Microservices

EIN FACHVORTRAG DER GRUPPE FOSSIL

Agenda

Was sind Microservices?

Vor- und Nachteile von Microservices?

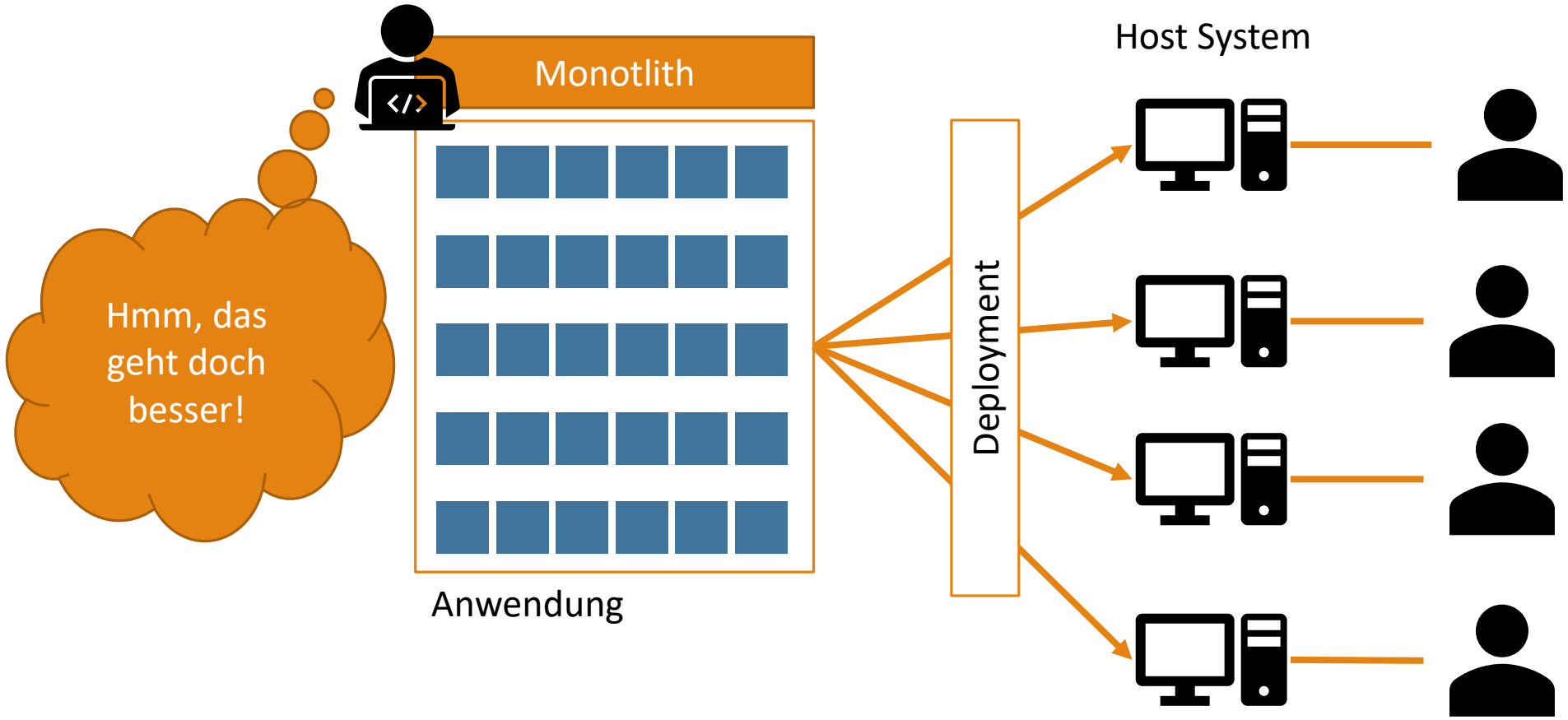
Microservices Tools

Dokumentation der API

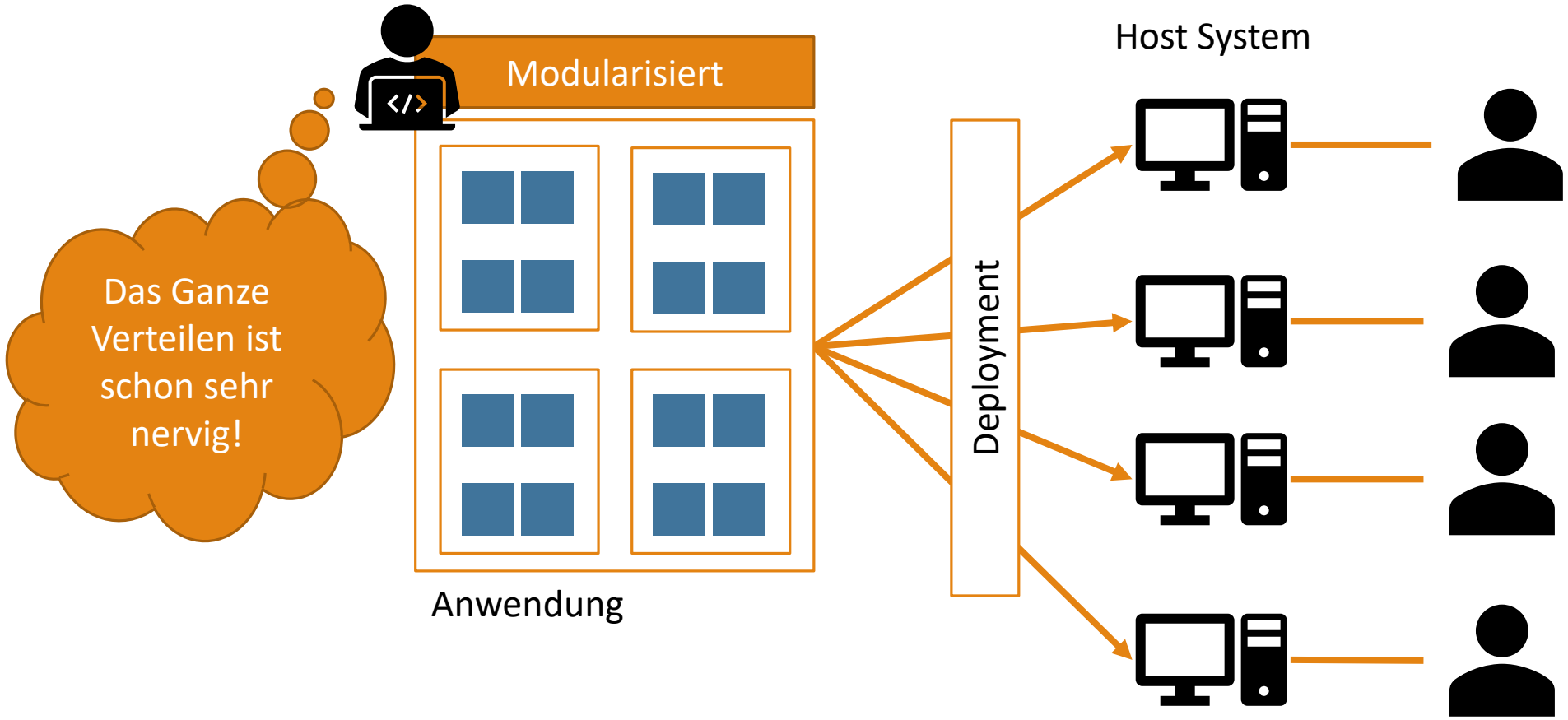
Beispiel Amazon AWS (Lambda)

Was sind Microservices?

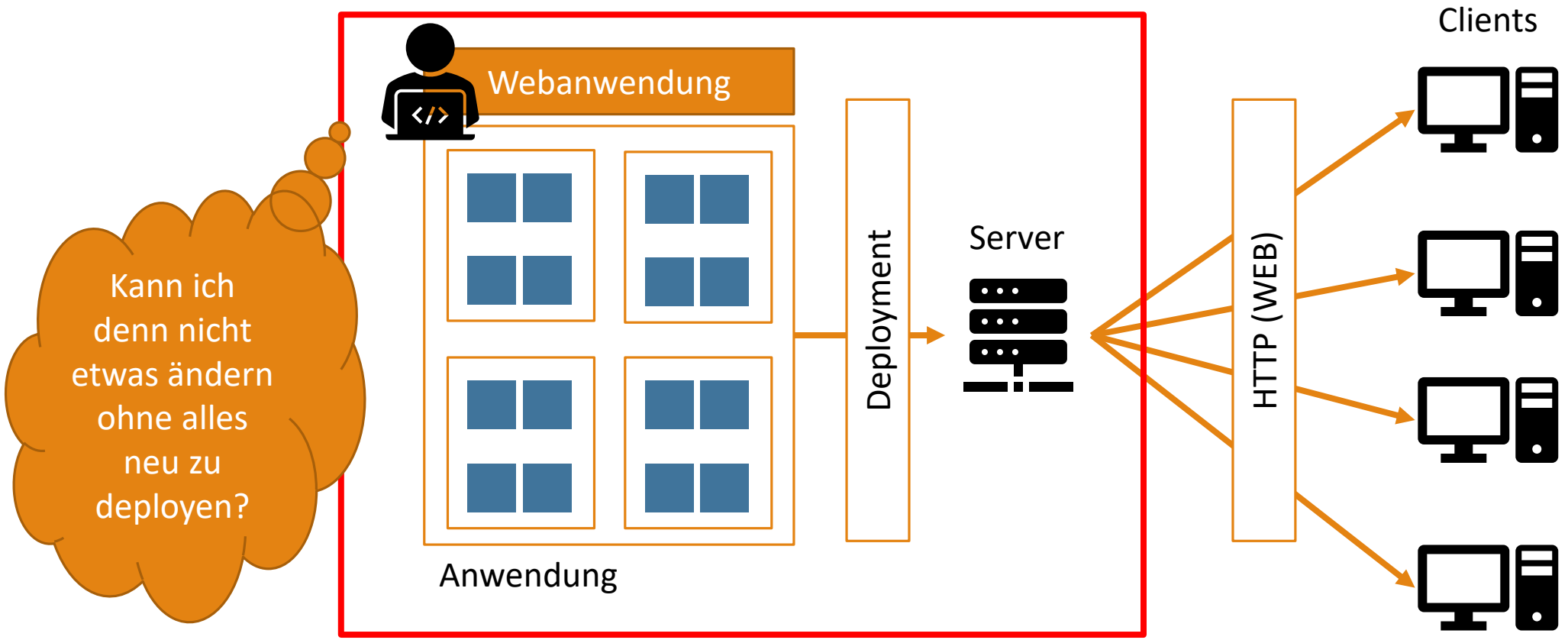
Klassische Anwendungsarchitekturen 1



Klassische Anwendungsarchitekturen 2

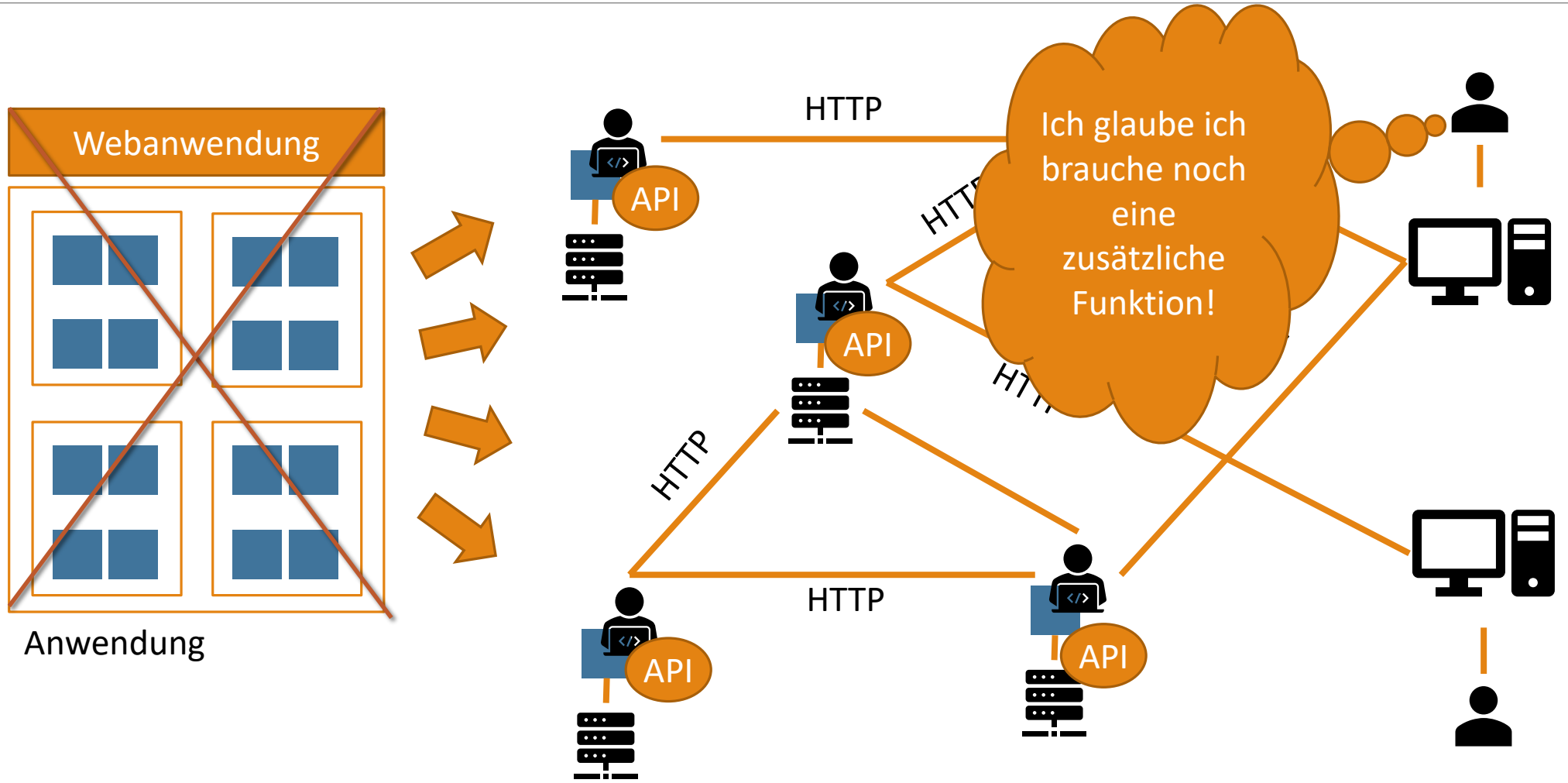


Architektur von Webanwendungen

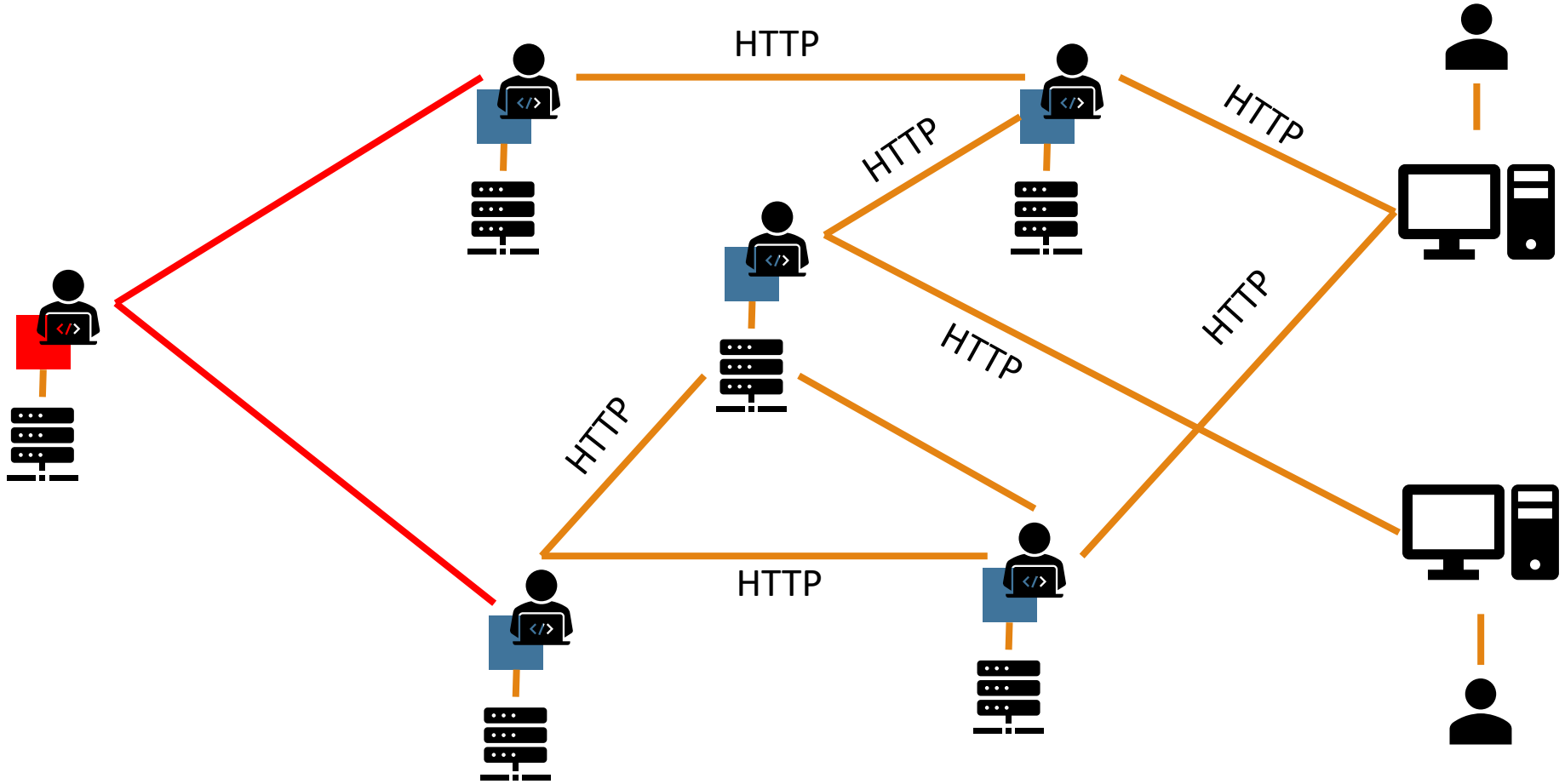


Die Anwendung muss als Ganzes auf den Server!

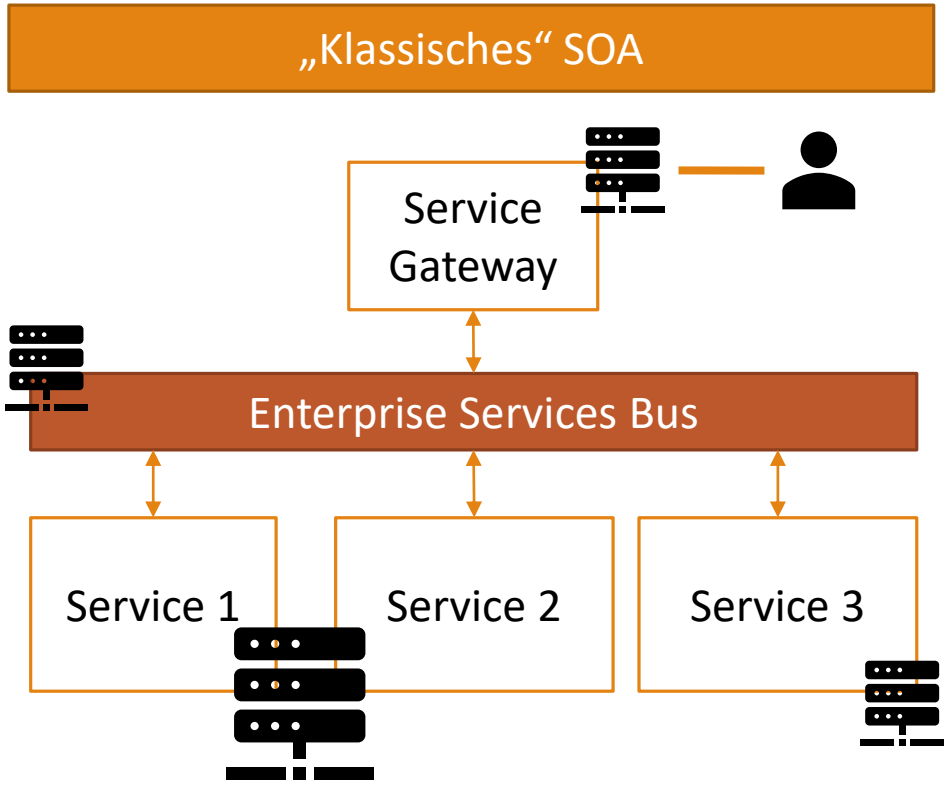
Mircroservices: Verteilte Anwendungen!



Neue Anforderung = neuer Service!



Abgrenzung zu SOA



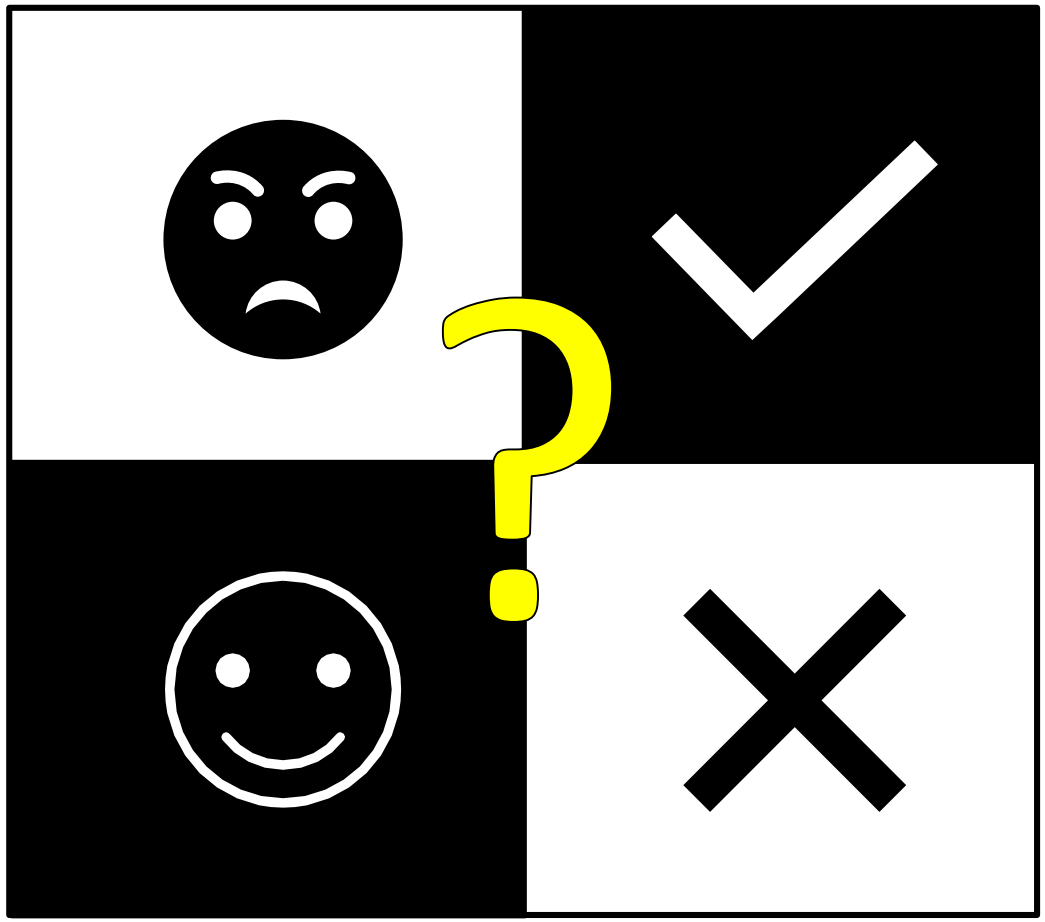
SOA „klassisch“ interpretiert	Microservices
Prozessorientiert	Anwendungs-orientiert
Ein Service Gateway (Schnittstelle extern)	Beliebig viele externe Schnittstellen
ESB benötigt	Kein ESB
Verschiedene Protokolle	RESTful HTTP API
Nutzung gemeinsamer Komponenten	Möglichst Isolierte Kompoenenten und Datenbestände

Microservices…

1. … sind ein Architekturstil für verteilte Anwendungen und als eine spezielle Interpretation von SOA zu verstehen
2. … sind spezialisierte Services, die für sich alleine stehen und über eine RESTful http Schnittstelle ihre Dienste anbieten
3. … sind Services, die über eine Dokumentation ihrer API verfügen
4. … halten wenn sinnvoll die benötigten Daten selbst vor

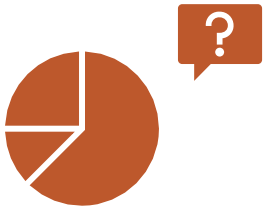
Vor- und Nachteile von Microservices?

Vor- und Nachteile: Nur schwarz-weiß?



Geeignet, wenn…

→ hohe Flexibilität gewünscht ist

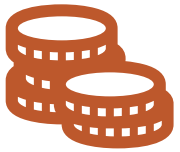


→ Viele Unbekannte existieren

→ Unabhängigkeit einzelner Teams gewünscht ist

Ungeeignet, wenn…

→ das gewünschte Ziel eigentlich doch ein Monolith ist, getarnt als Microservices



→ man kosteneffizient arbeiten will

Vorteile

1. Flexible Bereitstellung von einzelnen Funktionen
    1. autonome Implementierung durch unterschiedliche Teams
    2. Bereitstellung von unterschiedlichen Servers möglich
    3. Unterschiedliche Sprachen möglich
    4. schnellere Updates möglich
2. Kommunikation der einzelnen Funktionen durch HTTP oder RESTful APIs, daher egal, welche Sprache genutzt wird
3. Unabhängigkeit in der Skalierung
4. Testing einzelner Funktionen einfacher (anstatt ALLES zu testen, kann man einen Service testen)

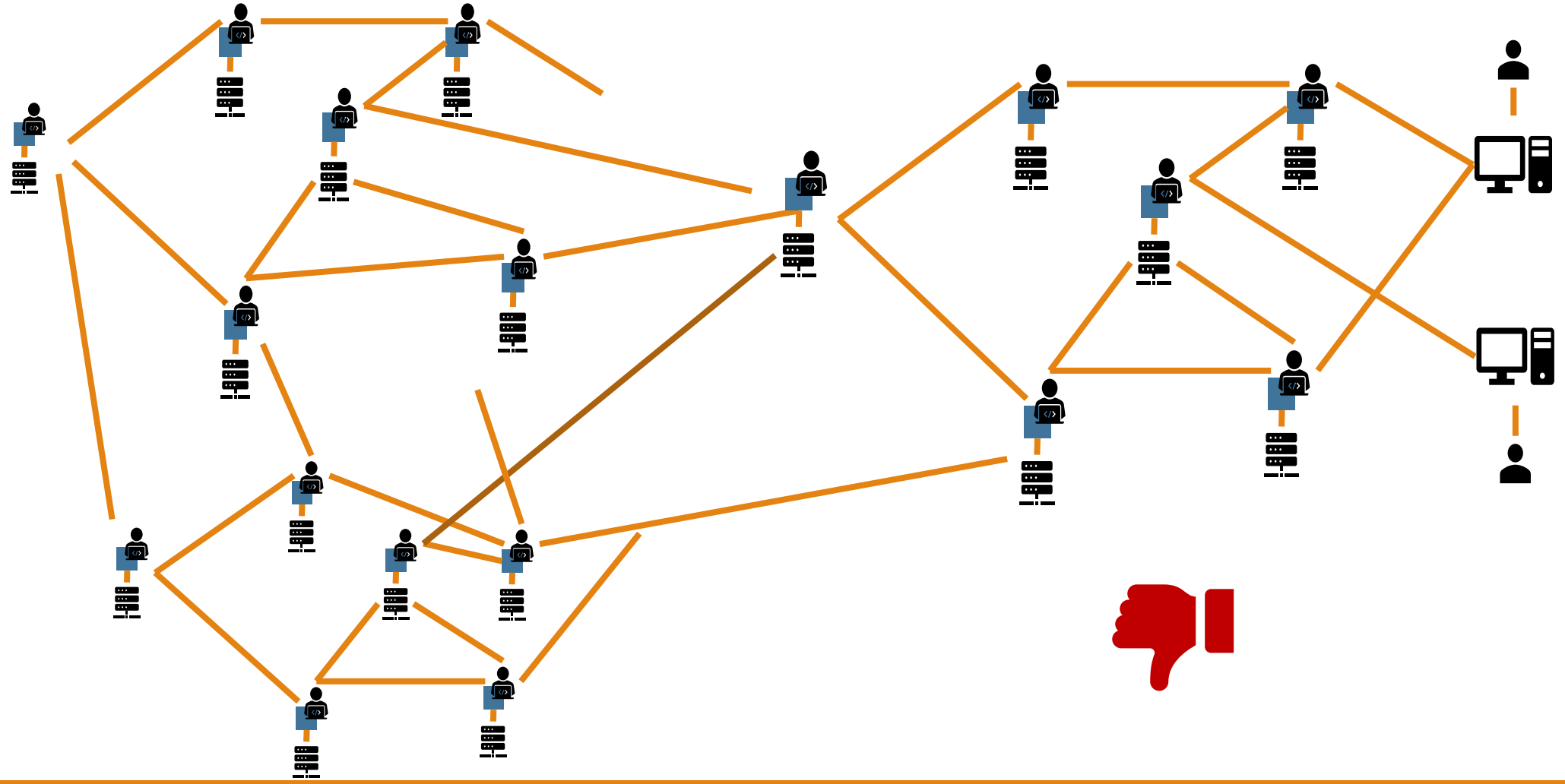
Vorteile

1. Austausch einzelner Funktionen problemlos möglich, da kein anderer Service angefasst wird
2. Sensible Daten können gezielt verstärkt geschützt werden
3. Einbauen/Testen neuer Technologien einfacher

Nachteile bzw. Herausforderungen

1. Service A will mit Service B kommunizieren
   1. Netzwerkverzögerungen
   2. Überlastung
   3. Scheitern der Kommunikation
   4. Service B könnte fehlerhaft sein
2. Verteiltes System → erhöhte Komplexität der gesamten Struktur
   1. Zu vermeiden: eine Kette (oder gar Kreis!) aus Service-calls
3. Kostspielig: viele kleine Services auf vielen Geräten anstatt ein großer Service auf einem Gerät
4. Debuggen fällt schwer
   1. WO liegt genau das Problem?
   2. Kein Stack zum Prüfen
   3. Asynchrone Struktur

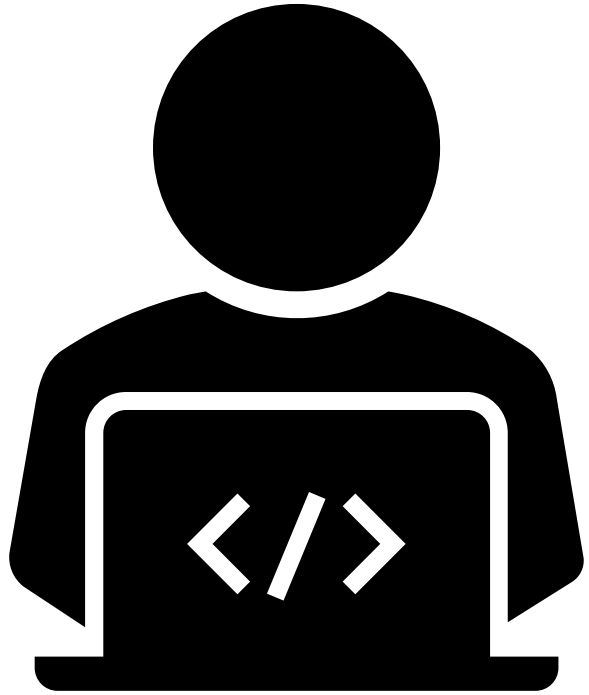
Spezielle Herausforderung: Spaghetti“code“



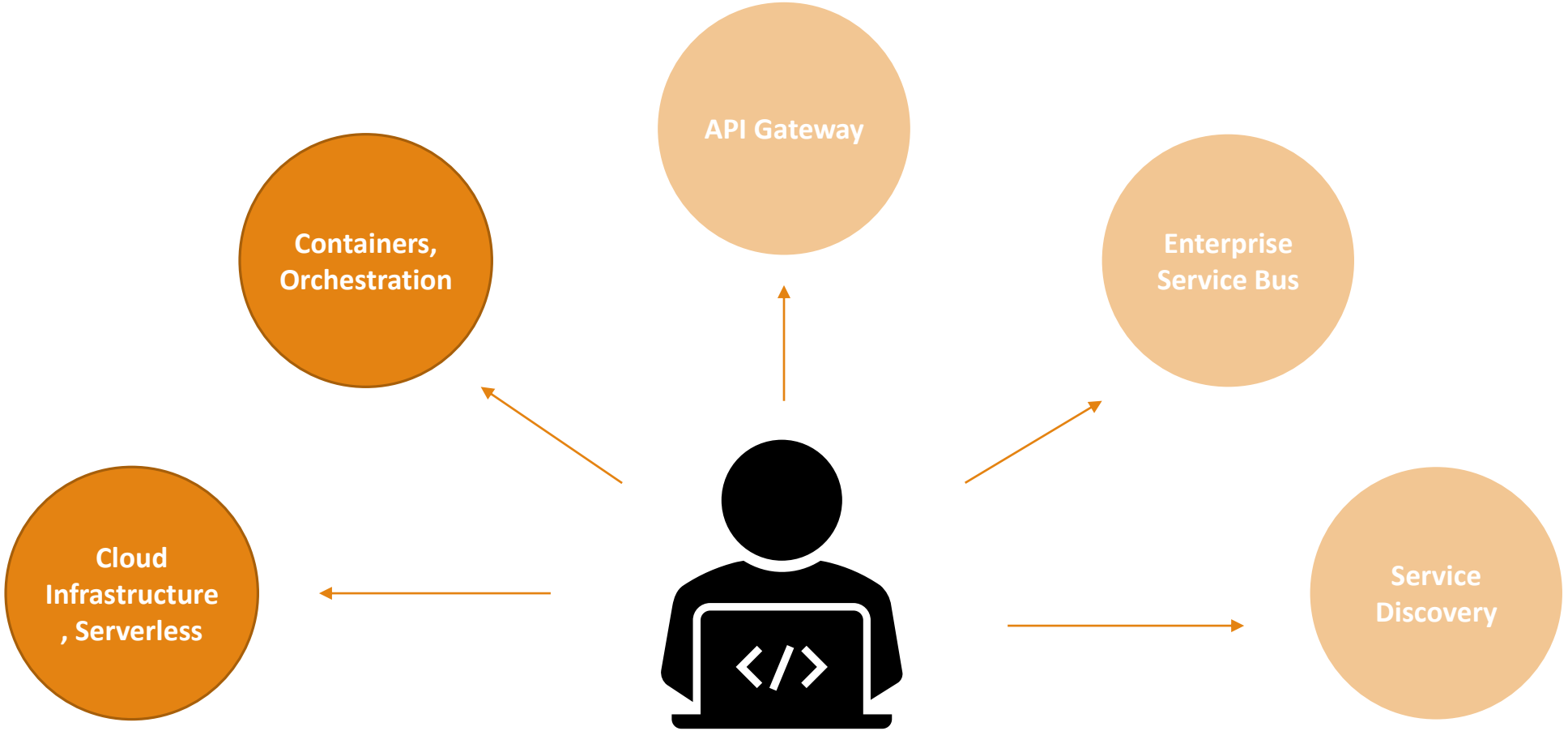
Take away message

1. Microservices sind nicht das Non-plus ultra der Architekturen
2. Man muss wissen, was man mit seinem Produkt erreichen möchte
3. Microservices können viele Vorteile bieten, müssen aber diszipliniert gewartet und strukturiert aufgebaut sein
4. Bei schlechter Wartung und schlechtem Aufbau wird die Microservice Architektur zur Falle

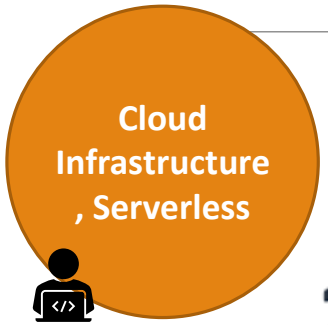
Microservices Tools



Umgebung und Verwaltung für Microservices



Cloud Plattformen



aws

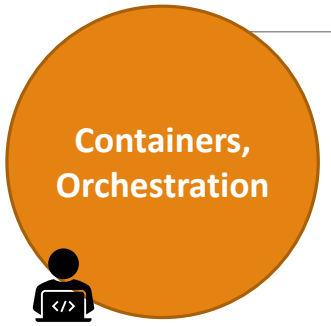
Azure

Google Cloud Platform

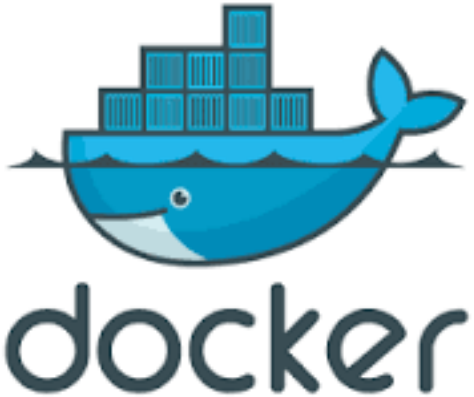
openstack

rackspace
the open cloud company

Container Plattform und Orchestration

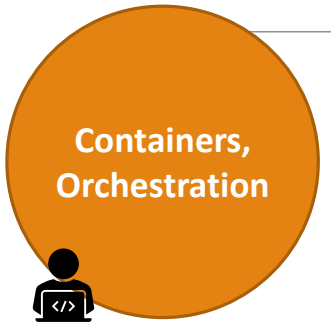


Containers

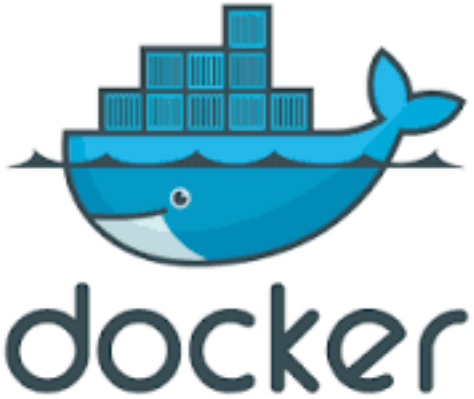


Orchestration

Container Plattform und Orchestration



Containers

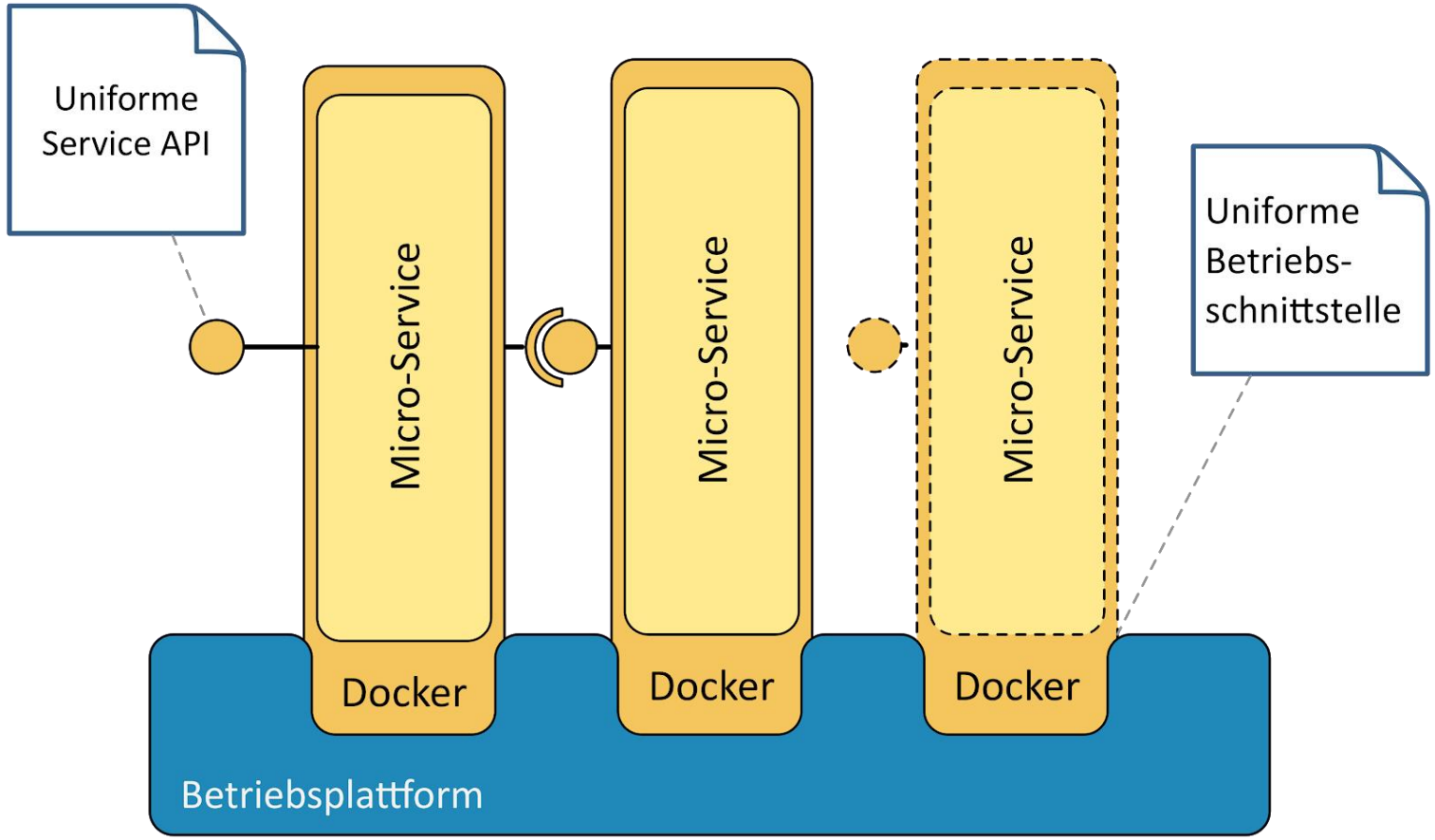


Orchestration

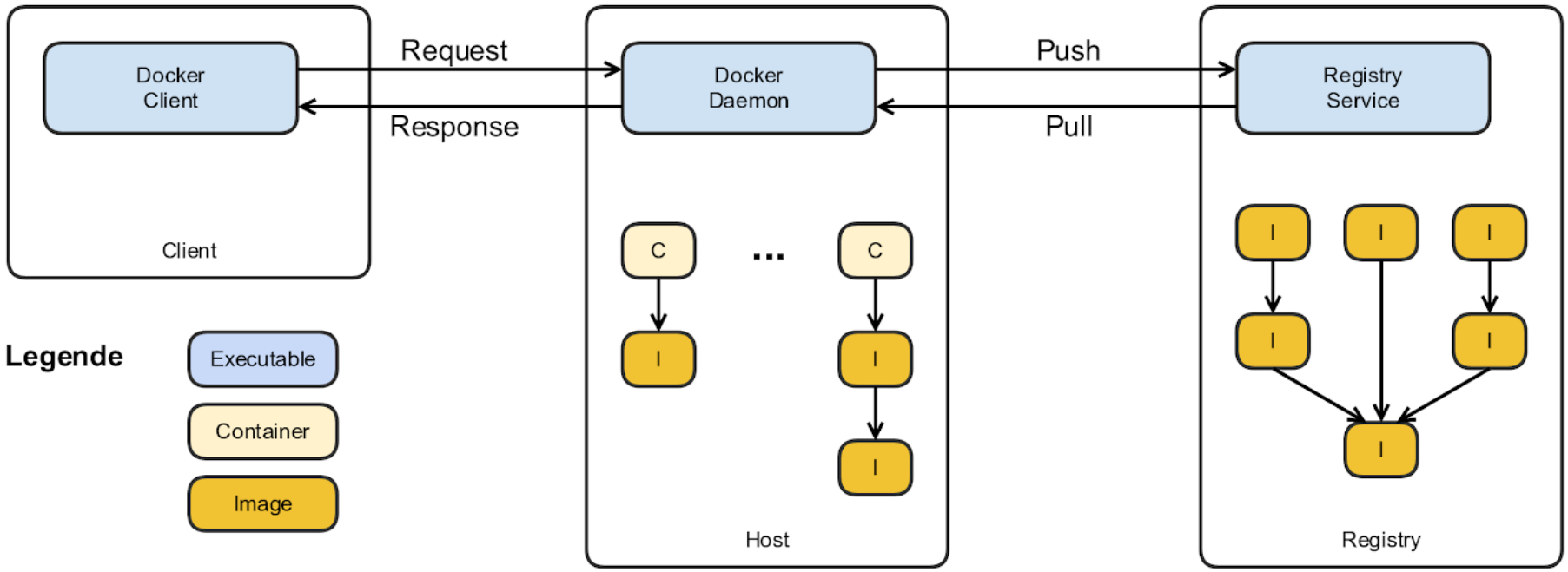
Vorteile von Containern

1. Abstraktion vom unterliegenden Betriebssystem und Hardware
2. Microservice sind unabhängige Prozesse auf minimalen Systemen
3. keine Kernel Überladung
4. bringen Abhängigkeiten selbst mit
5. transparente Abhängigkeiten

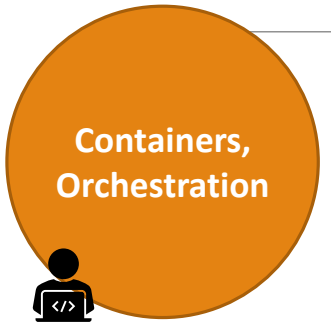
Wie funktionieren Service-Container?



Wie funktionieren Service-Container?



Container Plattform und Orchestration

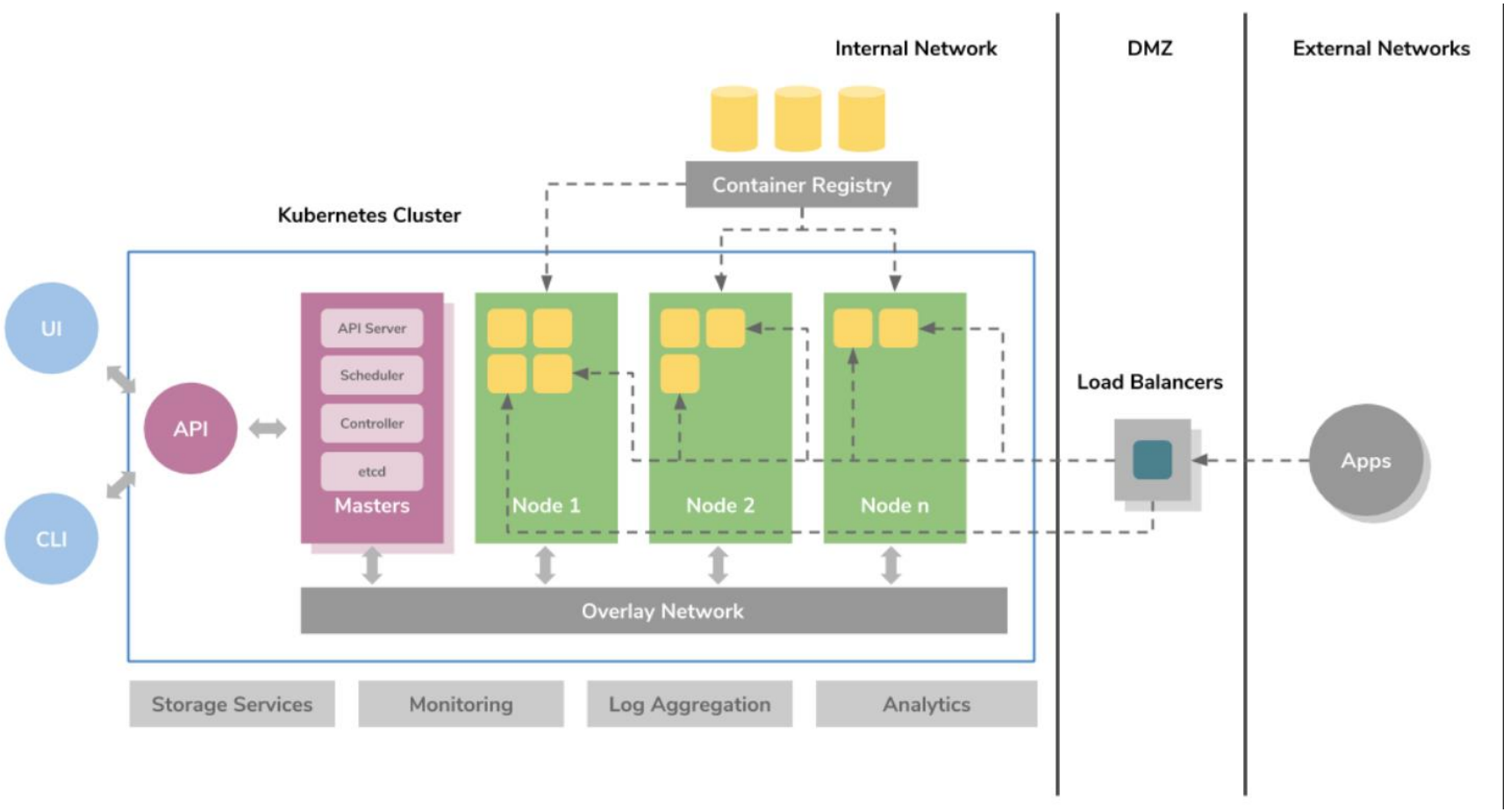


Containers

Orchestration

kubernetes

Wie funktionieren Service-Orchestration?



Dokumentation der API

Warum sollte man Dokumentieren?

Ist die erste Anlaufstelle bei Problemen und/oder Fragen zur API Benutzung

- Übersicht über Funktionen
  - Erklärung der Syntax
  - wofür zu benutzen (Beispiele)
  - wie genau zu benutzen

Was sollte sie können?

- einfach Navigierbar und Verständlich sein
- automatisch Generierbar sein
- einfach Modifizierbar sein
- integrierte Tests enthalten
- Beispiele enthalten

Womit kann man das machen?

Beipiel Tools:

Swagger

Blueprint

Swagger

- Großes Toolset (Design, Entwicklung und Testing)
- Dokumentation als JSON oder YAML
- Erweiterbar durch Module (Generator)
- Open Source

API Blueprint

- Fokussiert auf Dokumentation
- Markdown basierend
- Github Integration (Kostenpflichtig)
- Open Source

Zwei Beispiele

Authentication Service	
Client Management	
Function	Create a new API client
Method	POST
URI	/clients.{format}

Request Body (raw)	{
	"application": "Virtual Car Collector",
	"secret": "pbZCmrFSBqkYtMh"
	}

Status	201 Created

Response Body	New client object
	
	{
	"_links": {
	"self": {
	"href": "http://virtual-vehicles.com:8587/clients/55482c1bc830337da8c6bd81"
	},
	"up": {
	"href": "http://virtual-vehicles.com:8587/clients"
	}
	},
	"application": "Virtual Car Collector",
	"secret": "pbZCmrFSBqkYtMh",
	"apiKey": "PT2lOldZG6hSEDddushzSN1G",
	"id": "55482c1bc830337da8c6bd81",
	"createdAt": "2015-05-05T02:34:03.400Z",
	"updatedAt": "2015-05-05T02:34:03.400Z"
	}

MongoDB Document View (BSON)	{
	"_id" : ObjectId("55482c1bc830337da8c6bd81"),
	"className" : "com.example.authentication.objectid.Client",
	"application" : "Virtual Car Collector",
	"secret" : "pbZCmrFSBqkYtMh",
	"apiKey" : "PT2lOldZG6hSEDddushzSN1G",
	"createdAt" : ISODate("2015-05-05T02:34:03.400Z"),
	"updatedAt" : ISODate("2015-05-05T02:34:03.400Z")
	}

Getting a book

To retrieve a book you need to know its ISBN, which should be provided as part of the URI.

Example of a request:

```
GET /api/book/isbn-t1 HTTP/1.1
Accept: application/json
```

```
$ curl 'http://localhost:8080/api/book/isbn-t1' -i -H 'Accept: application/json'
```

Example of a success response:

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

{
  "isbn" : "isbn-t1",
  "title" : "test title",
  "author" : "test author",
  "_links" : {
    "self" : {
      "href" : "http://localhost:8080/api/book/isbn-t1"
    }
  }
}
```

Definition of the returned properties:

Path	Type	Description
isbn	String	The Book's ISBN
title	String	The Book's title
author	String	The Book's author

Beispiel Amazon AWS (Lambda)

Aufsetzen eines Service

Im Terminal:

```
serverless create --template aws-nodejs
```

```
serverless deploy
```

In der AWS Console:

API Gateway vorschalten

Weitere Templatebeispiele:

- aws-alexa-typescript
- aws-python
- aws-python3
- aws-ruby
- aws-java-maven
- aws-csharp
- aws-go
- aws-go-mod

```
//Ueberpruefung auf den Namen
module.exports.hello = async (event) => {
  const name = _.get(event, 'queryStringParameters.name');
  if (name === undefined || _.isEmpty(name)) {
    return {
      statusCode: 400,
      body: JSON.stringify({
        message: 'Name parameter not valid'
      })
    };
  }

  //Ausgabe je nach Ergebins
  const message = isKnown ? `Welcome back, ${name}` : `Hello,

  return {
    statusCode: 200,
    body: JSON.stringify({
      message
    })
  };
};
```

Implementierung des Service “greet”

```python
# Ueberprueft ob der Name schon bekannt ist
def known(event, context):
    if "name" in event:
        name = event["name"]
        known = False

        names = ["Kim", "Mara", "Basti"]

        try:
            names.index(name)
            known = True
        except:
            known = False

        return {
            "statusCode": 200,
            "body": json.dumps({
                "known": known
            })
        }
    else:
        return {
            "statusCode": 400,
            "body": json.dumps({
                "message": "Name parameter not valid"
            })
        }
```

Implementierung des Service “known”

```
//Aufruf des anderen Service (known)
function getKnown(name) {
  return new Promise((resolve, reject) => {
    lambda.invoke({
      FunctionName: 'known-dev-known',
      Payload: JSON.stringify({
        name
      })
    }, (err, data) => {
      if (err) {
        reject(err);
      }
      resolve(JSON.parse(data.Payload));
    })
  });
}

  //Auslesen der Antwort des known-service
  const isKnownResponse = await getKnown(name);
  const isKnown = JSON.parse(isKnownResponse.body).known;

  //Ausgabe je nach Ergebins
  const message = isKnown ? `Welcome back, ${name}` : `Hello, ${name}`;

  return {
    statusCode: 200,
    body: JSON.stringify({
      message
    })
  };
};
```

Aufruf des Service “known” durch “greet”