienstgrenzen hinweg sichtbar machen.
- PostgreSQL: WITH-Abfragen — datenändernde Anweisungen in `WITH`, mit denen eine Anweisung eine Zeile ändern und die Änderung anderswo aufzeichnen kann.

Glossar

Für Ihre tatsächlichen Rahmenbedingungen auswählen

Beschreiben Sie den Arbeitsablauf, die vorhandenen Schnittstellen und die Nutzer, die die Veränderung benötigen. Wir können eine prüfbare Lieferung definieren und die Übergabe so gestalten, dass Ihr Team die gelieferten Bestandteile warten kann.

Preise ansehen Kontakt aufnehmen →

Die Begriffe praktisch nutzbar machen

Fragen Sie, wie „Was sind Microservices?“ auf Ihre Organisation zutrifft. Ein konkretes Beispiel aus Ihrem bestehenden System reicht aus, um das Gespräch zu beginnen.

Unsere Leistungen entdecken

welcome@engineer.company +45 50 10 61 84 Kopenhagen, Dänemark

Nehmen Sie diese Seite mit

Kontaktkarte

Mit Sorgfalt und Liebe gestaltet von D.

CVR 45969703 · Tagensvej 105C, 3. th, 2200 København N

Copyright © 2025 – heute · Engineer ApS · Alle Rechte vorbehalten.

Gedruckt von https://engineer.company/de/glossary/microservices/