Hochschule
Albstadt-Sigmaringen

Albstadt-Sigmaringen University

Microservice Architecture

Dipl. Ing. Sven Eppler (FH)
sodge IT GmbH

Wie funktioniert eigentlich Netflix?

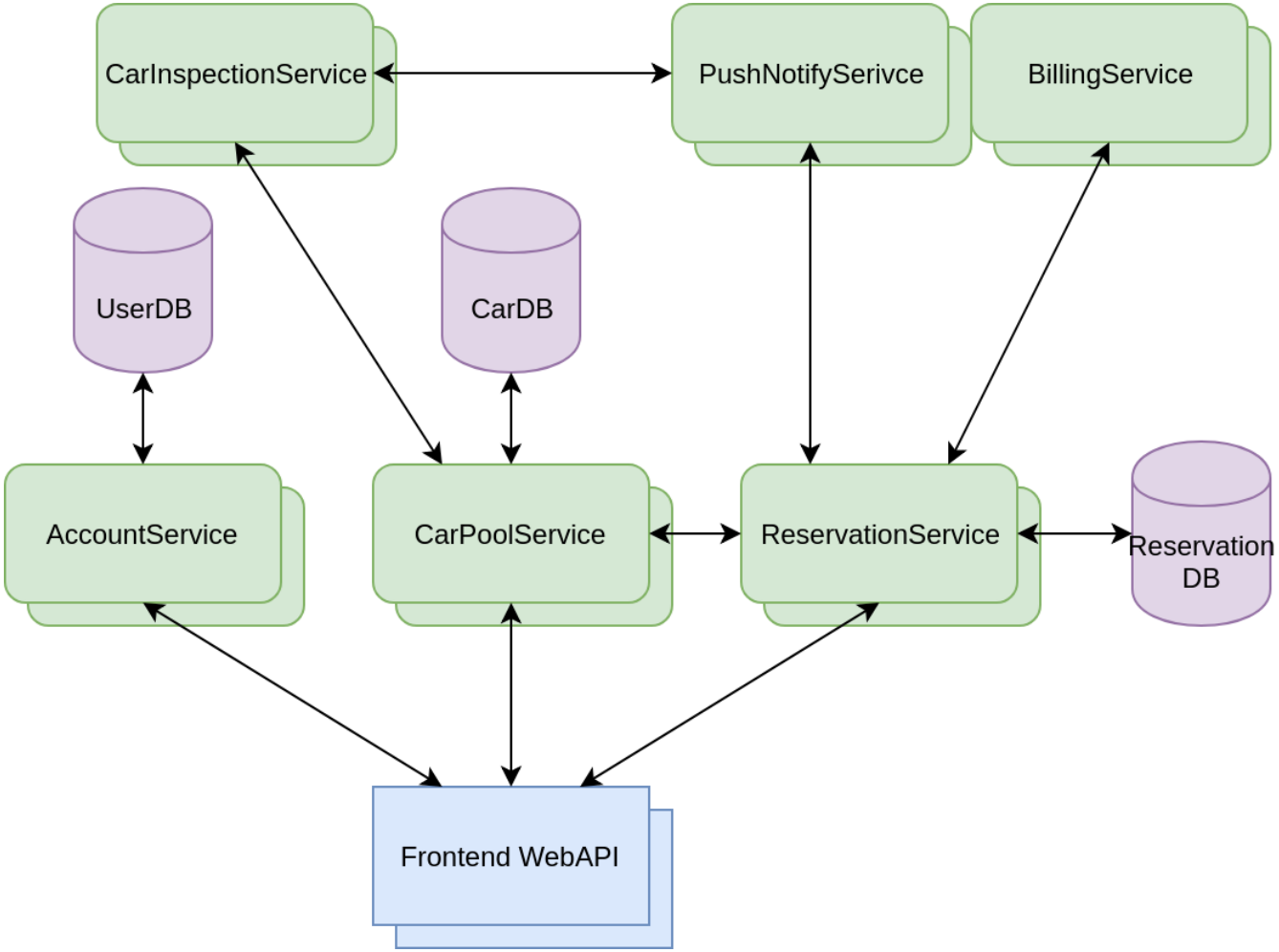
Microservice all the things!

- **Die Idee:**
  Aufteilen der Verantwortlichkeiten in kleine, selbständige Programme
  - (Vgl. Unix-Philosophie: „Do one thing and do it well“)
- **Microservices sollen möglichst stateless sein**
- **Microservices kommunizieren untereinander um Abhängikeiten aufzulösen**
- **Einzelne Services lassen sich beliebig skalieren**
  - Eine Instanz, zehn Instanzen, 1000 Instanzen
- **Skalierung über „Bare Metal“-Grenzen hinweg**

Microservice all the things!

- **Einsatz von unterschiedlichsten Technologie/Sprache im Gesamtsystem (Python, C++, Perl, Java, Rust, Go, Erlang)**
  - Unterschiedliche Entwickler konzentrieren sich auf unterschiedliche Bereiche der Software
- **Deployments können feingranulierter getestet werden (z.B. 50/50 Update)**
- **Erhöhen der Ausfallsicherheit**
  - Eine abgestürzte Instanz legt nicht die Anwendung lahm
  - Retry-Strategien bei fehlerhaften Antworten
  - Zero-Downtime-Upgrades möglich

Microservice Architektur-Beispiel



Microservice Architektur-Beispiel // Verantwortlichkeiten

- **CarInspectionService**
  - Wann muss ein Fahrzeug zur Inspektion?
- **PushNotifyService**
  - Nutzern via PushNotifications updates schicken
- **BillingService**
  - Die Abrechnung einer Fahrzeugmiete erstellen
- **ReservationService**
  - Die Buchungen von Fahrzeugen verwalten
- **AccountService**
  - Die Benutzeraccounts verwalten
- **CarPoolService**
  - Den Überblick über die Fahrzeuge und Ihre Standorte behalten

Microservice Architektur-Beispiel

- **Schnittstellen**
  - Die Frontend WebAPI stellt eine REST-Schnittstelle zur Verfügung
  - Weitere Schnittstellen können die selbe Infrastruktur nutzen (z.B. Commandline-Tools, SupportTeam GUI-Anwendungen)
- **Datenhaltung**
  - Jeder Service nutzt eine für seine Anwendung optimierte Datenhaltung
  - Schema-Upgrades im einen Service betreffen keine anderen Services
  - Öffentliche Schnittstellen zwischen den Services müssen eingehalten werden

Wie wird man Herr über 1000 Microservices?

Von Bare-Metal zu Virtualisierung zu Containern

- **Klassische Server waren echte Maschine**
  - Skalierung erfordert echte Man-Power
  - Firmen wie Google, Amazon und Microsoft betreiben mittlerweile Server im Millionenbereich
    1 Mio Server installieren bedeutet: 10 Jahre, jeden Tag ~300 Server
  - Plus Konfiguration und Einrichtung, Redundanz, etc.
- **Danach Hardware Virtuallisierung mithilfe von VirtualMachines**
  - Es gibt ein Host-System das die Resourcen verwaltet
  - Jede VM hat ein eigenes Betriebssystem
  - Alle VMs und das Host-System teilen sich die Hardware
  - Flexiblerer Einsatz des Host-Systems für verschiedenste Server-Aufgaben
  - Automatisierung durch aufspielen von VM-Images
  - Failover-Szenarien waren leichter zu realisieren
  - Z.B: VirtualBox, VMWare ESXi, Hyper-V

Von Bare-Metal zu Virtualisierung zu Containern

- **Die nächste Stufe ist Betriebsystem Virtualisierung**
  - Das Host-System managed isolierte Prozesse in Containern
  - Software wird in Containern „eingesperrt“
  - Alle Container teilen sich den selben Betriebssystem-Kernel
  - Container basieren auf Basisimages und einem „Layered Filesystem“ mit „copy on write“
  - Container lassen sich als Instanzen (z.B. auch in unterschiedlichen Versionen) beliebig starten und beenden
  - Anforderung: Linux-Box mit Docker. Danach kann ein Server einfach „alles“ sein
  - z.B. LinuxContainers (LXC), Docker, Docker-Swarm

Orchestrierung von Containern

- **Unter Orchestrierung versteht man das**
  - Starten von Containern
  - Überwachen von Containern
  - Einhalten von Abhängikeiten von Containern
  - „Entdecken“ von abhängigen Containern (ServiceDiscovery)
  - Loggen von Events und Logausgaben von Containern
  - Managen von persistentem Speicher
  - Beenden von Containern
- **Tools zur Orchestrierung**
  - Docker Swarm
  - Docker Compose
  - Kubernetes

Docker Live Demo

- **Docker getting started:**
  https://docs.docker.com/get-started/
- **Im Repo unter:**
  - ./Beispiele/Docker/SimpleWebApp

Service Discovery

- **Wie finden sich isoliert von einander laufende Services in Containern?**
- **DNS**
  - Der DNS-Server wird einfach nach einem Service-Name gefragt und liefert die IP-Adresse zurück
- **ServiceDiscovery-Registry**
  - Jeder Service meldet sich nach dem Start bei der Zentrale
  - Die Services schicken sog. „Heartbeats“ z.B. 1x pro Minute
  - Fragende Services bekommen die Antwort von der Zentrale

Load Balancing

- **Frontend-Proxies**
  - Routen traffic je nach Auslastung zu anderen Backend-Services
  - Frontend-Proxies müssen trotzdem hohe „Durchgangslast“ aushalten
  - Können auch als „harte Weiche“ benutzt werden
    (Siehe BlueGreen Deployment)
- **DNS/ServiceDiscovery-Registry**
  - Der jeweilige Service-Discovery-Mechanismus liefert immer wieder andere IP-Adressen für den selben Service
  - Dadurch schickt der fragende Service seine Anfrage immer wieder an andere Services im Backend

Service Health/Monitoring

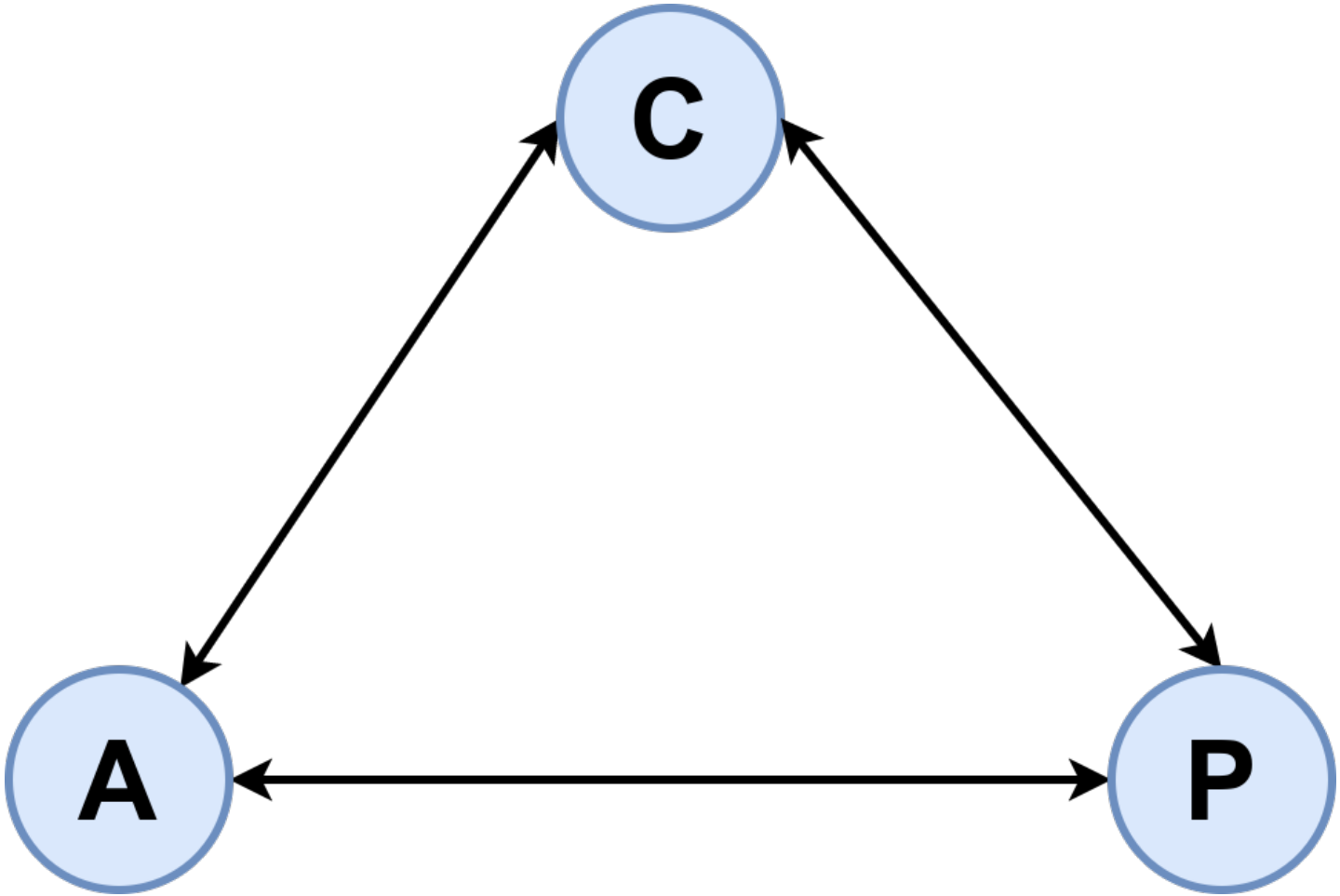
- Ausfallende Service-Instanzen sind „normal“
- Wichtig ist, diese Ausfälle zu registrieren und zu kompensieren
- Der Orchestrator hat z.B. die Aufgabe 10 Instanzen von Service1 bereit zu halten. Stirbt eine Instanz, startet er eine neue als Ersatz
- Gleichzeitig sollte die „fail rate“ gemessen werden. Also wie viele Instanzen sterben pro Zeit
  - Die „fail rate“ kann dann auch zur Bewertung von Upgrades genutzt werden: „failt“ die neue Version des Services öfter als die alte?

Verteilte Systeme
brauchen verteilte Lösungen

Eventual Consistency

- **Um einen "data monolith" zu verhindern, bekommt jeder Micrservice seine eigene Datenhaltung**
- **Daher dauert es, bis das gesamte System über alle Information verfügt**
- **Beispiel: Amazon Preis-Updates**
  - Bevor kein Preis für ein Produkt genannt werden kann, wird der "alte" Preis genannt
  - Nach und nach erfahren alle Instanzen vom "neuen" Preis
  - Nach einer gewissen Zeit werden alle Anfragen mit dem "neuen" Preis beantwortet

CAP-Theorem



CAP-Theorem

- **C = Consistency**
  - Die Daten sollen immer Konsistent sein (z.B. ACID bei SQL)
- **A = Availability**
  - Die Dienste sollen immer Verfügbar sein
- **P = Partition tolerance**
  - Die Dienste sollen weiter funktionieren wenn teile Ausfallen / untereinander nicht mehr kommunizieren können
- **Aussage CAP-Theorem:**
  - Es können nur zwei der drei Eigenschaften gleichzeitig eingehalten werden! Man muss sich daher entscheiden.

Optimistic Concurrency

- **Wenn zwei Services auf die selben Daten schreiben, wie erkennt man Konflikte?**
  - Klassische Datenbank mit Table/Row Locking → Konsitenz sequentialisiert die Requests, geringer througput!
- **Ein System das viele Requests gleichzeitig abarbeiten, ändert nur vergleichsweise selten gleichzeitig die selben Daten!**
  - Beispiel: Facebook. 2.2Mrd User, jeder bearbeitet aber "nur" sein Profil, kaum Schreibkonflikte.

Optimistic Concurrency

- **Idee: Änderungen an Daten, die sich seit dem Abrufen nicht geändert haben, sind konfliktfrei.**
  - Jeder Datensatz hat eine Version/Revision
  - Bem schreiben einer Änderung wird die gelesen Version/Revision als Bedingung gesetzt
  - Beim erfolgreichen schreiben wird die Version/Revision entsprechend erhöht
  - Schreibt jetzt ein zweiter Prozess der die "alte" Version/Revision geladen hat Daten zurück ins System, wird der Konflikt erkannt

Deploy your code

Deployment

- **Unter dem Begriff „deployment“ versteht man das ausrollen neuer Versionen von Anwendungen**
- **Im Microservice-Umfeld also das ausrollen einer neuen Version eines Services**
- **„Continious Deployment“ beschreibt das kontinuirliche Ausrollen neuer Versionen**
  - Traditionelles deployment findet selten statt, wird daher wenig oft „getestet“ und hat eine hohe Fehleranfälligkeit
  - CD macht das ausrollen zu „everyday business“, Fehler im Deployment werden früh gefunden.
  - Fehlerbehebungen in der laufenden Software werden schnell zum Kunden geliefert

Deployment Strategien

- **BigBang-Deployment**
  - Die neue Version wird als ganzen (alle Services, alle Frontend, alle Datenbank) in einem Zug ausgerollt
  - Tritt ein Fehler dabei auf, kommt es zum „Big Bang“ und alles steht
  - Zero-Downtime-Upgrades kaum möglich, da das Gesamtsystem gestoppt, getauscht und neu gestartet werden muss
- **Rolling**
  - Es wird „rollend“ eine Instanz nach der andere mit dem Update versehen
  - Ist das Upgrade von einer Instanz erfolgreich, wird direkt mit der nächsten begonnen, bis alle upgrades abgschlossen sind
  - Zero-Downtime-Upgrades sind dadurch möglich
  - Erlaubt kein echtes Erproben der neuen Version im Produktivumfeld, da direkt weiter „deployed“ wird

Deployment Strategien

- **BlueGreen / A-B**
  - Es existieren zwei identische Systeme das „blaue“ und das „grüne“
  - Am Anfang erreichen Kunden nur das „blaue“ System. Auf dem „grünen“ System wird die nächste Version entwickelt
  - Ist die Version fertig, wird der Loadbalancer umgeschaltet, Kunden landen ausschließlich auf den „grünen“ System, das „blaue“ wird upgedated und ist jetzt das neue Entwicklungssystem
  - Zero-Downtime-Upgrades möglich, der Loadbalance-Switch ist aber ähnlich wie „Big Bang“-Deployment

Deployment Strategien

- **Canary**
  - Nur einige Instanzen des bestehenden Systems werden durch neue ersetzt
  - Danach wird das Gesamtsystem beobachtet (z.B. „fail rates“)
  - Und basierend darauf sukzessive weitere Instanzen getauscht oder zurück gerollt
  - Zero-Downtime-Upgrade möglich
  - Deployment-Fehler haben geringe Auswirkung auf das Gesamtsystem
  - Nur einsetzbar bei „unkritischer“ Infrastruktur

Zentrales Logging

- **Jede Service-Instanz im System erzeugt eigene Logs**
- **Diese Logs sollten an einer zentralen Stelle gesammelt und durchsucht werden können**
  - Schwierigkeit: Ein einziger Request wandert z.B. durch drei Services. Um den Request zu verfolgen, müssen auch die Logs aller drei Services vorhanden sein
  - Zentrales Logging in Kombination mit einer „Request-ID“ ermöglichen das tracen solcher Requests zur Fehlerdiagnose
- **Insbesondere für Canary-Deployments sehr wichtig**
  - Warum ist die neue Instanz abgestürzt?

Microservice Beispiel

- Repo: /Beispiele/Node.js/Microservices/
- README.md beachten!

Microservice Beispiel Overview