CLOUD NATIVE COMPUTING

CLOUD-NATIVE

Unit:

Serverless Architectures

(2) Architekturen + API-Gateways

creative commons

CLOUD NATIVE COMPUTING

Urheberrechtshinweise

Diese Folien werden zum Zwecke einer praktikablen und pragmatischen Nutzbarkeit im Rahmen der **CCo 1.0 Lizenz** bereitgestellt.

Sie dürfen die Inhalte also kopieren, verändern, verbreiten, mit eigenen Inhalten mixen, auch zu kommerziellen Zwecken, und ohne um weitere Erlaubnis bitten zu müssen.

Eine Nennung des Autors ist nicht erforderlich (aber natürlich gern gesehen, wenn problemlos möglich).

Diese Folien sind insb. für die Lehre an Hochschulen konzipiert und machen daher vom **§51 UrhG (Zitate)** Gebrauch.

Die CCo Lizenz überträgt sich nicht auf zitierte Quellen. Hier sind bei der Nutzung natürlich die Bedingungen der entsprechenden Quellen zu beachten.

Die Quellenangaben finden sich auf den entsprechenden Folien.

KAPITEL 12

Serverless Architekturen

12.1 Eigenschaften von Microservices

12.2 Integrationsmuster für Microservices

12.3 Architekturelle Sicherheit

12.4 Skalierung von Microservices

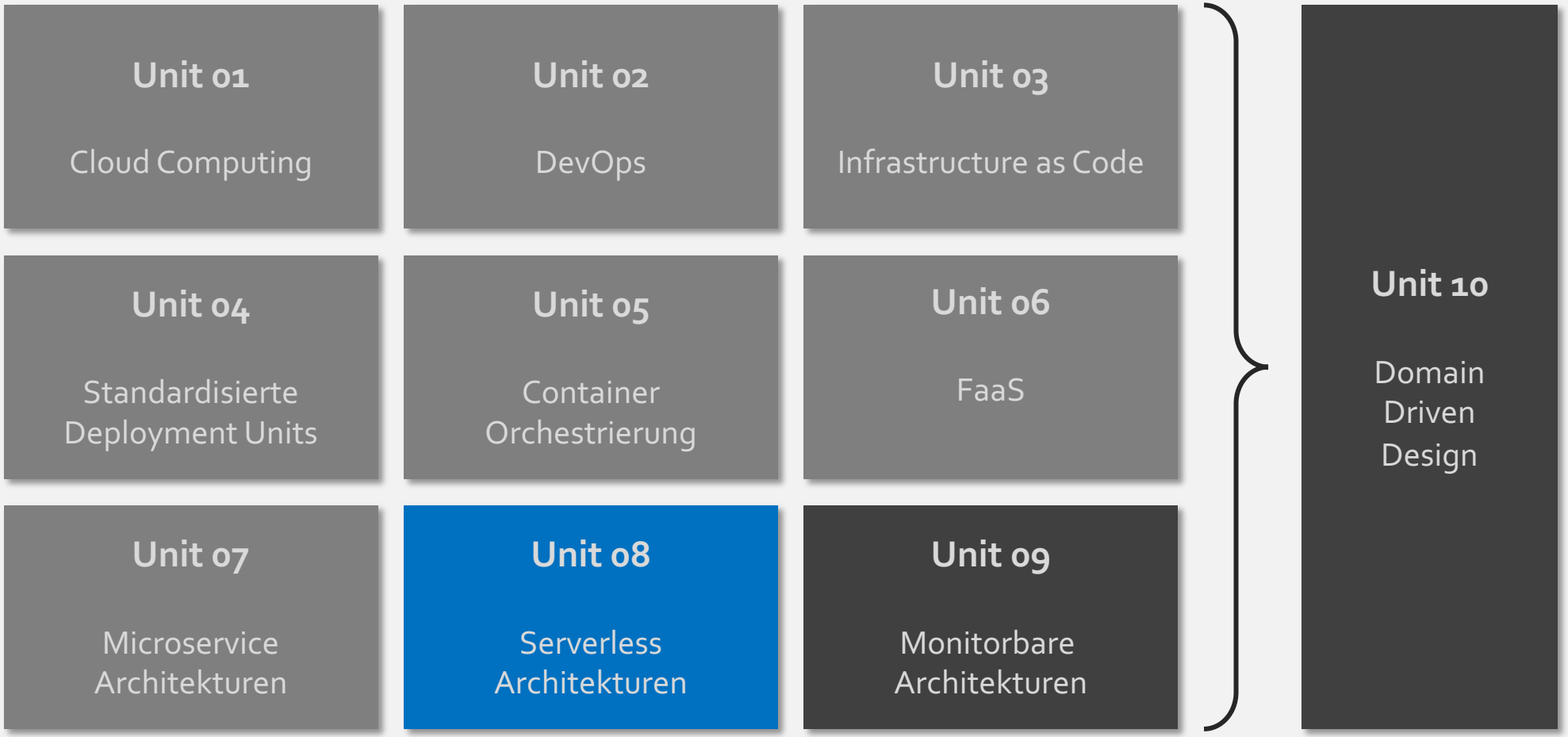
12.5 Prinzipien zur Entwicklung von Microservices

12.6 Serverless-Architekturen

- Architekturelle Konsequenzen von Serverless-Limitierungen
- Das API-Gateway Pattern
- Abgrenzung zu Microservices

INHALTSVERZEICHNIS

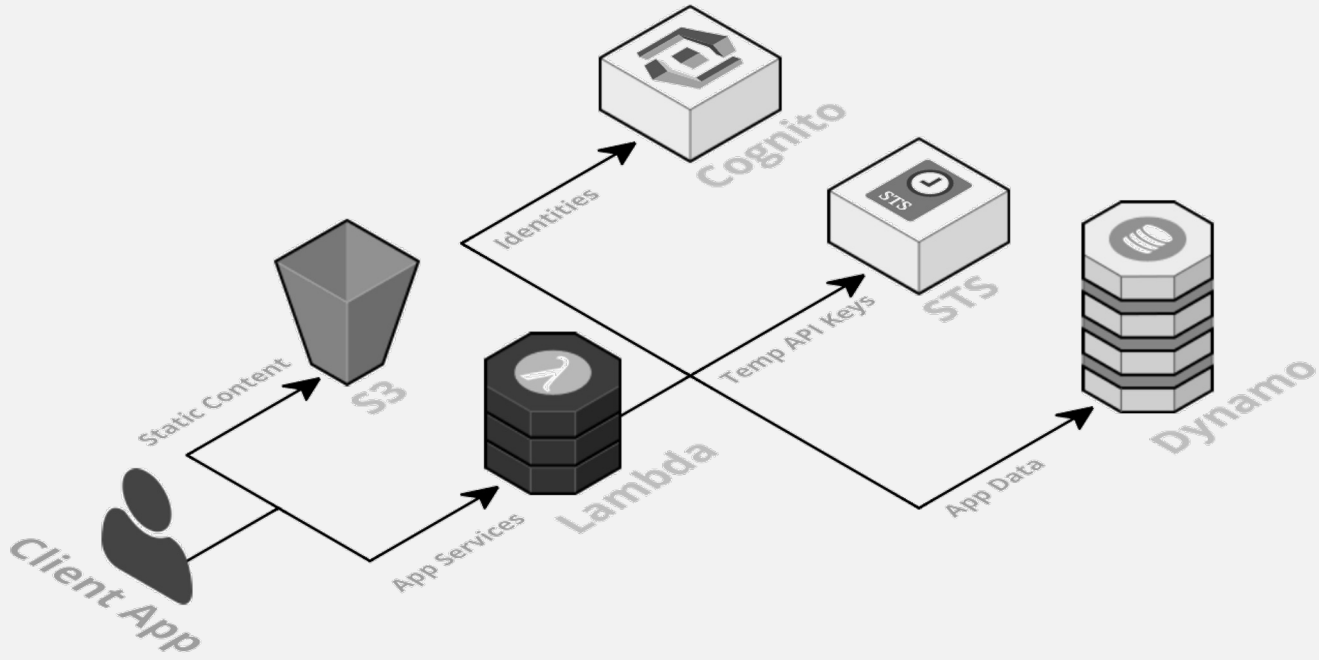
Überblick über Units und Themen dieses Moduls



CLOUD NATIVE COMPUTING

INHALTE

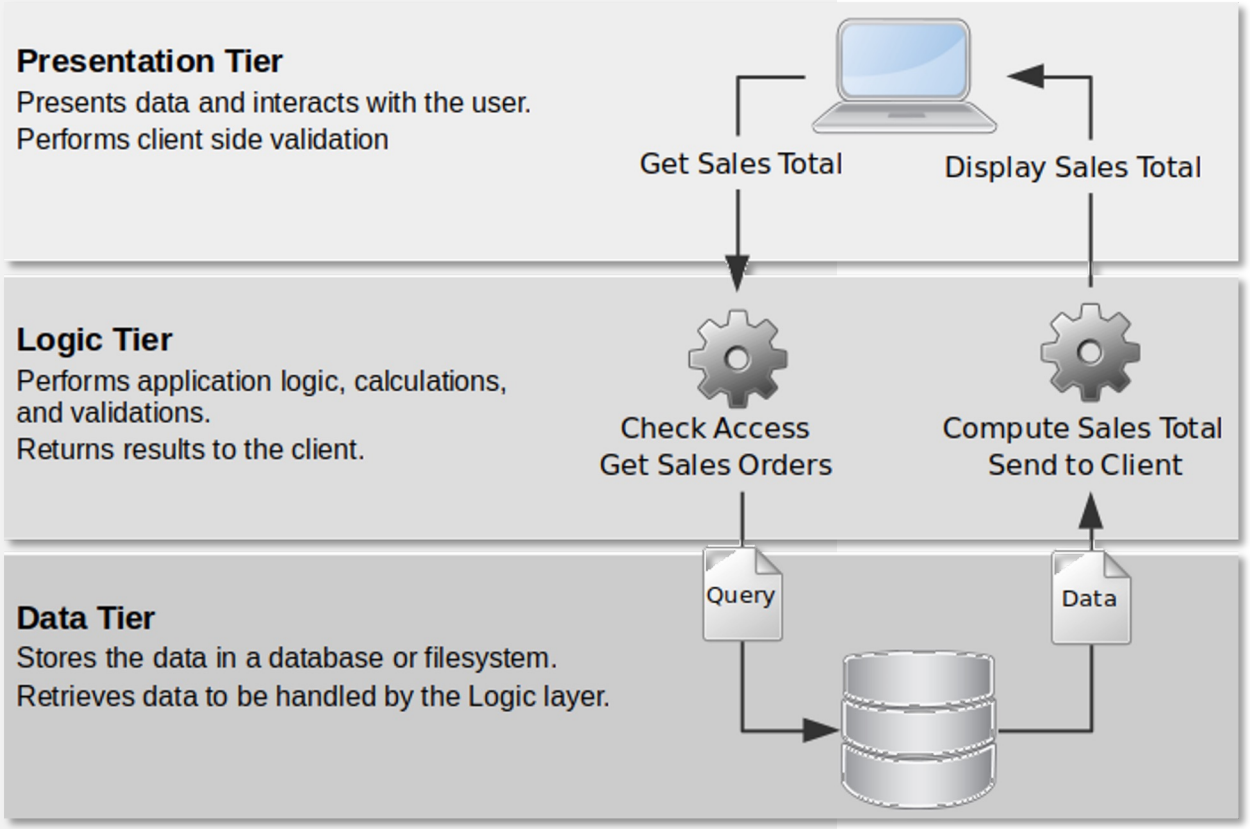
- Was ist Serverless?
- Limitierungen von Serverless
- **Serverless Architekturen**
- **Abgrenzung zu Microservices**



WIE WIRD SERVERLESS EINGESETZT?

Multitier-Architekturen

- Die Drei-Schichten-Architektur ist ein oft genutztes Muster für Anwendungen
  - Presentation Tier
  - Logic Tier
  - Data Tier
- Jede Schicht kann unabhängig voneinander skaliert werden
- Da FaaS zustandslos konzipiert ist, bietet es sich für den Data Tier nicht an
- Serverless-Konzepte werden daher meist auf folgenden Ebenen eingesetzt:
  - **Logic Tier**
  - Presentation Tier (Web Serving, eher seltener)
- Dies kann in Reinform oder auch in Kombination mit anderen Ansätzen wie Microservices oder SOA-Monolithen erfolgen

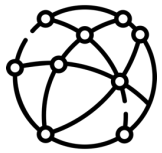


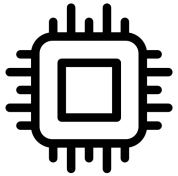
Quelle: The Odoo architecture, O'Reilly

KONZENTRATION AUF DAS WESENTLICHE

Welche Funktionalität sollte Serverless implementiert werden?

- In Serverless Architekturen werden meist nur dann Anwendungs-spezifische Funktionen bereitgestellt, wenn:
  - **Funktionalität ist sicherheitsrelevant** ist und muss vom Dienstanbieter in einer kontrollierten Laufzeitumgebung ausgeführt werden.
  - Die Funktionalität ist für Consumer-Clients oder Edge-Geräte zu **Processing- oder datenintensiv**.
  - Die Funktionalität ist so **Domänen-, Problem- oder Anwendungs-spezifisch**, dass kein externer Drittanbieter-Dienst existiert
- **Endpunkte** für Anwendungs-spezifischen Funktionen werden meist mittels **HTTP-/ REST-basierten APIs** bereitgestellt



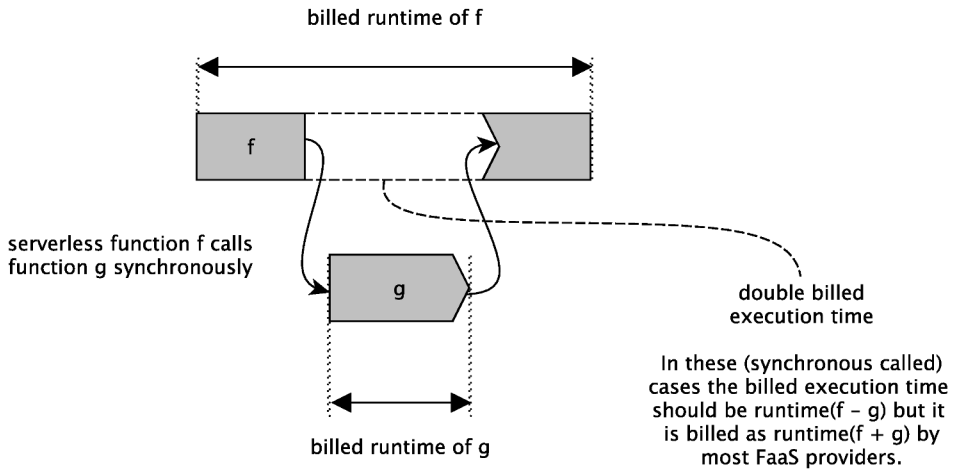


TECHNISCHE ENTSCHEIDUNGEN

In Serverless Architekturen

- Querschnittslogik (Authentifizierung, Persistenz, …) wird an **externe Drittanbietern-Dienste** delegiert
- Die Workflow-Steuerung von Services wird vom Service-Provider zum Service-Consumer verlagert und über bereitgestellte Anwendungen durchgeführt
- Client- oder Edge-Geräte verbinden Services von Drittanbietern häufig direkt (kann durch API-Gateways teilweise etwas reduziert werden)
- Diese **Consumer-Orchestrierung** hat folgende Auswirkungen:
  - Der Service-Consumer stellt die für die Service-Orchestrierung erforderlichen Ressourcen bereit
  - Da die Service-Komposition außerhalb des Bereichs der FaaS-Plattform erfolgt, werden Double-Spending-Probleme vermieden
  - **Dezentralere Gesamtarchitektur**

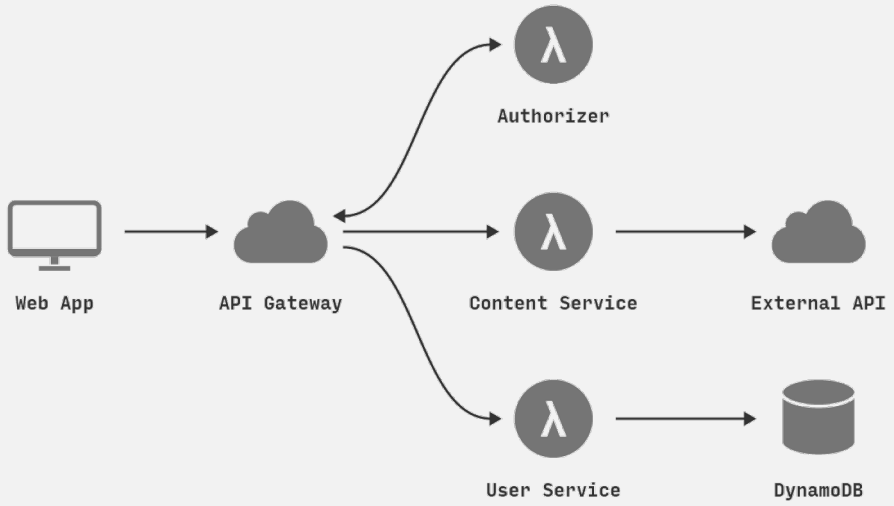
Double Spending Problem



API-GATEWAYS

Einleitung

- Ein API-Gateway ist ein Teil einer Architektur, die verwendet wird, um den **Zugriff auf mehrere APIs** zu verwalten
- Wird zwischen dem Client und den einzelnen APIs in einem System platziert
- Bietet nach außen einen **zentralen Endpunkt** an
- Anforderungen von verschiedenen Clients (Webbrowsern, mobilen Apps, …) werden an verschiedene APIs weitergeleitet
- API-Gateways erleichtern die effiziente **Verwaltung** und **einheitliche Bereitstellung** einer Vielzahl von APIs



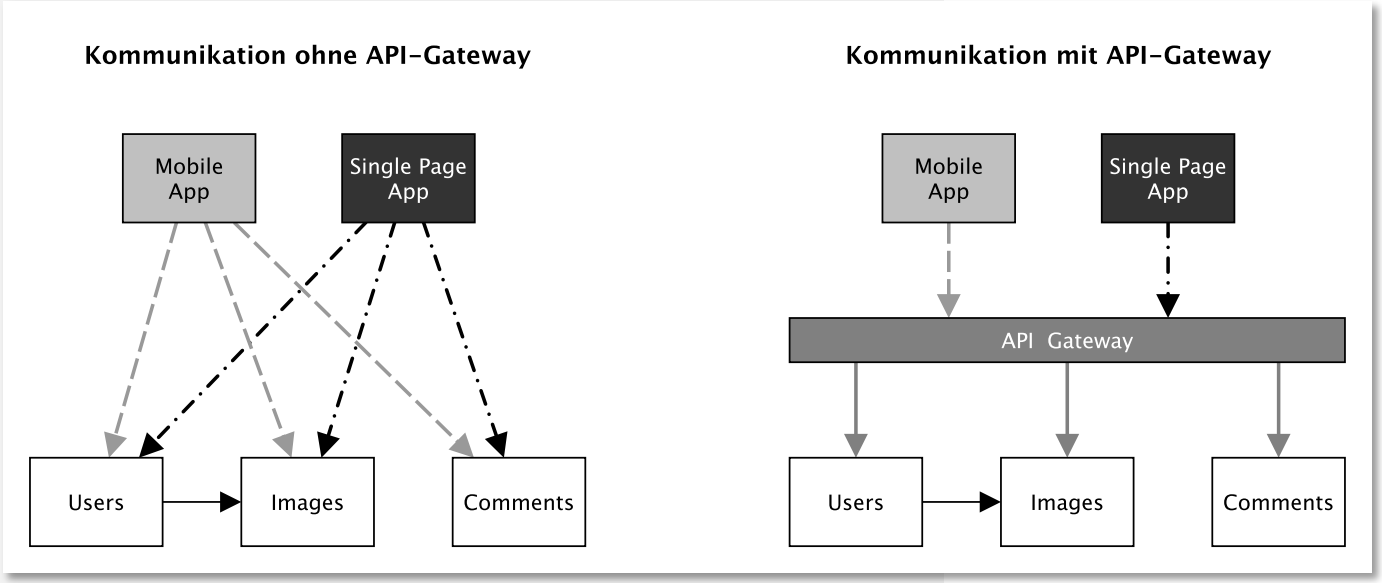
Funktionen von API-Gateways

- **Authentifizierung und Autorisierung**
  von Benutzern die auf die API zugreifen dürfen
- **Lastverteilung**
  Weiterleitung von Anfragen an verschiedene APIs, um so die Last auf mehrere Systeme zu verteilen
- **API-Management**
  Schnittstelle zur Verwaltung von APIs (hinzufügen, aktualisieren, löschen)
- **Caching**
  Proxy-Cache um die Anzahl wiederkehrender Anfragen an APIs zu reduzieren
- **Rate Limiting**
  Für eine API wird eine maximale Anzahl von Anfragen pro Zeiteinheit festlegt, die ein Client senden darf

API-GATEWAYS

Adressieren Nachteile von Service-Architekturen

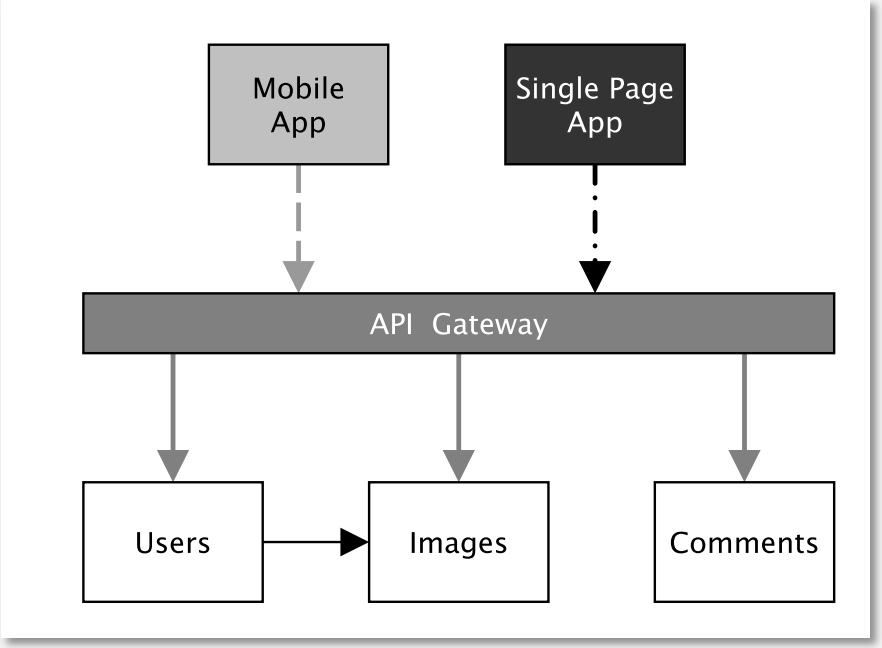
- **Viele APIs/Schnittstellen für Clients**
  werden dann problematisch, wenn sich die Backends verändern
- **Größere Angriffsfläche**
  Eigentlich interne Schnittstellen von Services müssen für Clients auch von extern erreichbar sein
- **Querschnittliche Belange (Cross-cutting)**
  Häufig erforderliche und wiederkehrende Funktionalitäten sind pro Service zu definieren
  - Authentifizierung
  - SSL-Terminierung
  - Rate Limiting und Throttling
  - Same-Origin-Policy (CORS, Cross.Origin-Resource- Sharing)



API-GATEWAYS

Einsatzprinzipien

- Die Kommunikation der Clients zu den Upstream Services verläuft immer über das API-Gateway als **einheitlichem Endpunkt**
- Die Clients müssen folglich auch nur noch eine Adresse kennen
- Somit ist eine Hauptaufgabe eines API-Gateways das **Reverse Proxying**
- In dieser zentralen Stelle lassen sich **querschnittliche Belange** dann zentral umsetzen, sodass diese nicht von jedem Service separat bereitgestellt werden müssen, wie bspw. einheitliche CORS-Regeln (Same-Origin-Policy) oder Authentication



API-GATEWAY

Ja oder nein?

- Kernargumente für ein Gateway sind die Einfachheit für Clients und die Kapselung der Service-Architektur nach außen
- Bei mehr als einen Endpunkt, ist es oft hilfreich diese zu bündeln
- Die Verwaltung von "Endpunktlisten" in Clients sollte vermieden werden
- Da Architekturen sich gem. der DevOps Philosophie evolutionär entwickeln, werden sich im Laufe der Zeit auch die Endpunkte immer wieder ändern
- API-Gateways vermeiden, dass Clients Änderungen direkt ausgesetzt werden und ermöglichen es, APIs kompatibel zu halten, selbst wenn sich die Service-Landschaft sich immer wieder verändert

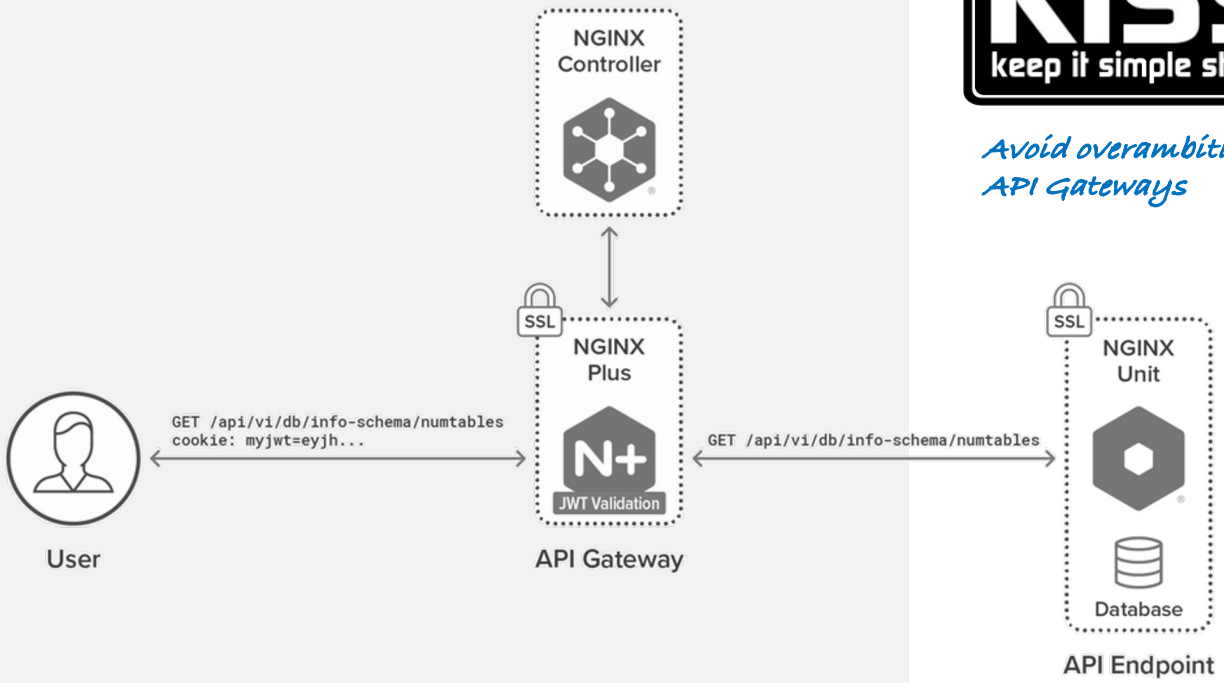
Zu berücksichtigende Randbedingungen:

- Ein API-Gateway ist ein konzeptioneller **Single Point of Failure**, der in die Architektur eingebracht wird.
- Damit ergibt sich eine Reihe von Anforderungen an die Verfügbarkeit, Belastbarkeit und an die Fähigkeiten des Gateways.
- Ein Gateway muss mindestens so verfügbar sein, wie die Höchstanforderung an einen beliebigen Service dahinter.
- Es muss auch alle Kommunikationstechnologien unterstützen, welche die aufgerufenen Services benötigen (beispielsweise HTTP/2 oder Websockets)

API-GATEWAY

Empfehlungen zum „richtigen" Einsatz

- Einige API-Gateway Produkte versprechen unzählige Features mit entsprechender Auswirkung auf die Komplexität
- Ein **Austausch** eines API-Gateways ist meist problemlos möglich und für die Clients völlig **transparent**
- Daher kann man exakte Anforderungen abwarten und komplexere Lösungen erst **bei Bedarf** einführen
- Man sollte vermeiden, **Geschäftslogik** oder "Features" aus den Services in das Gateway zu verschieben
- Das führt oft zu unübersichtlichen und schwer zu wartenden Systemen mit vielen Abhängigkeiten zum API-Gateway



Einfache API-Gateways lassen sich auch mit Reverse Proxies wie bspw. NGINX realisieren. NGINX ist der Standard Ingress Controller in Kubernetes. Anders ausgedrückt. Ein Kubernetes Ingress kann oft als API-Gateway schon ausreichen.

KISS keep it simple stupid

Avoid overambitious API Gateways

VOR- UND NACHTEILE

Von Serverless Architekturen

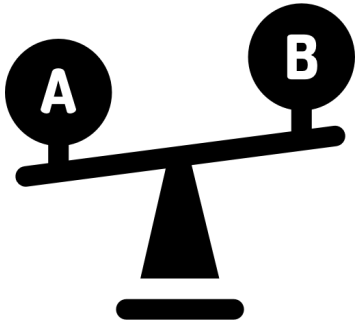
Vorteile

- **Skalierbarkeit**
  Services einfach zu skalieren, da Cloud-Provider automatisch Ressourcen bereitstellen und bedarfsgerecht skalieren
- **Kosten**
  Services sind oft kosteneffektiver, da nur für tatsächlich genutzte Ressourcen abgerechnet werden (Scale-to-Zero)
- **Wartung**
  Services sind oft einfacher zu warten, da Cloud-Provider die zugrunde liegende Infrastruktur und die Wartung übernimmt

Nachteile

- **Kaltstartprobleme**
  Services können eine Verzögerung beim ersten Start aufweisen, da die erforderlichen Ressourcen erst angefahren werden müssen
- **Limitierungen**
  Services können bestimmte Einschränkungen aufweisen, wie z.B. begrenzte CPU- und Speicherkapazitäten, limitierte Laufzeiten
- **Debugging**
  Services können schwieriger zu debuggen sein, da sie aus vielen kleineren Funktionen bestehen, die in unterschiedlichen Umgebungen ausgeführt werden

CLOUD NATIVE COMPUTING



Man sollte Vor- und Nachteile von Serverless bei der Wahl geeigneter Use Cases für Serverless zu berücksichtigen.

ZUSAMMENFASSUNG

Und Abgrenzung zu Microservices

CLOUD NATIVE COMPUTING

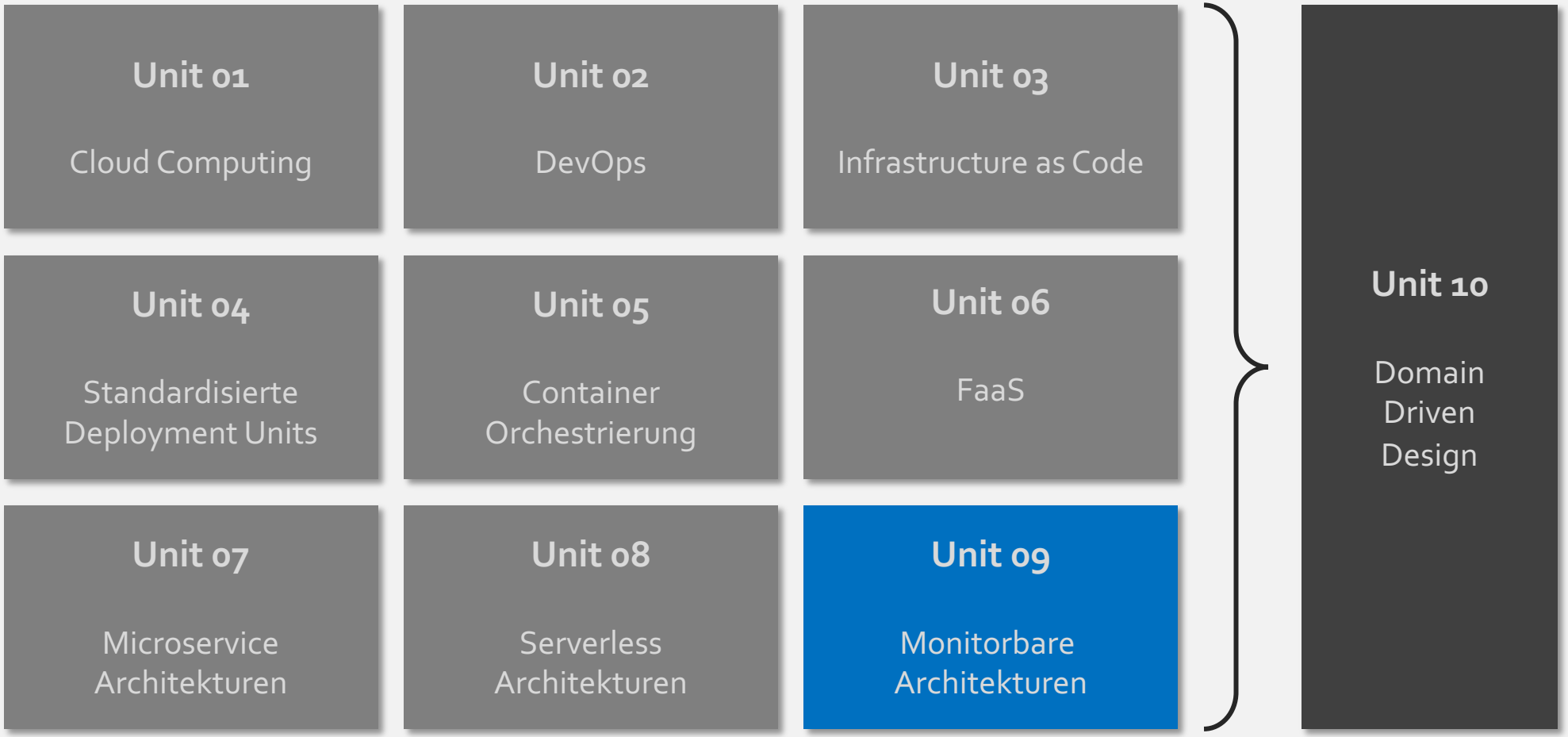
- Serverlose Architekturen sind Designs, die konsequenter als Microservices **Drittanbieter BaaS-Dienste** (Backend as a Service) nutzen
- Benutzerdefinierter Code wird auf ein Minimum reduziert (Logic-Tier) und in verwalteten, kurzlebigen Containern auf einer FaaS-Plattform (Function as a Service) ausgeführt.
- Dadurch entfällt bei Serverless-Architekturen ein Großteil der Notwendigkeit für Always-on Komponenten
- Serverless Architekturen können von **reduzierten Betriebskosten**, Komplexität und schnelleren Entwicklungszyklen (Time-to-Market) profitieren
- Dies wird allerdings durch eine erhöhte **Abhängigkeit** von Cloud-Providern und unterstützenden BaaS-Diensten erkauft

Abgrenzung zu Microservices

- Serverless und Microservice Architekturen widersprechen sich nicht.
- Sie können gemeinsam auftreten und sind kompatibel, da sie viele Gemeinsamkeiten teilen (bspw. die Präferenz für Statelessness aufgrund des Fokus auf horizontaler Skalierbarkeit).
- Microservices eignen sich für lang laufende, komplexere Dienste mit hohem Ressourcen- und Managementbedarf.
- Demgegenüber führen Serverless-Architekturen Funktionen nur bei Bedarf aus und eignen sich damit insbesondere für die ereignisgesteuerte Abläufe.
- Serverless Services können in Microservice-Architekturen als Dienste auftreten und andersherum.

AUSBLICK

Überblick über Units und Themen dieses Moduls



KONTAKT

Disclaimer

Nane Kratzke

+49 451 300-5549

nane.kratzke@th-luebeck.de

kratzke.mylab.th-luebeck.de