CLOUD NATIVE COMPUTING

CLOUD-NATIVE

Unit:

Microservices

(4) Integrationsstile

REST + GraphQL

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

Microservice Architekturen

12.1 Eigenschaften von Microservices

12.2 Integrationsmuster für Microservices

- Datenbank-basierte Integration
- gRPC, REST
- API Versioning
- Event-Driven Architectures

12.3 Architekturelle Sicherheit

- Circuit-Breaker, Bulkhead
- Idempotente API-Operationen

12.4 Skalierung von Microservices

- Load Balancing
- Messaging
- Scaling for Reads/Writes
- Command Query Responsibility Segregation (CQRS)
- Caching

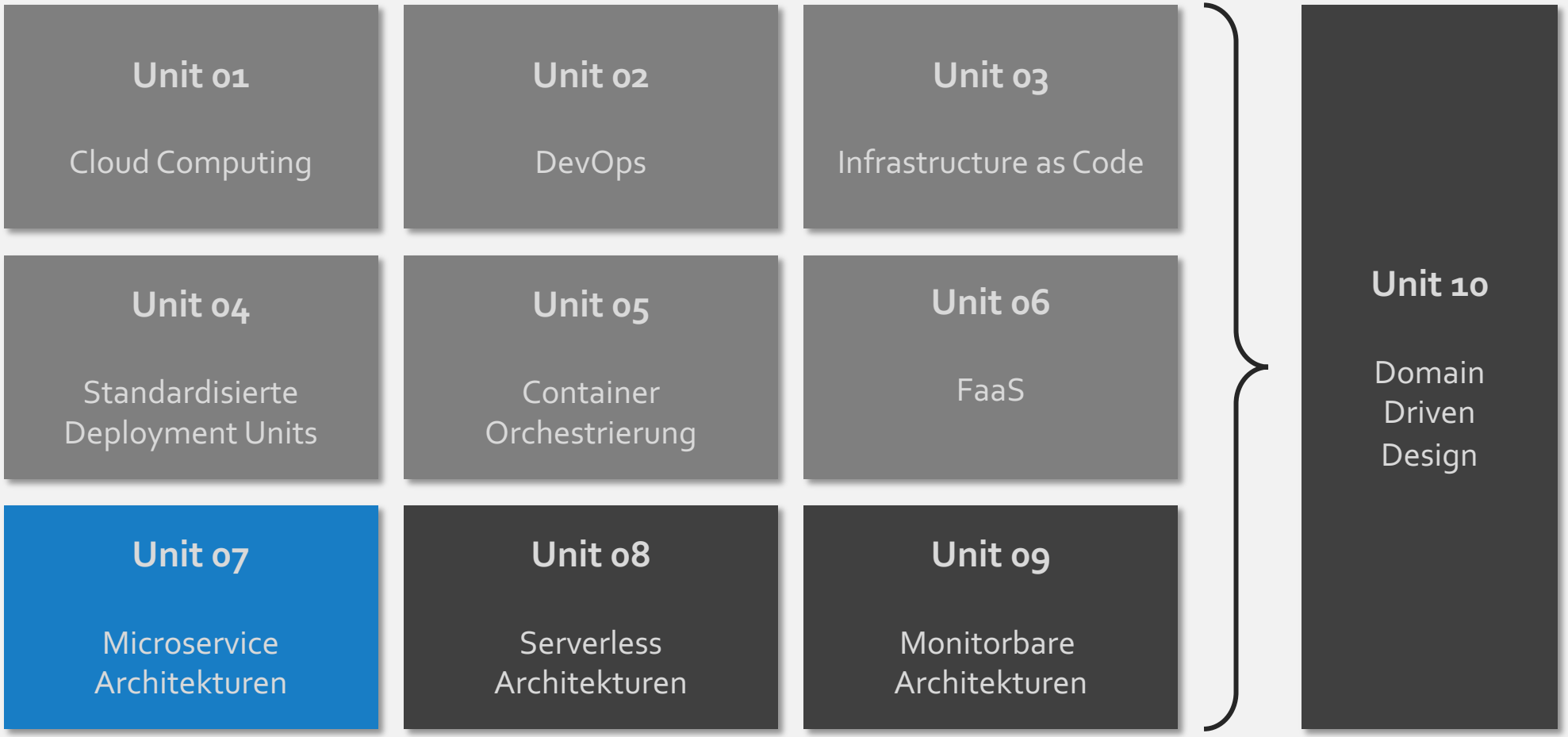
12.5 Prinzipien zur Entwicklung von Microservices

12.6 Serverless-Architekturen

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

INHALTSVERZEICHNIS

Überblick über Units und Themen dieses Moduls



INHALTE

CLOUD NATIVE COMPUTING

Microservices

- Was ist das für ein Architekturstil?
- Vor- und Nachteile von Microservices

Randbedingungen

- Lose Kopplung und hohe Kohäsion
- Bounded Context und Domain Driven Design
- Conways Law
- Service Ownership und Team-Strukturen

Integration

- Shared Databases
- (g)RPC, REST, GraphQL
- Messaging (Event-driven)

Skalierung von Microservices

- Failure is Everywhere
- Architectural Safety Measures
- Scaling
- Caching

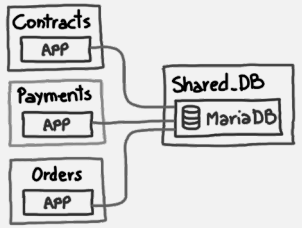
7 Prinzipien von Microservices

- Model Around Business Concepts
- Adopt a Culture of Automation
- Hide Internal Implementation Details
- Decentralize All the Things
- Independently Deployable
- Isolate Failure
- Highly Observable

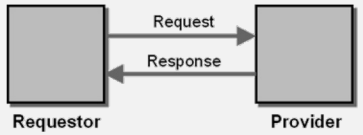
INTEGRATION

Integrationsstile

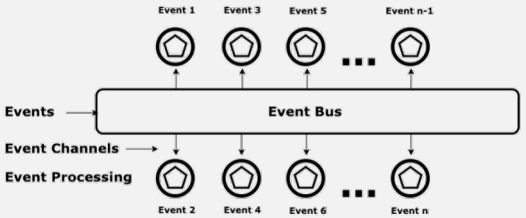
- Daten-basierte Integration
  - The shared database



- Request/Response-basierte Integration
  - Remote Procedure Calls
  - **REST**
  - **GraphQL**

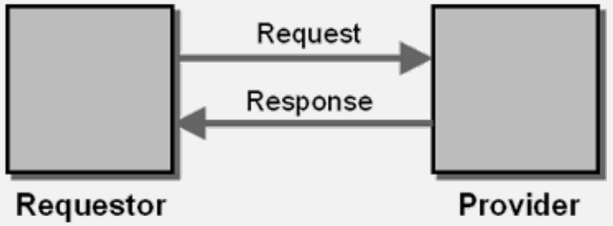


- Ereignis-basierte Integration
  - Event Driven Architectures
  - Reactive Programming



INTEGRATION

Request/Response-basierte Integration (synchron)



UDP TCP/HTTP

RPC

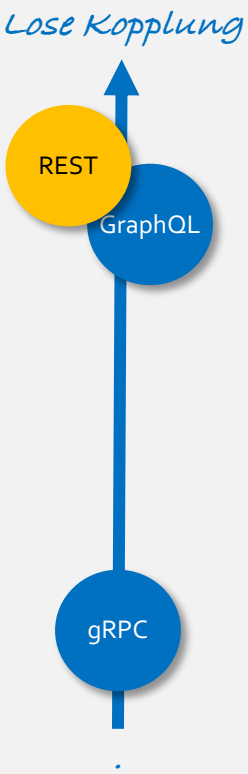
gRPC

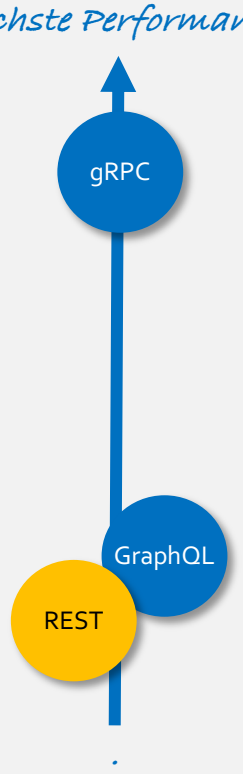
GraphQL

REST

IDL oder Schema
(erhöhen Kopplung zw. Komponenten)

Keine IDL erforderlich
(nur bei HATEOAS, ansonsten implizit)





IDL: Interface definition language

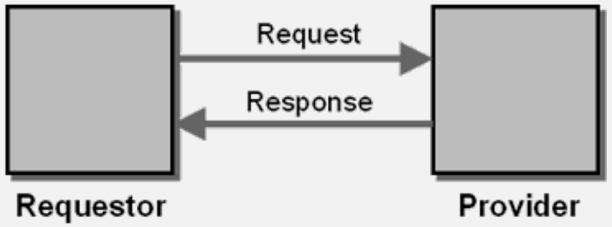
Häufig muss man also zwischen Performance und loser Kopplung abwägen.

Achtung:
Es gilt auch die „ewige" Empfehlung erst Performanceprobleme anzugehen, wenn es einen aus dem Betrieb ersichtlichen Grund dazu gibt!

INTEGRATION

Request/Response: Representational State Transfer (REST)

- REST hat das Ziel, einen Architekturstil zu schaffen, der einheitliche Schnittstellen zu **Ressourcen** im Netz ermöglicht
- Ressourcen kann dabei über verschiedene **Medientypen repräsentiert** werden (XML, JSON, HTML, ...)
- Anders als bei vielen verwandten Architekturen(SOAP, WSDL) kodiert REST allerdings keine Verarbeitung (Create, Read, Update, Delete) in Uniform Resource Identifiern (URI).
- Ein Vorteil von REST liegt darin, dass im WWW bereits ein Großteil der für REST nötigen Infrastruktur (z. B. Web- und Application-Server, HTTP-fähige Clients, usw.) vorhanden ist.
- Viele existierende Web-Dienste sind per se REST-konform.



CLOUD NATIVE COMPUTING

INTEGRATION

Request/Response: REST – 5 Prinzipien

Client-Server

1

- Server stellt Dienst bereit
- Dienst kann vom Client angefragt werden
- Einfache Skalierbarkeit der Server
- Unterschiedlich schnelle Entwicklung von Server und Client möglich

Caching

2

- HTTP Caching als Performance-Optimierung
- Gefahr: Clients greifen auf veraltete Cache-Daten zurück

Zustandslosigkeit (Stateless)

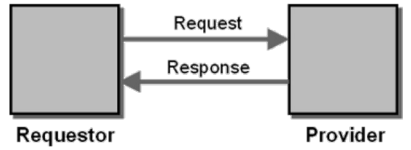
3

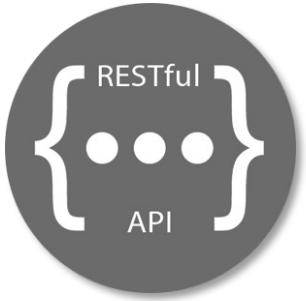
- Nachrichten enthalten alle für die Verarbeitung notwendigen Daten für Server und Client
- Keine Speicherung von Zustandsinformationen zwischen zwei Nachrichten
- Horizontale Skalierbarkeit eines Services

Mehrschichtige Systeme

4

- Mehrschichtiger Systemaufbau
- Nur eine Schnittstelle für den Anwender
- Verborgene dahinterliegende Ebenen
- Horizontale Skalierbarkeit jeder Ebene
- Schutz durch Firewalls möglich





INTEGRATION

Request/Response: REST – Einheitliche Schnittelle 5

Selbstbeschreibende Nachrichten 5.1

- REST-Nachrichten sind selbstbeschreibend (beinhalten alle erforderlichen Daten zur Verarbeitung)
- Manipulation von Ressourcen mittels HTTP-Standardmethoden (GET, POST, PUT, DELETE)

Repräsentationen von Ressourcen 5.2

- Unterschiedliche Darstellungsformate für Ressourcen
- z.B. HTML, JSON, XML
- Statusänderungen einer Ressource sollen nur über eine Repräsentation erfolgen

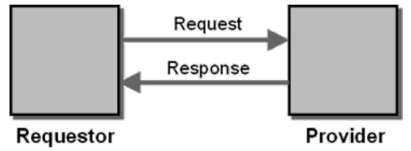
Adressierbarkeit von Ressourcen 5.3

- Jede Ressource eines Service hat Uniform Resource Identifier (URI)
- Konsistente Adressierbarkeit erleichtert die Nutzung in Service-of-Service Systemen
- Vielzahl von Anwendungen (Clients) können einheitlich auf den Service zugreifen

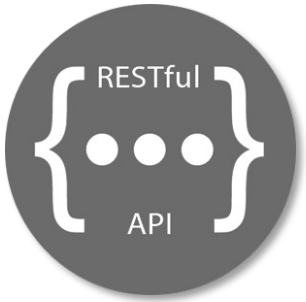
CRUD	SQL-92	HTTP (REST)
Create	INSERT	POST (seltener PUT)
Read (Retrieve)	SELECT	GET
Update	UPDATE	PATCH oder PUT
Delete (Destroy)	DELETE	DELETE

CRUD umfasst die vier grundlegenden atomaren Operationen persistenter Speicher und wird häufig wie hier gezeigt in REST-konforme Service-APIs übersetzt.
Erläuternd sind SQL-Statements relationaler Datenbanken angegeben.

CLOUD NATIVE COMPUTING



REST hat u.a. zum Ziel die Einheitlichkeit und einfache Nutzbarkeit der Schnittstelle mittels der folgenden Prinzipien zu gewährleisten.



INTEGRATION

Request/Response: REST – Beispiel (Python FastAPI-Bibliothek)

```python
from fastapi import FastAPI

app = FastAPI()

# Poor man's database
course_items = [
  {"course_name": "Python"},
  {"course_name": "SQLAlchemy"},
  {"course_name": "NodeJS"}
]

# Course resource endpoint
@app.get("/courses/")
def read_courses(start: int, end: int):
    return course_items[start : start + end]
```

Beispiel für eines GET-Requests, der Kurs-Informationen im JSON-Format abruft.

```
GET /courses?start=1&end=3
HTTP/1.1 Host: my.favourite.edu
Accept: application/json
...
```

Exemplarische Antwort

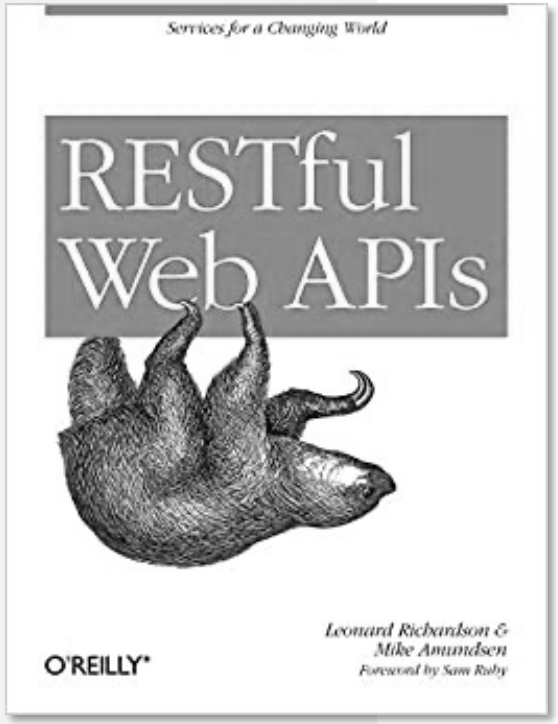
```
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: ...

[
   {"course_name": "SQLAlchemy"},
   {"course_name": "NodeJS"}
]
```

INTEGRATION

Request/Response: REST – Maturity Model

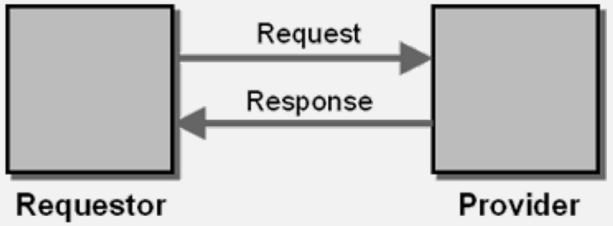
Level	Eigenschaften
0 Swamp	• verwendet **XML-RPC** oder **SOAP** • der Service wird über einen einzelnen URI adressiert • verwendet eine einzelne HTTP-Methode (oft POST)
1 Resources	• verwendet verschiedene URIs und Ressourcen • verwendet wenige HTTP-Methoden (oft POST)
2 HTTP Verbs	• verwendet verschiedene URIs und Ressourcen • verwendet mehrere HTTP-Methoden
3 Hypermedia Controls	• basiert auf HATEOAS und verwendet daher **Hypermedia** für Navigation • verwendet verschiedene URIs und Ressourcen • verwendet mehrere HTTP-Methoden
4 Code on Demand	• ausführbaren Code (z.B. JavaScript) wird an Client gesendet • der kann lokal ausgeführt werden • API kann so auf den Client angepasst werden • hohe Flexibilität in der Anwendungsentwicklung

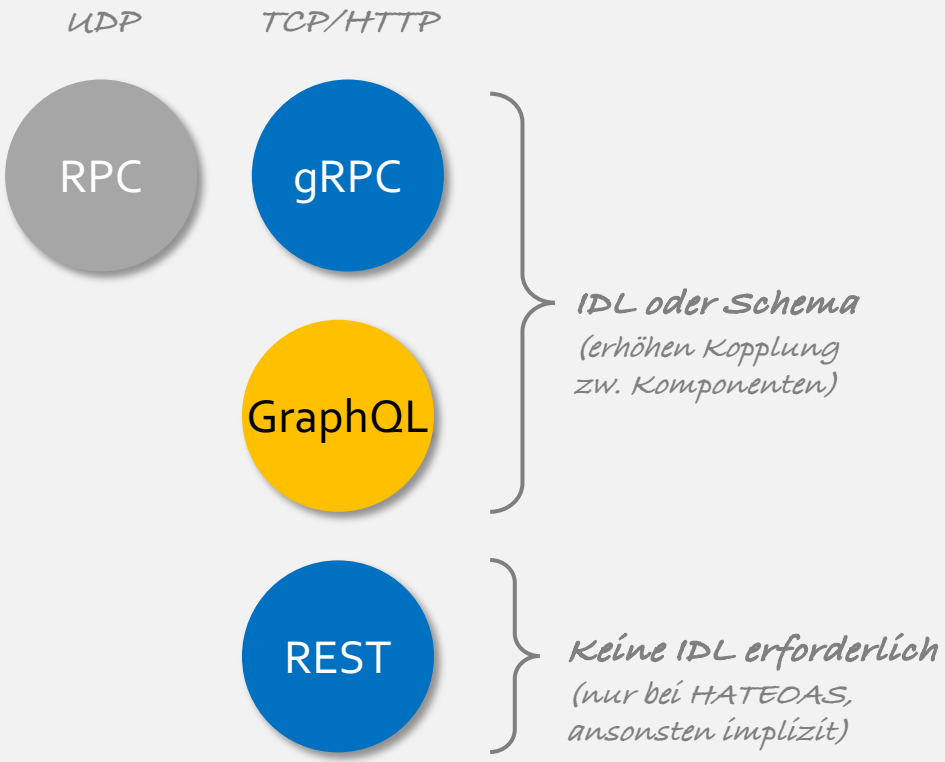


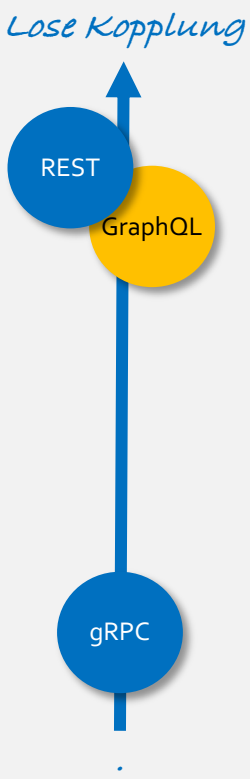
Das Richardson Maturity Model (RMM) ist ein von Leonard Richardson entwickelter Maßstab, der angibt, wie REST-konform ein Service implementiert wurde.

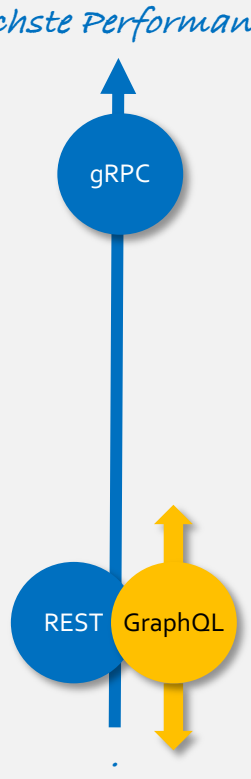
INTEGRATION

Request/Response-basierte Integration (synchron)









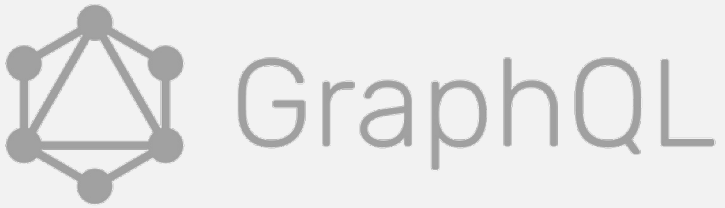
IDL: Interface definition language

Häufig muss man also zwischen Performance und loser Kopplung abwägen.

Achtung:
Es gilt auch die „ewige" Empfehlung erst Performanceprobleme anzugehen, wenn es einen aus dem Betrieb ersichtlichen Grund dazu gibt!

GRAPHQL

Abfragesprache für APIs, Einsatzzwecke



- GraphQL ist eine Abfragesprache für APIs, die ursprünglich von Facebook entwickelt wurde und inzwischen als Open-Source verfügbar ist.
- Im Gegensatz zum REST-Ansatz, bei dem jede Ressource in der API durch eine eigene URL repräsentiert wird, ermöglicht GraphQL, alle benötigten Daten auch abhängiger Ressourcen in einer einzigen Abfrage anzufordern.
- Während bei REST der Server entscheidet welche Daten gesendet werden, kann bei GraphQL der Client mittels einer Query festlegen, welche Daten übertragen werden.
- Dadurch kann der Overhead reduziert und weniger Daten mittels weniger Requests übertragen werden.

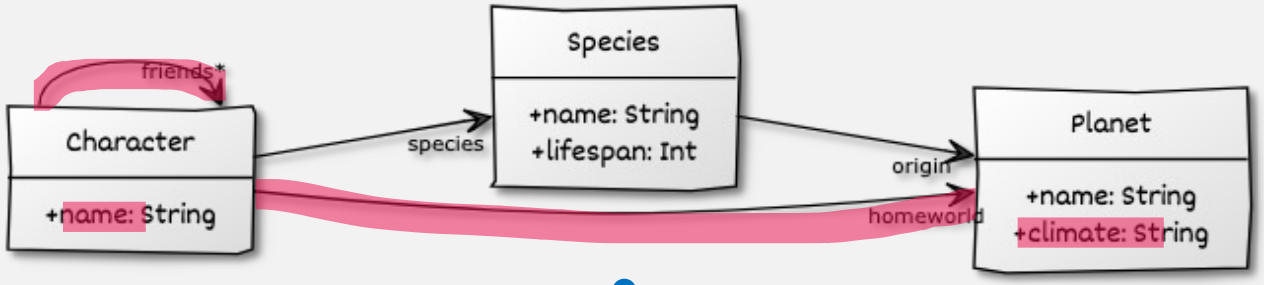
Einsatzzwecke

- **Mobile Apps**
  Bei langsamen und unzuverlässigen Netzwerken kann GraphQL hilfreich sein, da nur die benötigten Daten übertragen werden
- **Große und komplexe APIs**
  Bei APIs mit vielen Ressourcen und vielen Abfragen kann GraphQL genutzt werden, um eine einzige, flexible Abfragemöglichkeit zu bieten
- **Stateful Services**
  GraphQL wird überwiegend als Query-Layer für Stateful Services (Persistenz-Services) eingesetzt.

GRAPHQL

APIs mit abfragbarem Schema

- GraphQL-APIs sind in Form von **Typen und Feldern** organisiert, nicht in Form von Endpunkten pro Ressource
- Von **einem einzigen API-Endpunkt** hat man also vollen Zugriff auf alle in einem Schema definierten Typen und deren Felder sowie Verweise auf weitere Typen
- Mittels einer **Query** selektiert man ein Teil dieser Daten
- GraphQL verwendet Typen, um sicherzustellen, dass Clients nur nach Daten fragen, die für die Schnittstelle auch definiert sind
- Apps können diese Typen verwenden, um die Entwicklung von Parsing-Code zu vermeiden



Query

```
{
  hero {
    name
    friends {
      name
      homeWorld {
        name
        climate
      }
      species {
        name
        lifespan
        origin {
          name
        }
      }
    }
  }
}
```

```
type Query {
  hero: Character
}

type Character {
  name: String
  friends: [Character]
  homeWorld: Planet
  species: Species
}

type Planet {
  name: String
  climate: String
}

type Species {
  name: String
  lifespan: Int
  origin: Planet
}
```

Schema

GRAPHQL

Vor- und Nachteile (im Vergleich zu REST)

CLOUD NATIVE COMPUTING

Vorteile

- **Weniger Overhead**
  Anders als bei REST können mehrere Daten zu mehreren Ressourcen in einer einzigen Abfrage abgefragt werden
- **Flexibilität**
  Anders als bei REST kann da das Schema problemlos erweitert werden kann
- **Schnellere Entwicklung**
  Anders als bei REST müssen Entwickler nicht mehrere APIs oder mehrere Endpoints verwenden, um alle benötigten Daten abzurufen. Dies kann die Entwicklung beschleunigen.

Nachteile

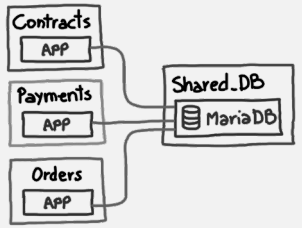
- **Lernkurve**
  Da es sich um eine Abfragesprache handelt, ist mit einer steileren Lernkurve zu rechnen (allerdings wird verlässlicher ein guter Maturity Level als bei REST erreicht)
- **Schwieriger zu Cachen**
  GraphQL ist aufgrund von Query-Heterogenität schlechter zu cachen
- **Stärkere Kopplung**
  Zusammenarbeiten zwischen Client und Server ist durch GraphQL-Schema-Definition klar definiert. Dies erhöht allerdings auch die Kopplung
- **Sicherheitsbedenken**
  Da GraphQL sehr flexibel ist, kann es schwierig sein, eine gute Absicherung gegen unerwünschte Zugriffe zu gewährleisten.

Hinweis: Performance-Steigerungen durch GraphQL sind möglich, insbesondere wenn mehrere Ressourcen abgefragt werden. Dies ist jedoch stark von der Query und dem Schema abhängig und sollte nicht pauschal erwartet werden.

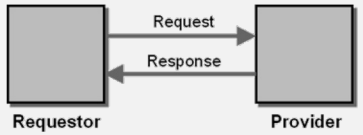
AUSBLICK

Integrationsstile

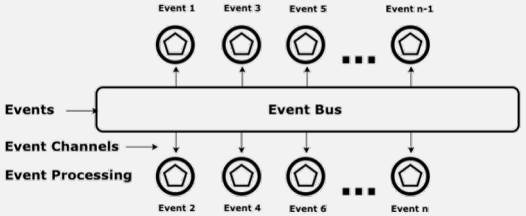
- Daten-basierte Integration
  - The shared database



- Request/Response-basierte Integration
  - Remote Procedure Calls
  - REST
  - GraphQL



- **Ereignis-basierte Integration**
  - Event Driven Architectures
  - Reactive Programming



KONTAKT

Disclaimer

Nane Kratzke

+49 451 300-5549

nane.kratzke@th-luebeck.de

kratzke.mylab.th-luebeck.de