Die Implementierung

Phase IV

Implementierung

> Convert a design into a complete information system includes acquiring and installing system environment, creating and testing databases, preparing test case procedures, preparing test files, coding, compiling, refining programs, performing test readiness review and procurement activities.

- Wir schauen uns an
  - Implementierungsthemen innerhalb der Komponente
  - Implementierungsthemen außerhalb der Komponente

Exkurs: Service vs. Component

- **Service**: A unit of functionality that is designed to be deployed independently and reused by multiple systems
- **Component**: A unit of functionality that is designed to serve as a part of a system or application.
- *"A service is similar to a component in that it's used by foreign applications. The main difference is that I expect a **component to be used locally** (think jar file, assembly, dll, or a source import). **A service will be used remotely** through some remote interface, either synchronous or asynchronous (eg web service, messaging system, RPC, or socket.)"* Martin Fowler
- Wir halten also fest:
  - Services bestehen aus Komponenten
  - Komponenten können Services aufrufen
- Die Prinzipien bleiben immer gleich Interface, Port, Connector

Wie kann man Komponenten in .NET abbilden?

Komponenten in .NET

- Eine Komponente kann mehrere Assemblies haben (zur Erinnerung: Modul → Design-Time vs. Komponenten → Runtime)
- Ein Prozess kann mehrere AppDomains haben
  - Wichtiges Konzept um gewisse Anforderungen zu realisieren → siehe Folgefolien
- Mehrere Komponenten sind oft Teil einer Applikation (e.g. in Form eines Services, welches e.g. Wetterdaten anbietet)
  - Komponenten wiederrum können andere Services aufrufen

AppDomains

- *„An AppDomain is a logical container for a set of assemblies.”*
- Es können mehrere AppDomains in einem Prozess laufen
- Vorteile:
  - Isolation Boundary / Security (Stichwort: Unit-Of-Mitigation)
  - Reliability (u.a. Themen aus Fault-Tolerance → Unhandeled Exceptions können nicht gefangen werden)
  - Unloading Assemblies
- Warum AppDomains?
  - Wenn mehrere Prozesse → Kommunikation zwischen Prozessen schwergewichtig (Cross-Process-Calls)
  - Prozess-Context-Switching im Betriebssystem teuer

AppDomains (siehe Übung)

```
private static AppDomain CreateBillingDomain()
{
    var domainInfo = new AppDomainSetup
    {
        // The application base directory is where the assembly manager begins probing for assemblies.
        ApplicationBase = Environment.CurrentDirectory
    };

    // Defines the set of information that constitutes input to security policy decisions.
    var evidence = AppDomain.CurrentDomain.Evidence;

    return AppDomain.CreateDomain("BillingDomain", evidence, domainInfo);
}

private static IBillingAddIn CreateBillingInstance(AppDomain billingDomain)
{
    return (IBillingAddIn)billingDomain.CreateInstanceAndUnwrap("Billing, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null", "Billing.BillingAddIn");
}
```

AppDomains

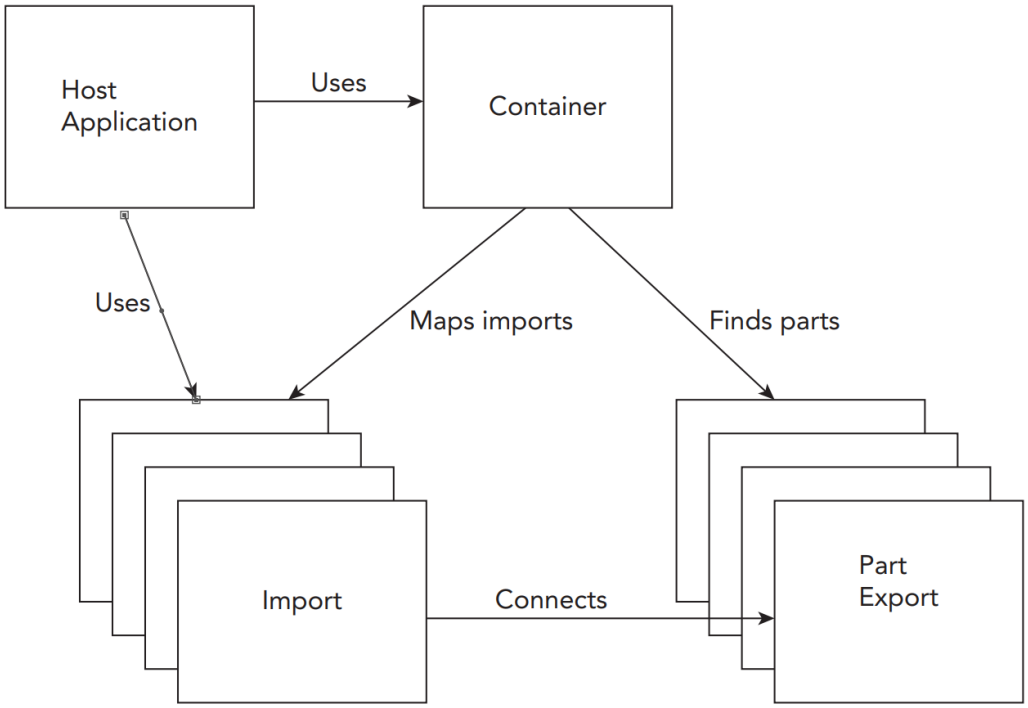
```
while (true)
{
    try
    {
        var result = instance.GetAverageSalaryByPersonId(new PersonId(123));

        Console.WriteLine($"Salary is {result.Amount}");
    }
    catch (ApplicationException exception)
    {
        Console.WriteLine($"Exception: {exception.Message}");

        AppDomain.Unload(billingAddIn);

        billingAddIn = CreateBillingDomain();
        instance = CreateBillingInstance(billingAddIn);
    }
```

Managed Extensibility Framework (MEF)

- Hilft laufzeit-erweiterbare Komponenten (Extensibility – siehe Kapitel Design) zu bauen → wir sprechen von „Application-Composition“
- Wir wollen **nicht** jedes Mal in Visual-Studio den Code neu kompilieren müssen, wenn wir die Komponente erweitern wollen
- Über Imports sagen wir: „Suche mir zur Laufzeit eine Implementierung des Interfaces `ICalculator` und lade das Assembly. Stelle mir anschließend eine Instanz zur Verfügung“.

Managed Extensibility Framework (MEF)

- https://cnistorage.azureedge.net/procsharp/procsharp7_b01.pdf



MEF

```
[Import(typeof(IPlugin1))]
public IPlugin1 Plugin1;

[Import(typeof(IPlugin2))]
public IPlugin2 Plugin2;
```

```
var pluginDirectory = System.IO.Path.GetDirectoryName(System.Reflection.Assembly.GetExecutingAssembly().Location);

Console.WriteLine($"Searching for plugins ins '{pluginDirectory}' ...");

var catalog = new DirectoryCatalog(pluginDirectory, "*.dll");

// Create the CompositionContainer with the parts in the catalog.
container = new CompositionContainer(catalog);

DebugMetadata();

// Fill the imports of this object.
try
{
    container.ComposeParts(this);
}
catch (CompositionException compositionException)
{
    Console.WriteLine(compositionException.ToString());
}
```

FH University of Applied Sciences TECHNIKUM WIEN

Wie kann man Ports in .NET abbilden?

Ports

- Klassen und Interfaces
- Wichtig:
  - Interface sollten immer selbstsprechend sein
  - Tests as documentation
  - Wenn nicht eindeutig, Dokumentation prosa
- Ports können auch statefull sein
  - https://github.com/quozd/awesome-dotnet#state-machines
  - Gewisse Interface-Methoden nur in einem gewissen State erlauben

Was sind bekannte Datenformate für die Kommunikation über Connectors?

Untergliederung

- Wir schauen uns die gängigsten Formate an
- semistrukturierten Datenformate (auch der Name des Elements wird mitgeschickt → Strukturinformation wird mitgeschickt)
  - JSON, XML
- Nicht semistrukturierten Datenformate (im Vorfeld wird das Schema „ausverhandelt“)
  - ProtoBuf, AVRO

XML (Extensible Markup Language)

- Beispiel:

```
<employees>
  <employee>
    <firstName>John</firstName> <lastName>Doe</lastName>
  </employee>
  <employee>
    <firstName>Anna</firstName> <lastName>Smith</lastName>
  </employee>
  <employee>
    <firstName>Peter</firstName> <lastName>Jones</lastName>
  </employee>
</employees>
```

Vorteile von XML

- XML-Schema: breite Vielfalt an Implementierungen, sehr ausgereift
- Xpath / Xquery ist „offizieller“ (W3-Konsortium)
- Namspace support: siehe https://de.wikipedia.org/wiki/Namensraum_(XML)
- Kommentare werden unterstützt

JSON

- Beispiel

```
{"employees":[
  { "firstName":"John", "lastName":"Doe" },
  { "firstName":"Anna", "lastName":"Smith" },
  { "firstName":"Peter", "lastName":"Jones" }
]}
```

Unterschiede JSON zu XML

- Nutze keinen End-Tag
- Ist kürzer / kompakter
- Ist dadurch schneller zu schreiben und zu lesen
- Hat das Konzept von Arrays

ProtoBuf (Avro ähnlich)

- Man definiert ein „Schema“ im Vorfeld (.proto)

```
message Person {
  string name = 1;
  int32 id = 2;  // Unique ID number for this person.
  string email = 3;

  enum PhoneType {
    MOBILE = 0;
    HOME = 1;
    WORK = 2;
  }

  message PhoneNumber {
    string number = 1;
    PhoneType type = 2;
  }

  repeated PhoneNumber phones = 4;

  google.protobuf.Timestamp last_updated = 5;
}

// Our address book file is just one of these.
message AddressBook {
  repeated Person people = 1;
}
```

ProtoBuf

- Es wird dann per Compiler für die Zielsprache eine „Parser-Klasse“ generiert
- Alternativ: auch per Reflektion https://github.com/protobuf-net/protobuf-net
- Extrem performant und kompakt (viele Zusatzoptimierungen)
- Im Vergleich zu JSON und XML: Feldname wird nicht mitgeschickt
  - Viele weitere Optimierungen: https://developers.google.com/protocol-buffers/docs/encoding

FH University of Applied Sciences TECHNIKUM WIEN

Wie kann man Connectors in .NET abbilden?

Welchen Connector brauch ich überhaupt?

- Thema im Design
- Nicht blindlinks immer auf REST-over-JSON-HTTP setzen
- Nicht-Funktionale Anforderungen treiben die Entscheidung stark … → JSON + HTTP 1.1 nicht das schnellste Werkzeug …
  - Entkopplung
  - Verfügbarkeit
  - …
  - was brauch ich? Ich brauch auch keine Hilti um ein Bild auf einer Trockenbauwand aufzuhängen

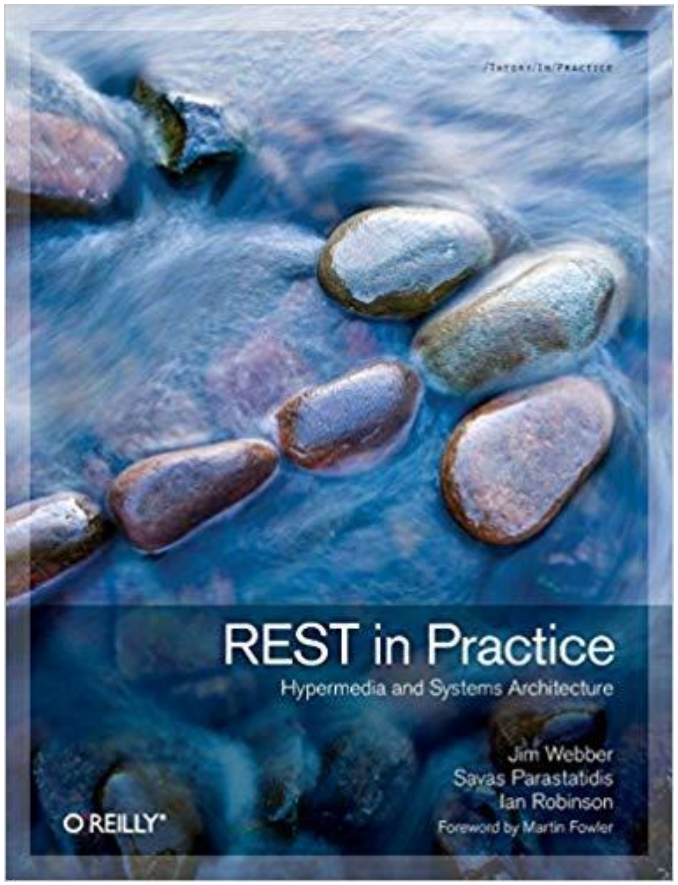
REST

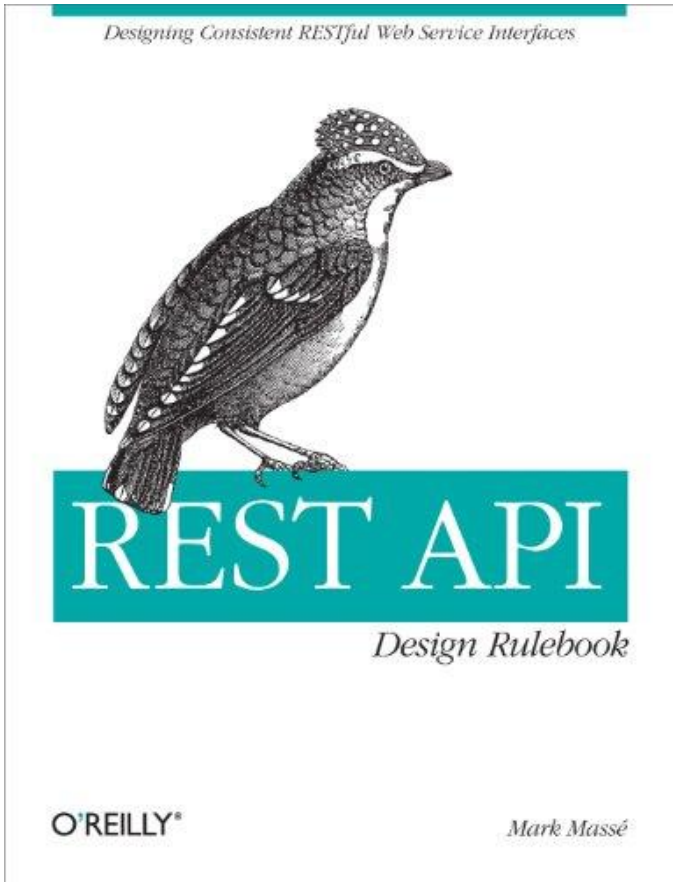
- Implementierung per ASP.NET WebAPI
- Hat inzwischen eigenes Dependency injection Framework (reicht für 99% der Anwendungsfälle): https://docs.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection?view=aspnetcore-3.0
- Auf der Consumer-Seite:
  - https://github.com/restsharp/RestSharp
  - https://github.com/canton7/RestEase

REST und die HTTP Verben

- Schnittstellen nutzen dabei HTTP Verben – die wichtigsten
  - `GET`: Gibt mir eine Ressource e.g. http://localhost:80/api/persons/12
  - `PUT`: Mach ein Update einer Person e.g. http://localhost:80/api/persons/12
  - `POST`: Erzeuge eine Person e.g.http://localhost:80/api/persons
  - `DELETE`: Lösche eine Person e.g. http://localhost:80/api/persons/12
  - `PATCH`: Teile der Ressource ändern e.g. http://localhost:80/api/persons/12
    - Siehe JSON-Patch: RFC6902

REST: Literatur



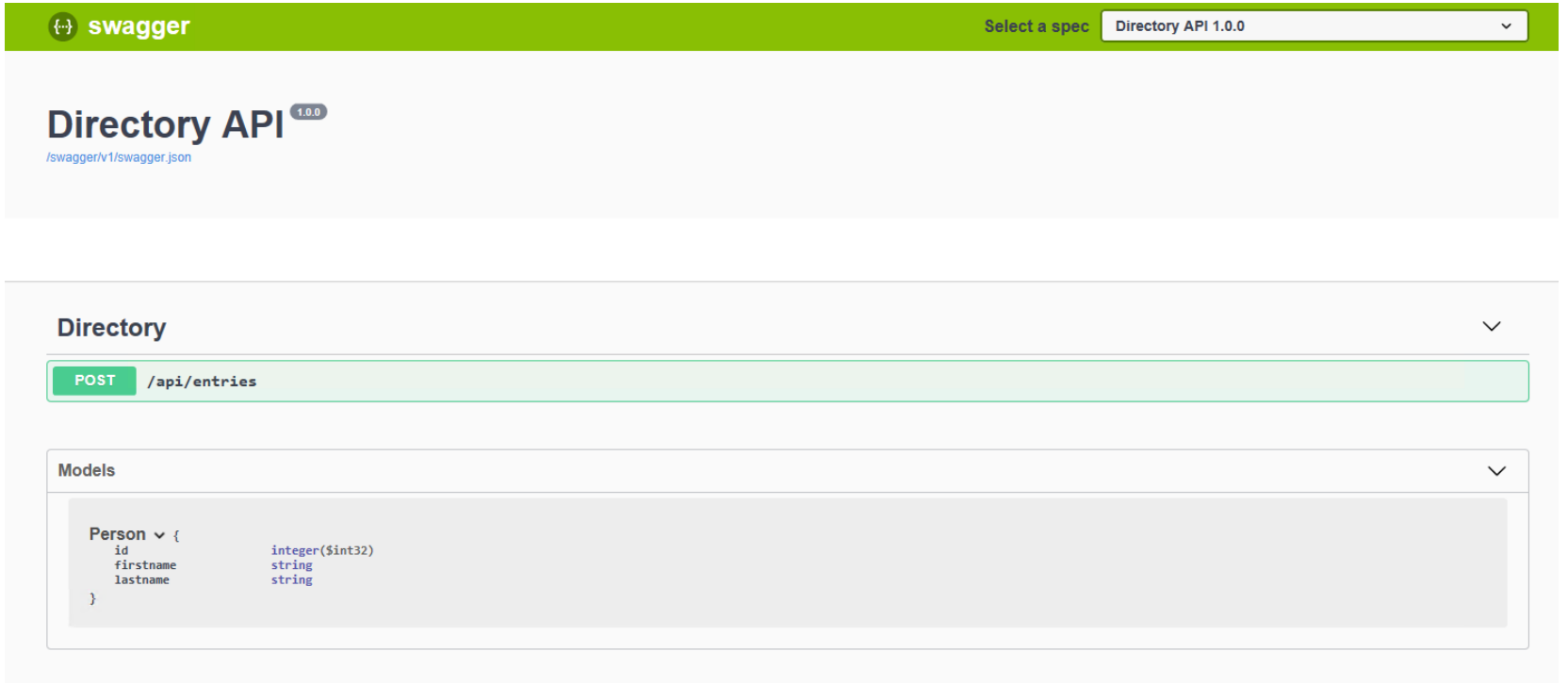


REST

```
[HttpGet]
public Task<IEnumerable<Person>> Get()
{
    return this.persistence.GetAllAsync();
}

[HttpPost]
public async Task<ActionResult<Person>> Create(Person person)
{
    var idOfCreatedPerson = await this.persistence.SaveAsync(person);

    var uriToCreatedRessource = this.BuildResponseUri(new Uri($"/api/entries/{idOfCreatedPerson}", UriKind.Relative));

    return Created(uriToCreatedRessource, person);
}
```

```
[HttpDelete("{id}")]
public Task Delete(int id)
{
    return this.persistence.DeleteAsync(id);
}
```

REST: Schnittstellendokumentation



GRPC

- Nutzt RPC-Style → meist Client-Server Modell (Peer-2-Peer natürlich auch möglich)
  - Client wartet auf Antwort. Nach Empfang der Nachricht kann der Client seine Verarbeitung fortführen.
  - Beim Einsatz von RPC können durch **Kommunikationsfehler** unterschiedliche Fehlerkonstellationen auftreten, die beachtet und bearbeitet werden müssen (e.g. Slow/No-Responses, Timeout, …)
- Definitiv Alternative zu JSON over HTTP
  - Performanter
  - Sprechende Server & Client Implementierungen (durch Codegenerierung)
  - Streaming von Responses
  - Etwas schwerer zu debuggen als JSON over HTTP (da nicht selbstbeschreibend → proto File wird benötigt)
  - Layer 7 Load-Balancer / Firewalls schwieriger / aufwendiger
- Dokumentation sehr gut: https://grpc.io/

GRPC: Iterface-Definition-Language (IDL)

- *.proto Files

```
// The greeting service definition.
service Greeter {
  // Sends a greeting
  rpc SayHello (HelloRequest) returns (HelloReply) {}
  // Sends another greeting
  rpc SayHelloAgain (HelloRequest) returns (HelloReply) {}
}

// The request message containing the user's name.
message HelloRequest {
  string name = 1;
}

// The response message containing the greetings
message HelloReply {
  string message = 1;
}
```

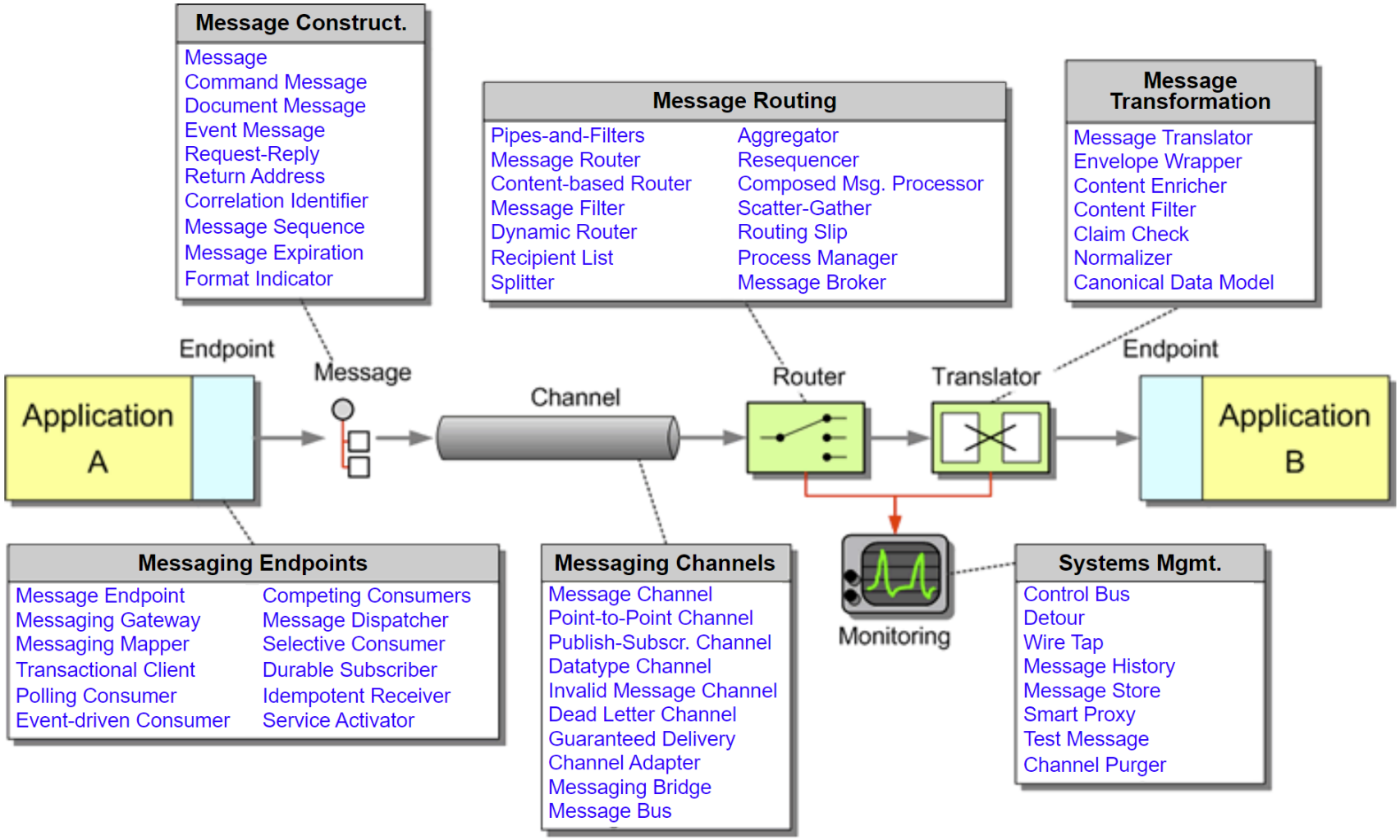
GRPC: Die Implementierung

```
class GreeterImpl : Greeter.GreeterBase
{
    // Server side handler of the SayHello RPC
    public override Task<HelloReply> SayHello(HelloRequest request, ServerCallContext context)
    {
        return Task.FromResult(new HelloReply { Message = "Hello " + request.Name });
    }

    // Server side handler for the SayHelloAgain RPC
    public override Task<HelloReply> SayHelloAgain(HelloRequest request, ServerCallContext context)
    {
        return Task.FromResult(new HelloReply { Message = "Hello again " + request.Name });
    }
}
```

Message-Broker

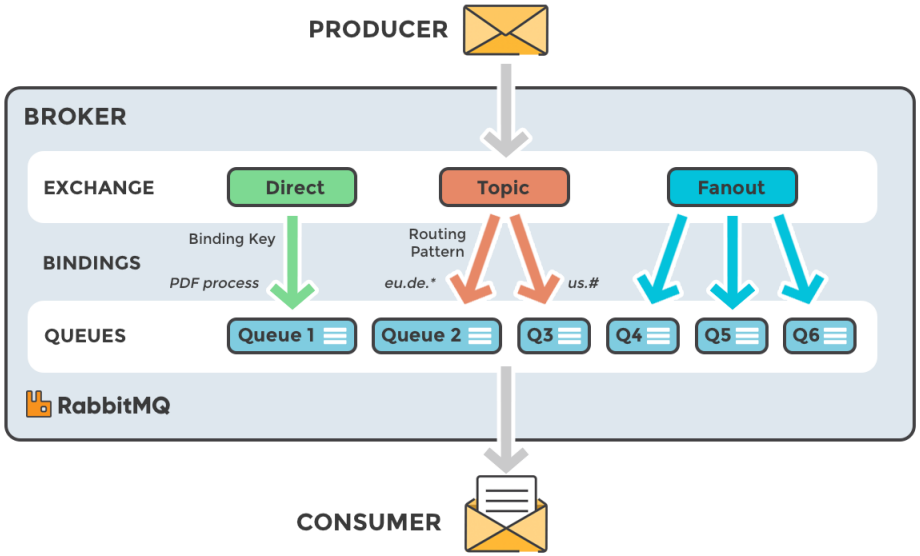
- Standalone Software-Komponente, welche die Aufgabe der Message-Verarbeitung übernimmt
- Eigenschaften:
  - Sender und Empfänger müssen sich nicht kennen
  - Es erfolgt eine zeitliche Entkoppelung, weil der Empfänger auch zeitverzögert abarbeiten kann
  - Der Broker ist außerhalb der Applikation und erlaub es daher, e.g. den Sender neu zu starten, ohne dass Nachrichten verloren gehen.
- Viele Patterns aus https://www.enterpriseintegrationpatterns.com/patterns/messaging/index.html wurden On-The-Top mit e.g. https://particular.net/nservicebus implementiert
  - Wichtig: Nutzen Sie die Nomenklatur dieser Seite – de facto anerkannter Standard

Connectoren: Message-Broker



Message-Broker: RabbitMQ

- Producer: Erzeugt Nachricht – Consumer: Konsumiert Nachricht
- Exchanges: Dienen nur zur Weiterleitung von Nachrichten – merken sich also keine Nachrichten.
  - **Direct**: Leitet Nachricht an eine bestimmte Queue weiter
  - **Topic**: Jede Nachricht kann einen Routing-Key haben → Weiterleitung anhand des Routing-Keys per Pattern Matching: somekey.*.foo.*.bar.#
  - **Fanout**: Wird an jede verbundene Queue weitergeleitet



Message-Broker: RabbitMQ

- Nachrichten in Queue bleiben solange vorhanden bis
  - Konsumiert von Consumer
  - TTL (Time-To-Live) abläuft (wenn angegeben)
  - Service neu gestartet wird (wenn Nachricht nicht als persistent markiert wurde)
- Mit anderen Worten: Wenn zwei Consumer die Nachricht benötigen → 2 Queues + Fanout
  - Intern speichert RabbitMQ die Nachricht meist nur 1 Mal – aber: es muss über beide Consumer „buchführen“

RabbitMQ: Publisher

```
Task PublishAsync<T>(
    IExchange exchange,
    string routingKey,
    bool mandatory,
    IMessage<T> message,
    CancellationToken cancellationToken = default
);
```

RabbitMQ: Consumer

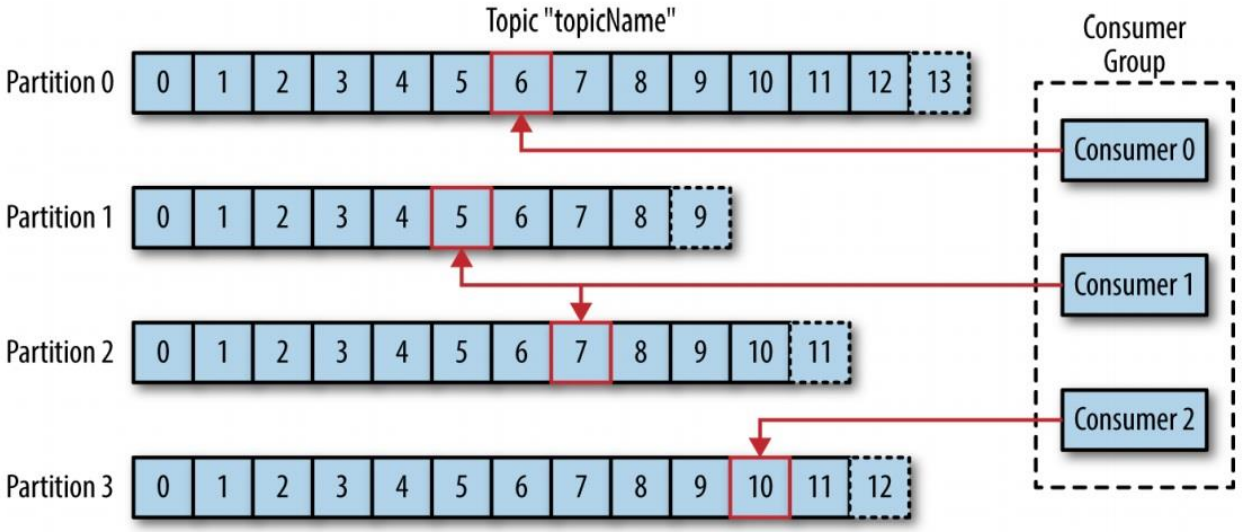
```
IDisposable Consume(IQueue queue, MessageHandler onMessage, Action<IConsumerConfiguration> configure);
```

RabbitMQ Fazit

- Nachrichten verschwinden nach dem Konsumieren
- Wenn Queues von 2 Consumers gelesen (https://www.enterpriseintegrationpatterns.com/patterns/messaging/CompetingConsumers.html)
  - Jeder Consumer liest seine Nachricht (eine Nachricht einer Queue wird nicht doppelt gelesen)
- Der Publisher weiß nicht, ob ein Consumer die Nachricht jemals verarbeitet hat
  - Muss auf höherer Ebene gelöst werden

Message-Streams: Kafka

- Nachricht bleibt nach dem „konsumieren“ bestehen
- Konsumieren heißt bei Kafka: Offset (ähnlich File-Descriptor) weiter zu schieben
  - D.h. lesen einer „Queue“ ist lesen wie aus einer File
  - Bei Kafka spricht man von einem Topic

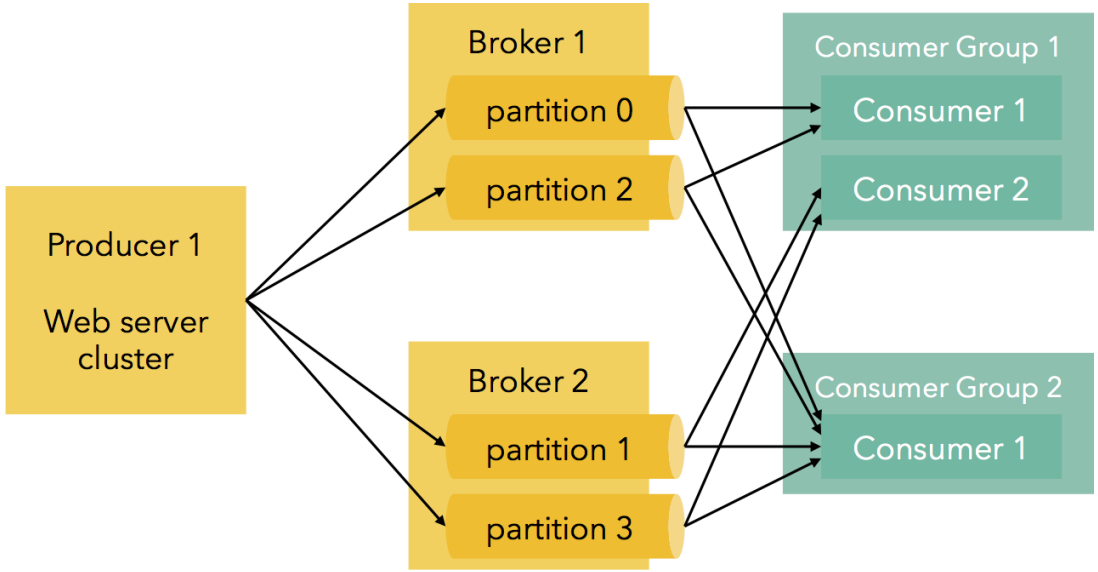


Message-Streams: Kafka

- Partition sind Unterteilungen in einem Topic
- Erlauben das Verteilen auf unterschiedliche physikalische Knoten
  - Auch wenn am Anfang alles auf einem Host läuft → mehrere Partions wählen → leichter, wenn Partition auf anderen Host verschoben wird
- Consumer-Groups: Lesen parallel die Nachrichten
- Stirbt ein Consumer: Übernimmt ein anderer die Aufgabe von diesem (wird von Kafka / Zookeeper gelöst)
  - Nachricht wird aber garantiert nur von einem Consumer verarbeitet

Connector: Message-Streams: Kafka

- Mehrere Consumer-Groups: Nachrichten aus Partition 0 werden von beiden Consumer-Groups konsumiert (innerhalb der Consumer-Group allerdings nur einmal)

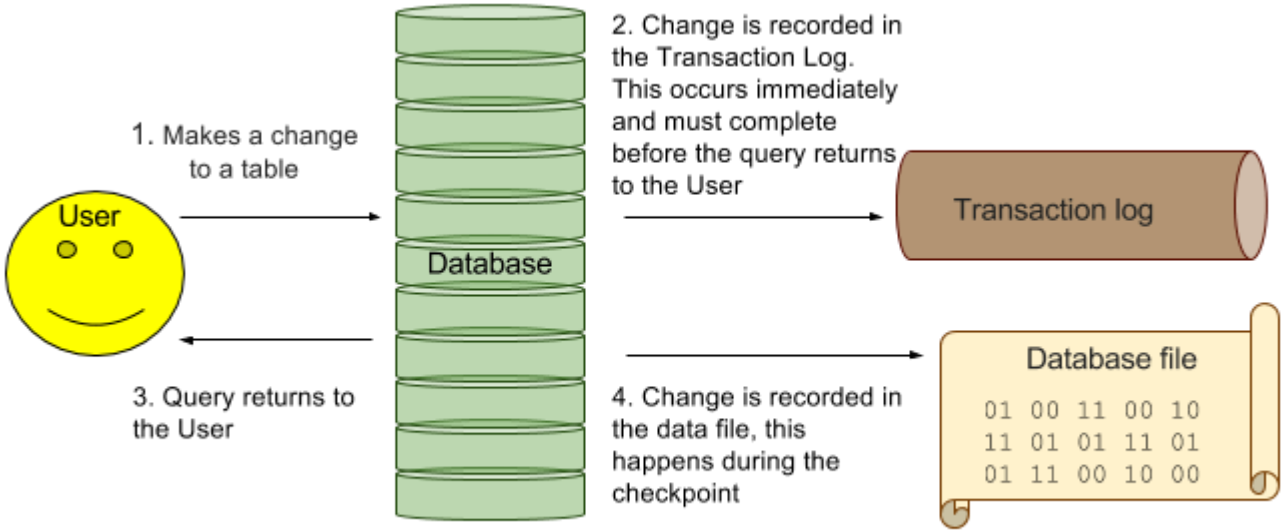


Connector: Message-Streams: Kafka: CDC

- CDC (Change-Data-Capture)
- Anwendungsbeispiel: Komponenten A speichert Kundendaten in einen SQL Server. Komponenten B braucht den Namen / Adresse der Kunden
- Möglichkeiten:
  - Man kann Änderungen per RabbitMQ an Komponente B schicken. Was passiert, wenn Komponente B abstürzt? Neuinitialisierung = ich brauche einen initialen Snaphot + Deltas
  - Per REST: Wenn Komponenten B nicht da ist, muss Komponente A sich das merken (Was weiß B schon?) oder Komponente B pollt Komponente A

Exkurs: Write-Ahead-Log (WAL)

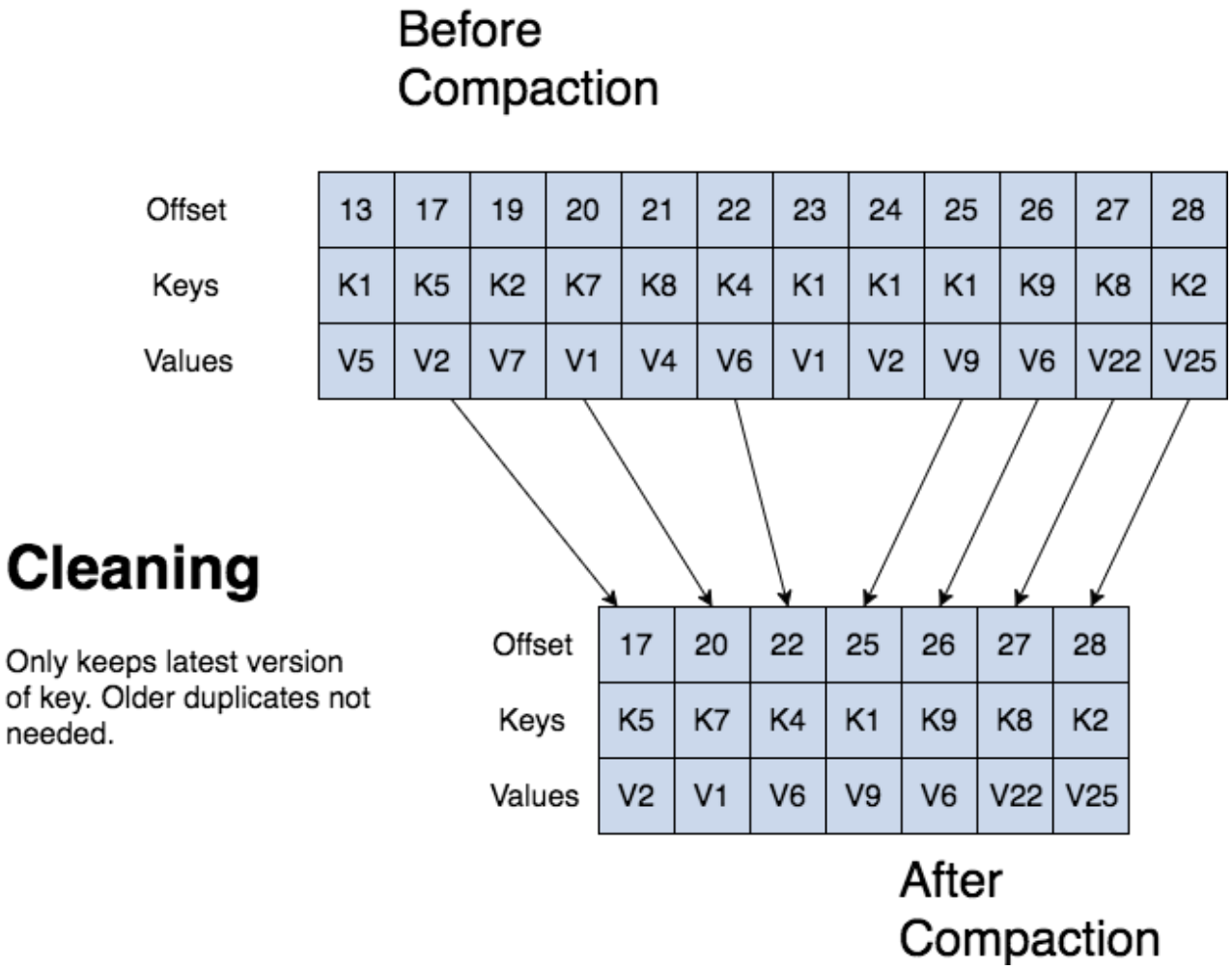
- Daten werden in ein Transaction-Log geschrieben, bevor sie in die eigentliche Datenbank File kommen
- Diese Log kann man mit Bordmitteln auslesen
- Oder man nimmt fertige Tools wie https://debezium.io/



Message-Streams: Kafka: CDC

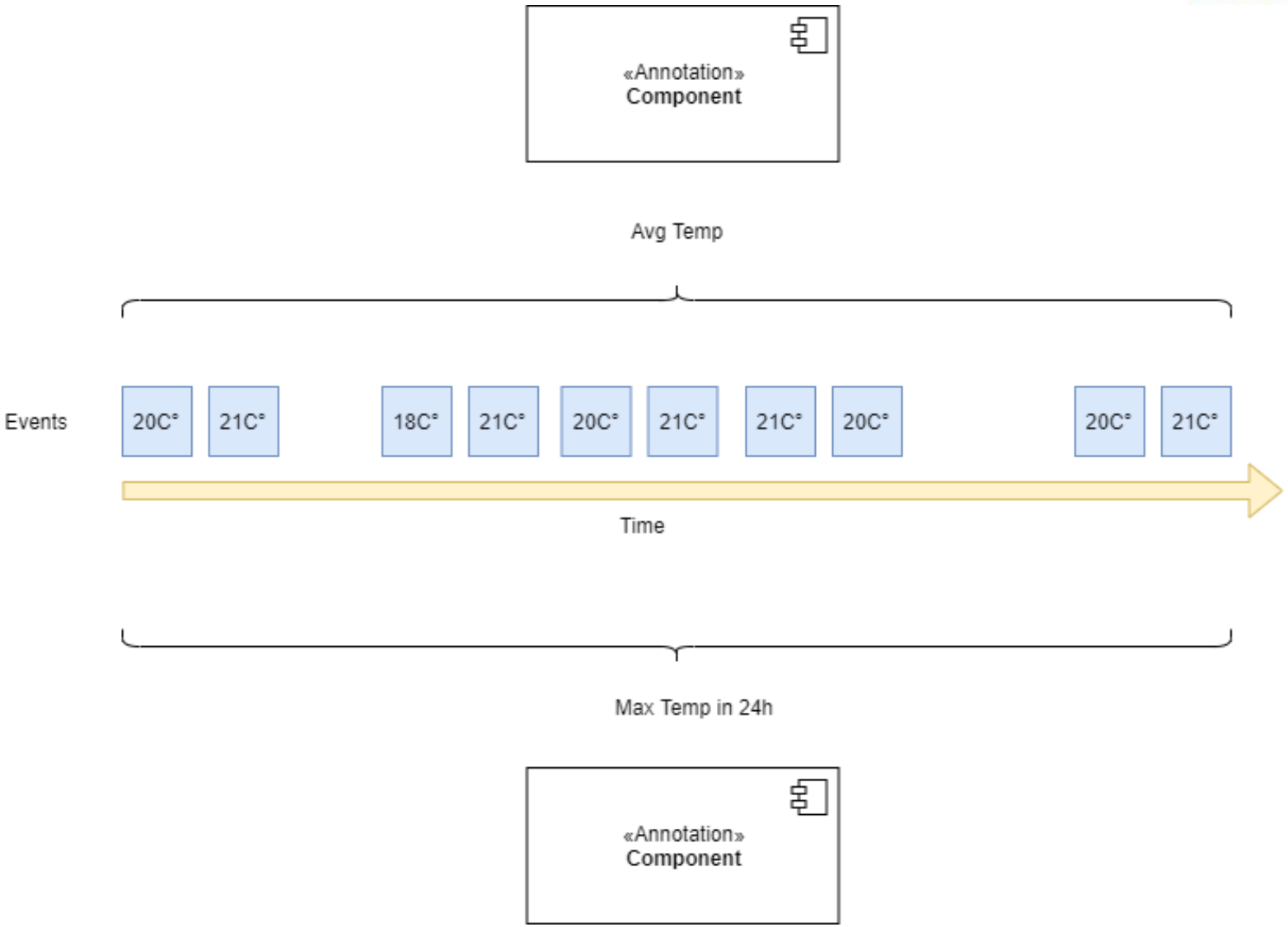
- Ich bekomme eine konsistente Ansicht auf die Kundendaten
- Diese Daten befinden sich in Kafka → Consumer können die Änderungen auslesen (e.g. `AddressChanged`)
- Wenn sich die Daten mehrfach ändern, muss ich mir eigentlich nur die letzte Änderung merken (Optimierung) → Log-Compaction

Message-Streams: Kafka: CDC



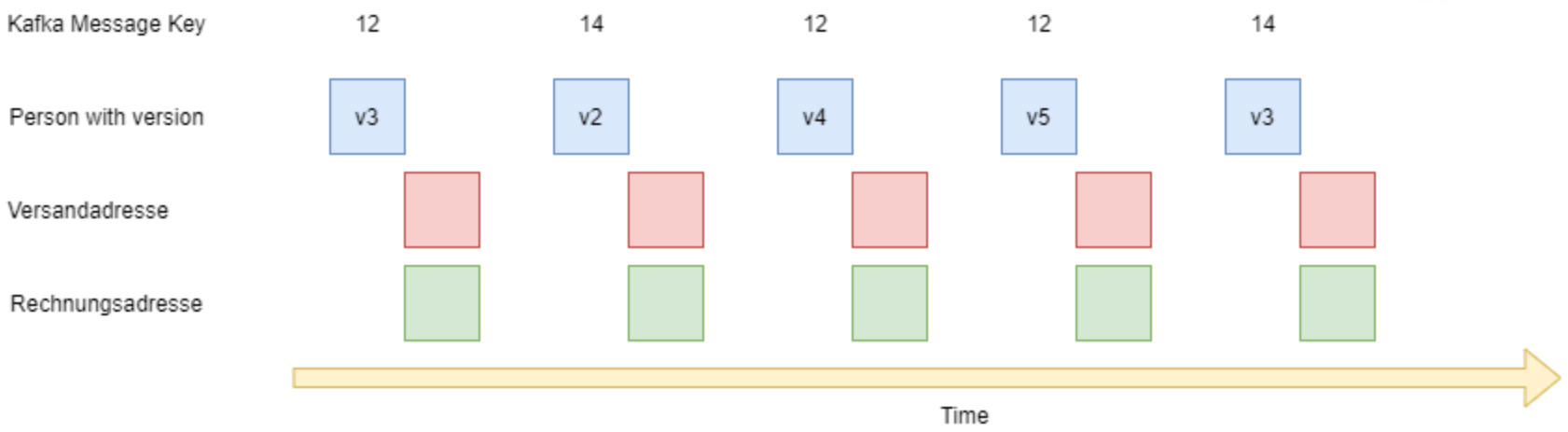
Praktische Anwendung Kafka

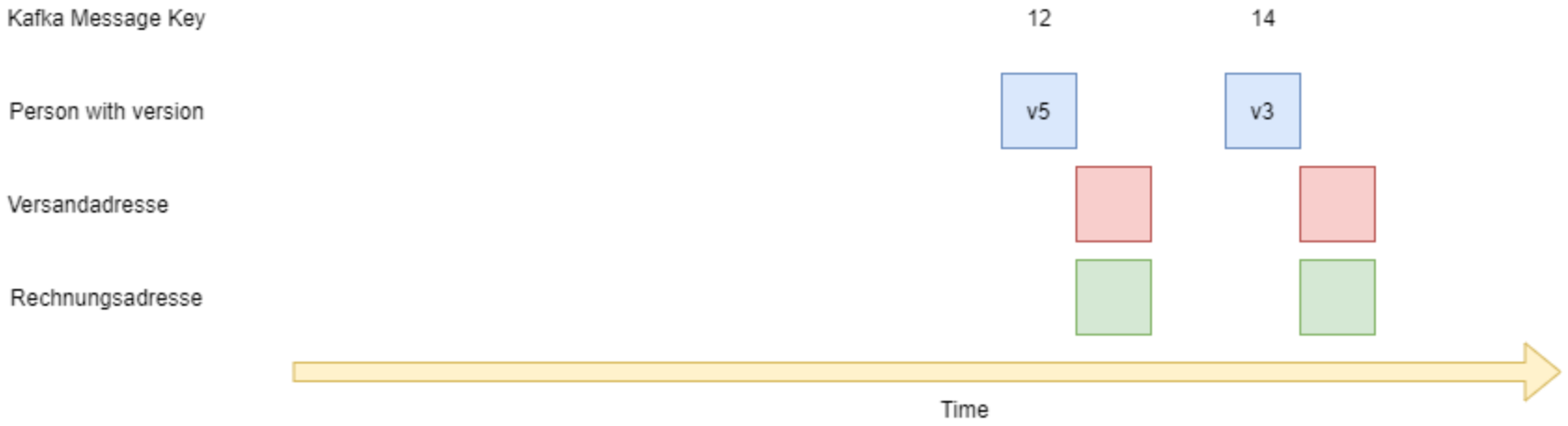
- Retention (d.h. Nachrichten werden nach einer gewissen Zeit verworfen)
- Eine Komponente generiert Events
  - Events sind etwas, was bereits geschehen ist → unveränderbar
- Beispiel: Wir haben eine Komponente, welche die Raumtemperatur misst und in Kafka „published“
- Andere Komponenten wollen auf diese Events reagieren
  - Komponente A: ist interessiert an Events, deren Wert den Threshold 20 Grad übersteigt
  - Komponente B: ist an der Durchschnittstemperatur interessiert
  - Nach einem Jahr kommt Komponente C: Will den Temperaturabfall in der Nacht bestimmen – kein Problem – alte Events noch vorhanden (sofern nicht durch retention verworfen)


«Annotation»
Component
Avg Temp
Events
20C°
21C°
18C°
21C°
20C°
21C°
21C°
20C°
20C°
21C°
Time
Max Temp in 24h
«Annotation»
Component

Praktische Anwendung Kafka II

- Log-Compaction: Wir geben jedem Event einen Key
- Kafka verwirft (je nach Einstellung) alle „alten“ Nachrichten und behält mindestens die letzte Version des Keys
- Beispiel: Komponente hat Personendaten (inkl. Versand- und Rechnungsadresse)
- Komponente A: Interessiert sich für die Versandadresse
- Komponente B: Interessiert sich für Rechnungsadresse
- Um Komponenten autonom agieren zu lassen → Daten werden in den Komponenten (Services) lokal gespeichert
- Wenn diese Daten aber durch Absturz nicht mehr aktuell sind → wie kann ich mir den Datenstand wieder aufbauen?
- Stream erneut lesen → von Interesse nur die letzten Änderungen


Kafka Message Key
12
14
12
12
14
Person with version
v3
v2
v4
v5
v3
Versandadresse
Rechnungsadresse
Time

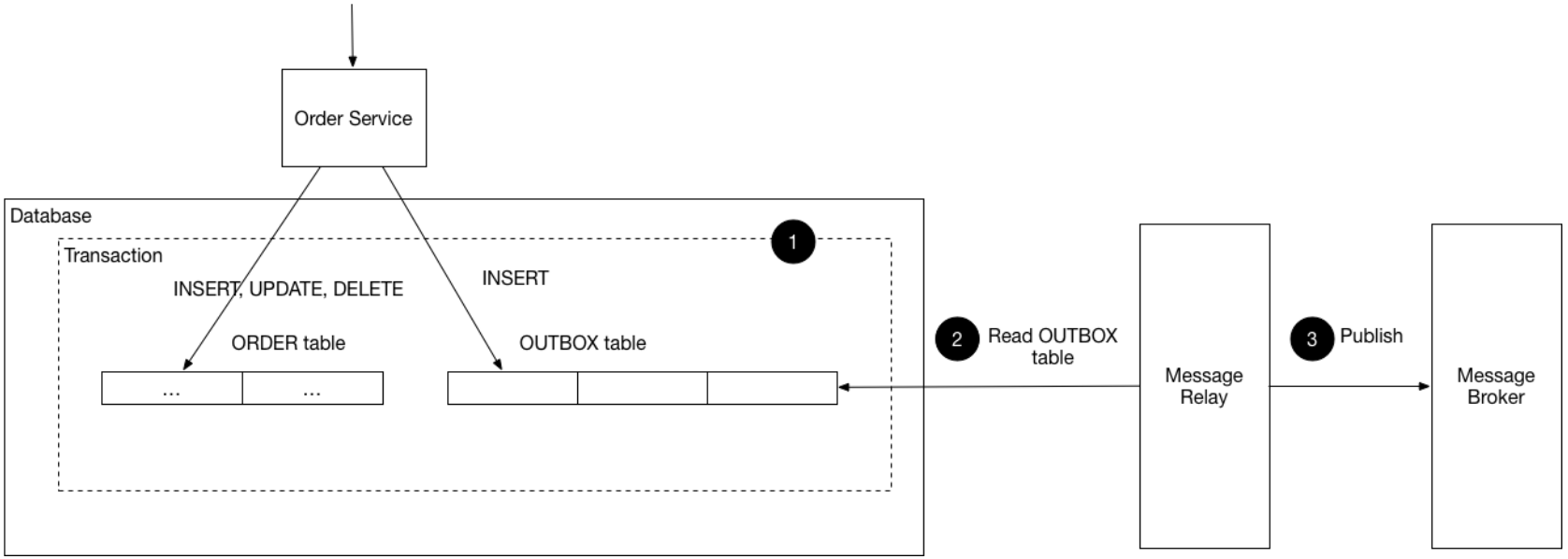

Kafka Message Key
12
14
Person with version
v5
v3
Versandadresse
Rechnungsadresse
Time

Kafka Challenge: Transaction

- Szenario:
  - Ich bekomme eine Anfrage: „Ändere der Namen des Kunden 12“
  - Speichern in Datenbank
  - Service crashed
  - ~~Speichern des Events `CustomerChanged` in Kafka~~
- Andere Services bekommen es nie mit …
- Lösungen:
  - Outbox Pattern
  - Log Tailing (haben wir bereits beschrieben bei CDC – WAL)

Kafka: Outbox Pattern

- Challenge auch im Message-Relay (kann crashen)
  - Messages können 2 Mal gepublished werden
  - Kann u.a. auf Consumer-Seite abgefangen werden (Idempotent Consumer)



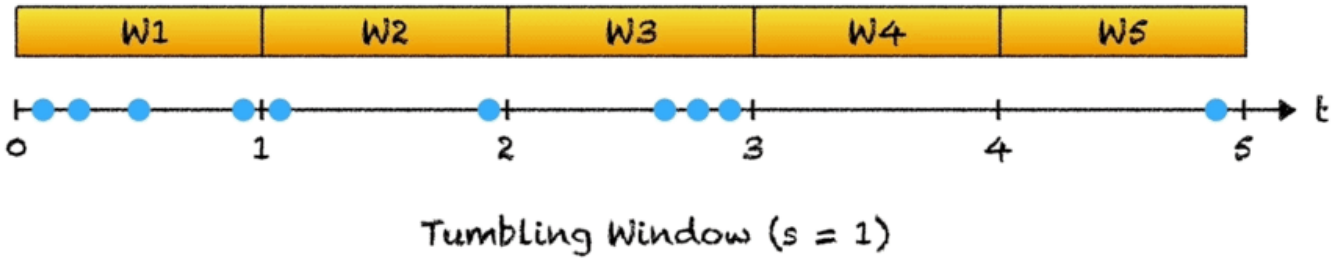
Quelle: https://microservices.io/patterns/data/transactional-outbox.html

Message-Streams: Kafka: ETL

- Wikipedia: Extract, Transform, Load (ETL) ist ein Prozess, bei dem Daten aus mehreren gegebenenfalls unterschiedlich strukturierten Datenquellen in einer Zieldatenbank vereinigt werden.
  - Extraktion: der relevanten Daten aus verschiedenen Quellen
  - Transformation: der Daten in das Schema und Format der Zieldatenbank
  - Laden: der Daten in die Zieldatenbank
- Rechts und Links befinden sich meist Komponenten, welche die Services nutzen

Message-Streams: Kafka: Stream-Analytics

- Es wird oft Batch-Processing bei ETL angewandt (e.g. Prozess läuft in der Nacht und kopiert von Oracle → Kafka → Elastic)
- Nachteil: Hohe Latency (=wann Daten in Elastic verfügbar sind)
- Stream-Driven-Architecture! (https://www.amazon.de/Streaming-Architecture-Designs-Apache-Streams/dp/1491953926)
- E.g. Window-Functions erlauben uns Aggregationen auf fein-gesteurte Zeitfenster zu machen (aber Thema in Datenverarbeitungs Vorlesung – daher nur kurz angeschnitten in dieser Vorlesung)



Kafka vs. RabbitMQ

Situation	Kafka	RabbitMQ
Message Konsumieren	Position wird weiter geschoben (Nachricht bleibt aber weiterhin da)	Nachricht verschwindet nach dem ACK
Message Routing	Kafka Streams (externe App)	Build-In: Topic Exchange mit Routing-Key
Admin UI	Einige 3rd Party – like „UI for Apache Kafka“	Build-In (sehr gut zum Debuggen)
Komplexität Verständnis	Schwer	Leicht
Performance	Sehr Hoch (Append-Log)	Mittel
Message Priority	Nein	Ja
Message Größe	Idealerweise 1MB	Idealerweise bis zu 128MB
Protokoll	Binär	Text-Basiert (Level-7 Firewall?)

ZeroMQ – I‘m brokerless

- Performance und Single-Point-Of-Failure (Message-Broker) könnte ein Grund für die Verwendung sein
- An allen Ecken und Enden optimiert
  - Wie Pakete möglichst vollstopfen?
- Viele Bindings in vielen Sprachen
- In Windows etwas „naja“
  - Linux: epoll, FreeBSD: kqueue
  - Windows: IO Completion ports (IOCP) → leider nicht unterstützt
  - Lösung: https://github.com/zeromq/netmq (100% nativ - aber vorher gut testen → andere .NET Client nutzt C-Binding)

ZeroMQ

```
using (var client = new RequestSocket())
{
    client.Connect("tcp://127.0.0.1:5556");
    client.SendFrame("Hello");
    var msg = client.ReceiveFrameString();
    Console.WriteLine("From Server: {0}", msg);
}
```

```
using (var server = new ResponseSocket())
{
    server.Bind("tcp://*:5556");
    string msg = server.ReceiveFrameString();
    Console.WriteLine("From Client: {0}", msg);
    server.SendFrame("World");
}
```

ZeroMQ

```
using (var subscriber = new SubscriberSocket())
{
    subscriber.Connect("tcp://127.0.0.1:5556");
    subscriber.Subscribe("A");

    while (true)
    {
        var topic = subscriber.ReceiveFrameString();
        var msg = subscriber.ReceiveFrameString();
        Console.WriteLine("From Publisher: {0} {1}", topic, msg);
    }
}
```

```
using (var publisher = new PublisherSocket())
{
    publisher.Bind("tcp://*:5556");

    int i = 0;

    while (true)
    {
        publisher
            .SendMoreFrame("A") // Topic
            .SendFrame(i.ToString()); // Message

        i++;
        Thread.Sleep(1000);
    }
}
```

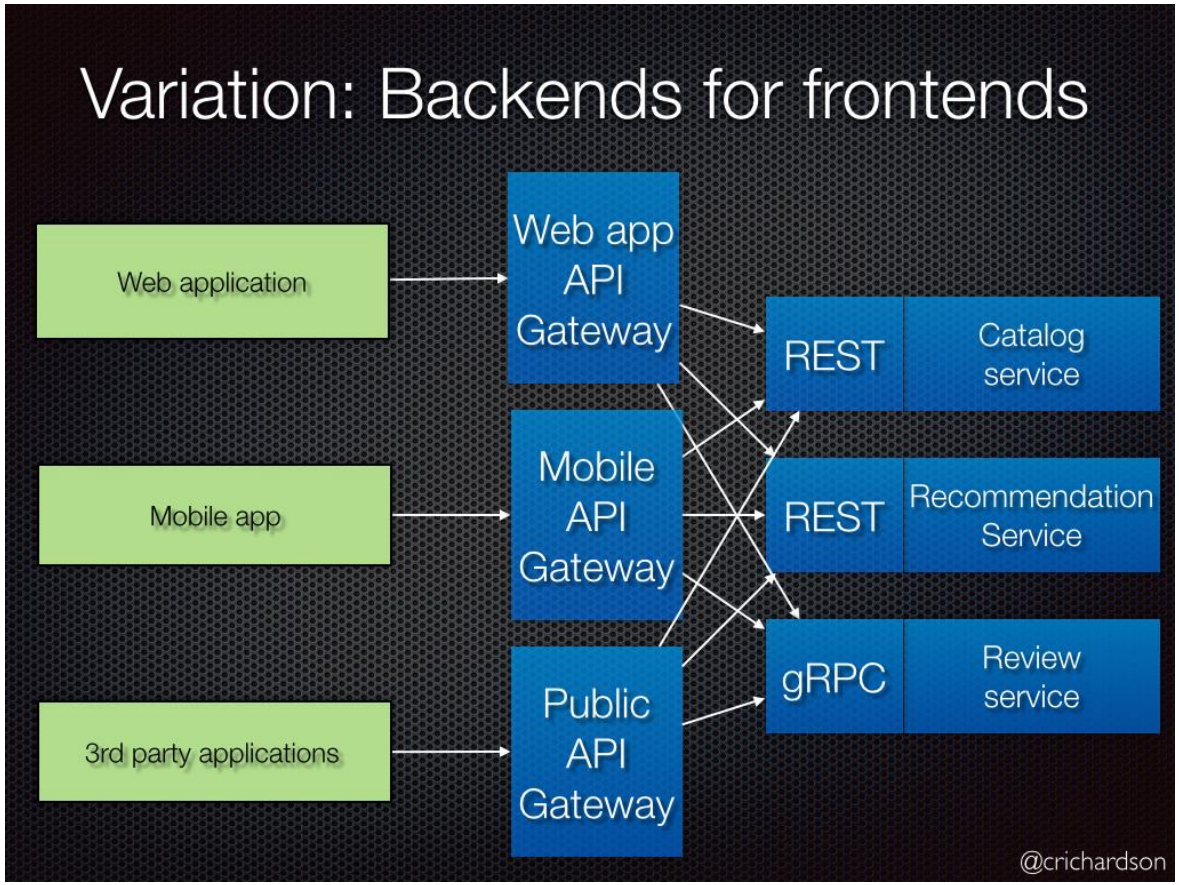
FH University of Applied Sciences TECHNIKUM WIEN

Gemeinsame Aufgaben und Services abstrahieren

API-Gateway

- Aus der 1. Vorlesung: Problem-Decomposition
  - Komplexe Probleme in Sub-Probleme zerteilen
- Aufrufe zu einzelnen Komponenten / Services orchestrieren
  - Unterschiedliche Anforderungen der Clients
  - Unterschiedliche Voraussetzungen (e.g. Netzwerk-Bandbreite)
  - Refactorings im Hintergrund einfacher
- Übernimmt aber auch allgemeine Aufgaben:
  - Authentifikation, Metrics, Traffic-Control, …
- Beispiel:
  - https://www.krakend.io/
  - https://github.com/Netflix/zuul

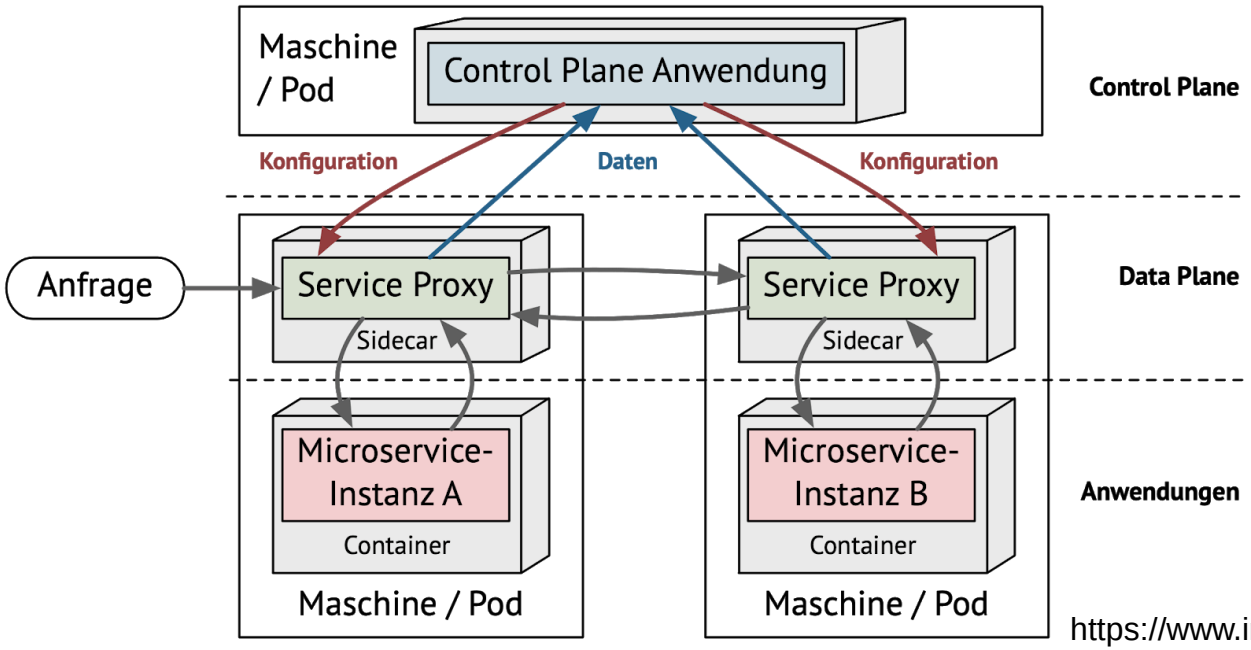
API-Gateway



https://microservices.io/patterns/apigateway.html

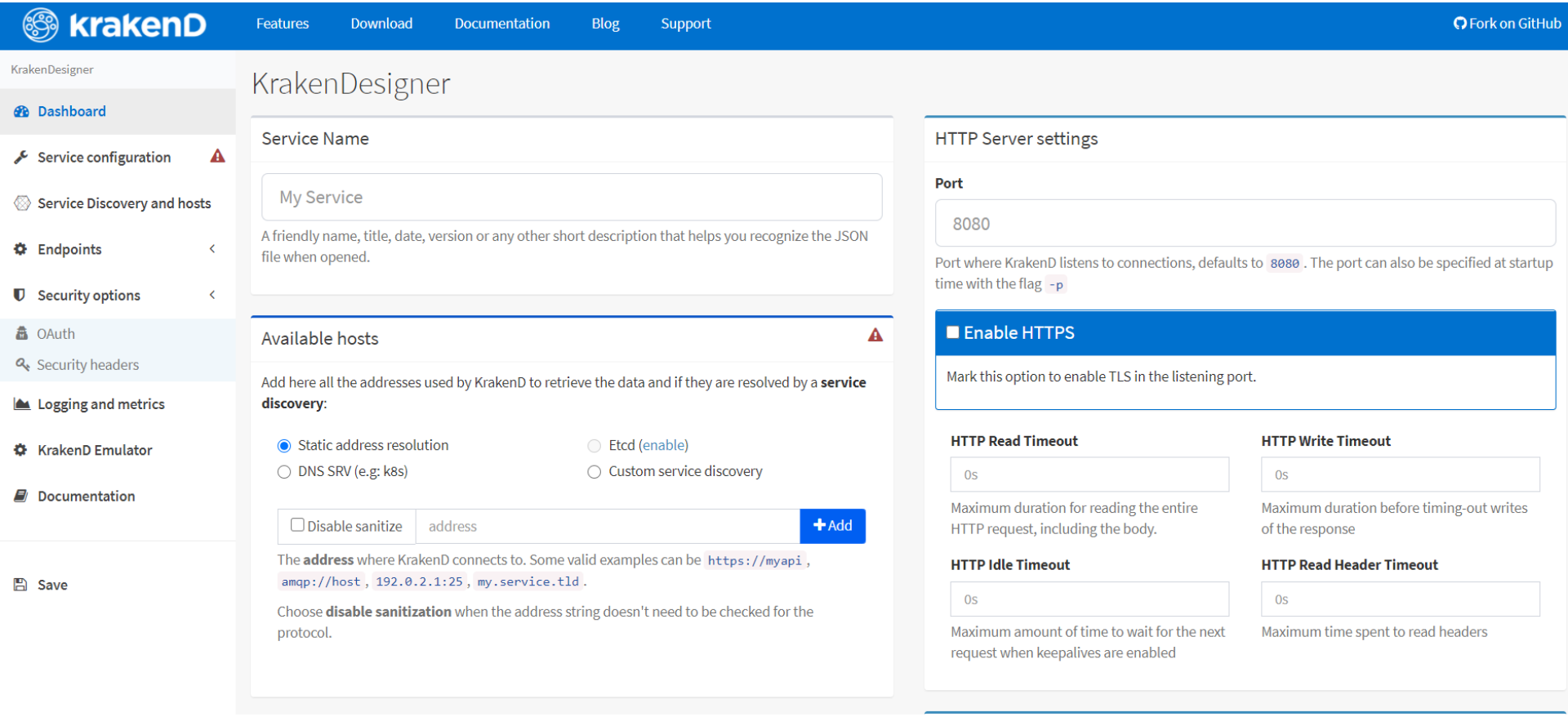
Service-Mesh

- Keine Frameworks oder Libraries
- Es wird ein ganzes Service zum Microservice zur Verfügung gestellt, welche die Aufgabe des Tracing, Logging, Curcuit Breaker, … übernimmt
- Auch bekannt unter „Sidecar Pattern“



https://www.innoq.com/de/articles/2019/10/service-mesh/

Kraken



https://designer.krakend.io/#!/service

Challenging Microservice: Dapr

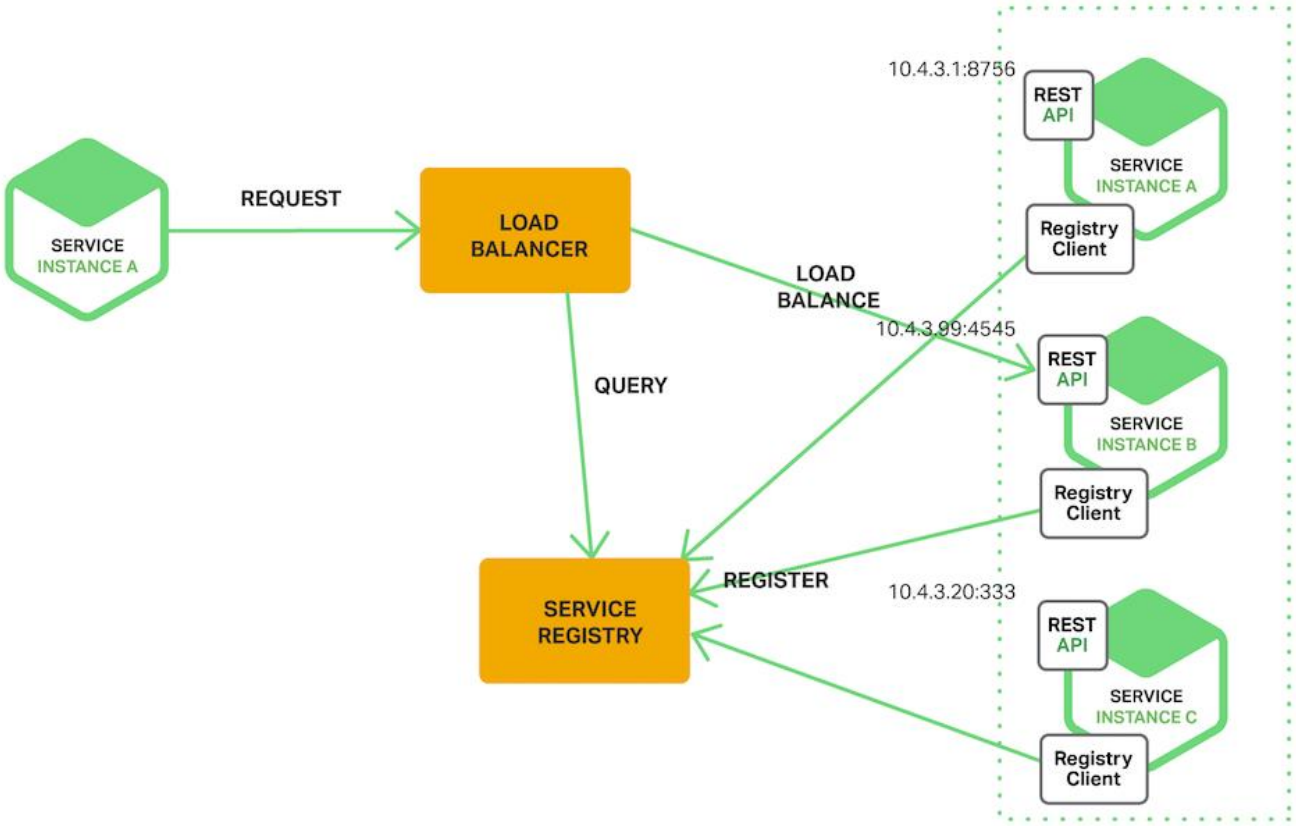
- Building Blocks
  - Service Invocation: gRPC oder HTTP
  - State Management: Redis, MongoDB, …
  - Publish & Subscribe
  - Securing Secrets
  - Bindings: Using bindings, you can trigger your app with events coming in from external systems, or interface with external systems.
  - Actors
- https://docs.microsoft.com/en-us/dotnet/architecture/dapr-for-net-developers/getting-started

Wo bist du? Komponenten / Services finden

Service-Discovery: Wie finden sich Komponenten?

- **Möglichkeit 1**: Komponenten / Services in Konfiguration hinterlegen (meist dann auch ein Backup-Service)
- **Möglichkeit 2**: Abstraktion – Service Discovery
- Eine Tool dafür ist Consul (https://www.consul.io/)
- Ermöglicht Services zu finden (typischer DNS oder Custom API)
- Health Checking: Consol prüft, ob Services / Komponenten noch leben
- Inkludiert einen Key / Value store

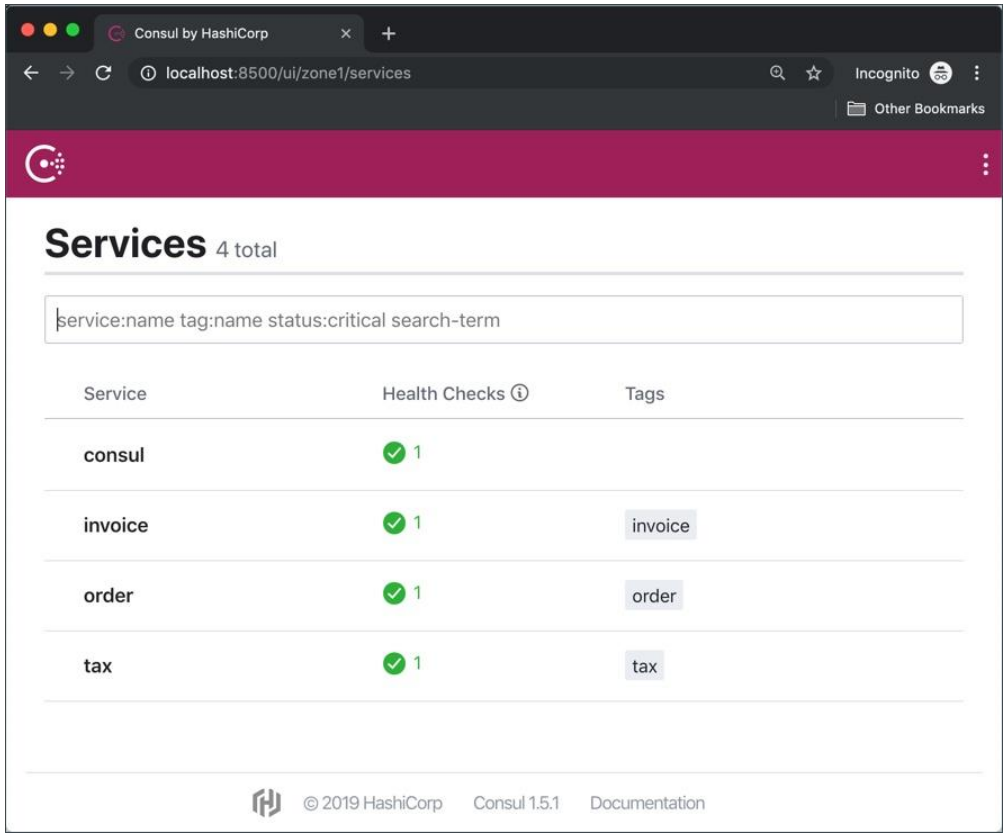
Service Discovery Beispiel



Quelle: nginx.com

Service-Discovery: Consul

```
{"service": {"name": "order", "tags": ["order"], "port": 5000}}
```



Service-Discovery: Consul

- Abfrage per DNS

```
dig @127.0.0.1 -p 8600 order.service.consul SRV
```

- Abfrage per HTTP

```
curl http://localhost:8500/v1/catalog/service/order
```

Service-Discovery: Consul

2. bash

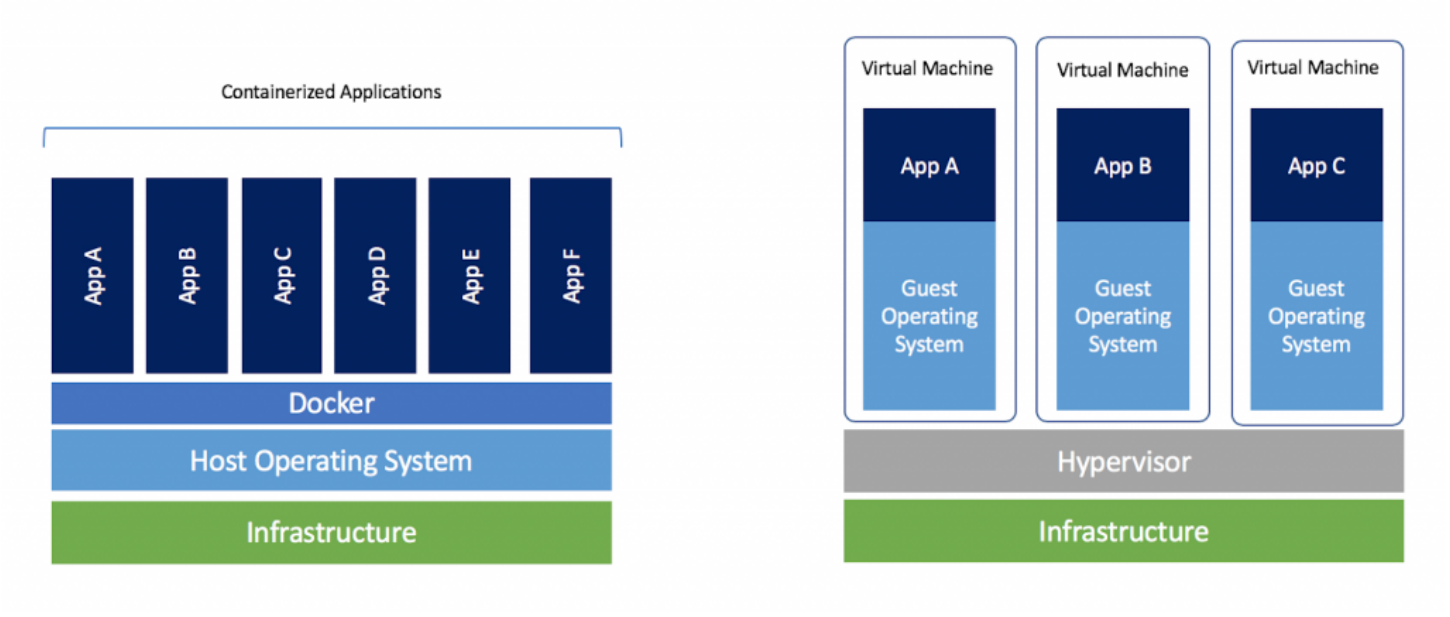
```
[~]$ curl -s http://localhost:8500/v1/catalog/service/order | jq
[
  {
    "ID": "256682b2-1b68-6477-82f7-68764049b977",
    "Node": "host1",
    "Address": "127.0.0.1",
    "Datacenter": "zone1",
    "TaggedAddresses": {
      "lan": "127.0.0.1",
      "wan": "127.0.0.1"
    },
    "NodeMeta": {
      "consul-network-segment": ""
    },
    "ServiceKind": "",
    "ServiceID": "order",
    "ServiceName": "order",
    "ServiceTags": [
      "order"
    ],
    "ServiceAddress": "",
    "ServiceWeights": {
      "Passing": 1,
      "Warning": 1
    },
    "ServiceMeta": {},
    "ServicePort": 5000,
    "ServiceEnableTagOverride": false,
    "ServiceProxyDestination": "",
    "ServiceProxy": {},
    "ServiceConnect": {},
    "CreateIndex": 11,
    "ModifyIndex": 11
  }
]
```

FH University of Applied Sciences TECHNIKUM WIEN

Komponenten hosten

Komponenten / Services hosten: Docker

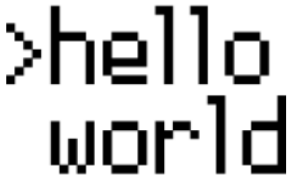
- Wikipedia: Docker vereinfacht die Bereitstellung von Anwendungen, weil sich Container, die alle nötigen Pakete enthalten, leicht als Dateien transportieren und installieren lassen. Container gewährleisten die Trennung und Verwaltung der auf einem Rechner genutzten Ressourcen.



Komponenten / Services hosten: Docker

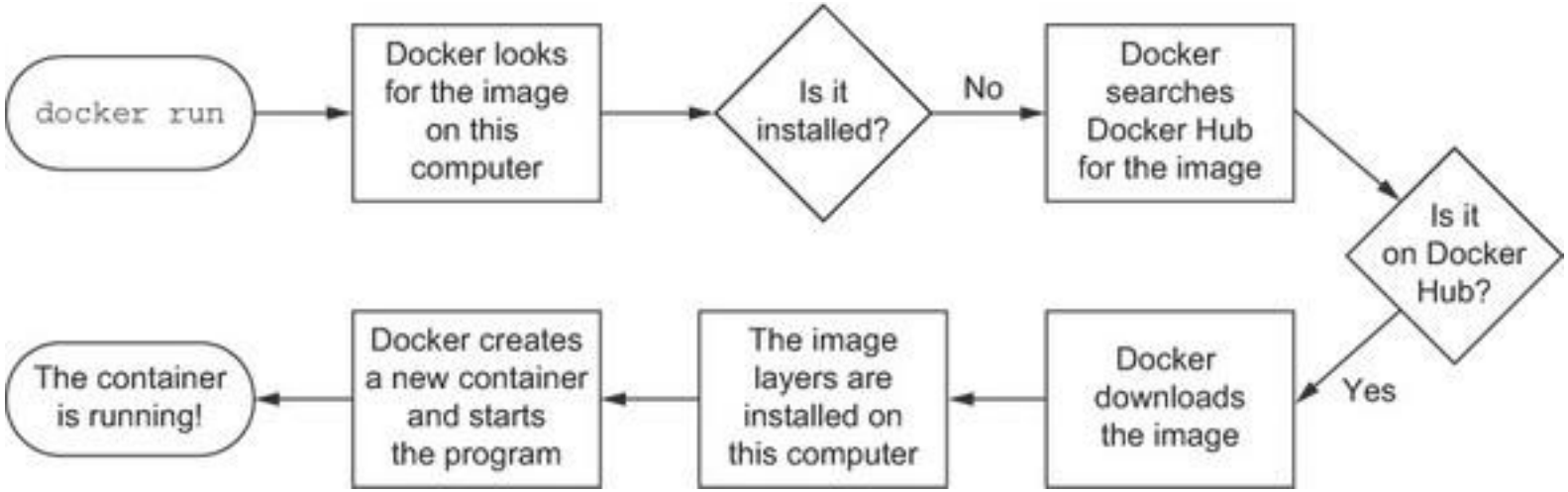
- `$ docker run hello-world`

```
Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
    (amd64)
 3. The Docker daemon created a new container from that image which runs the
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which sent it
    to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
 $ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker ID:
 https://hub.docker.com/

For more examples and ideas, visit:
 https://docs.docker.com/get-started/


$ docker images hello-world
REPOSITORY   TAG     IMAGE ID      SIZE
hello-world  latest  fce289e99eb9  1840
```



Komponenten / Services hosten: Docker

- Wichtig: State des Container ist an den State des Services im Container gebunden:
  - Wenn das Service läuft, läuft der Container
  - Wenn das Service nicht läuft, läuft der Container nicht

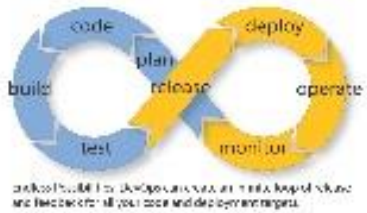


Überblick

Microservices

System Architecture: **Trends, Trends, Trends, …**

.msg



DevOps

Resource Virtualization

Platform as a Service (PaaS)

Deployment Automation

Configuration Automation

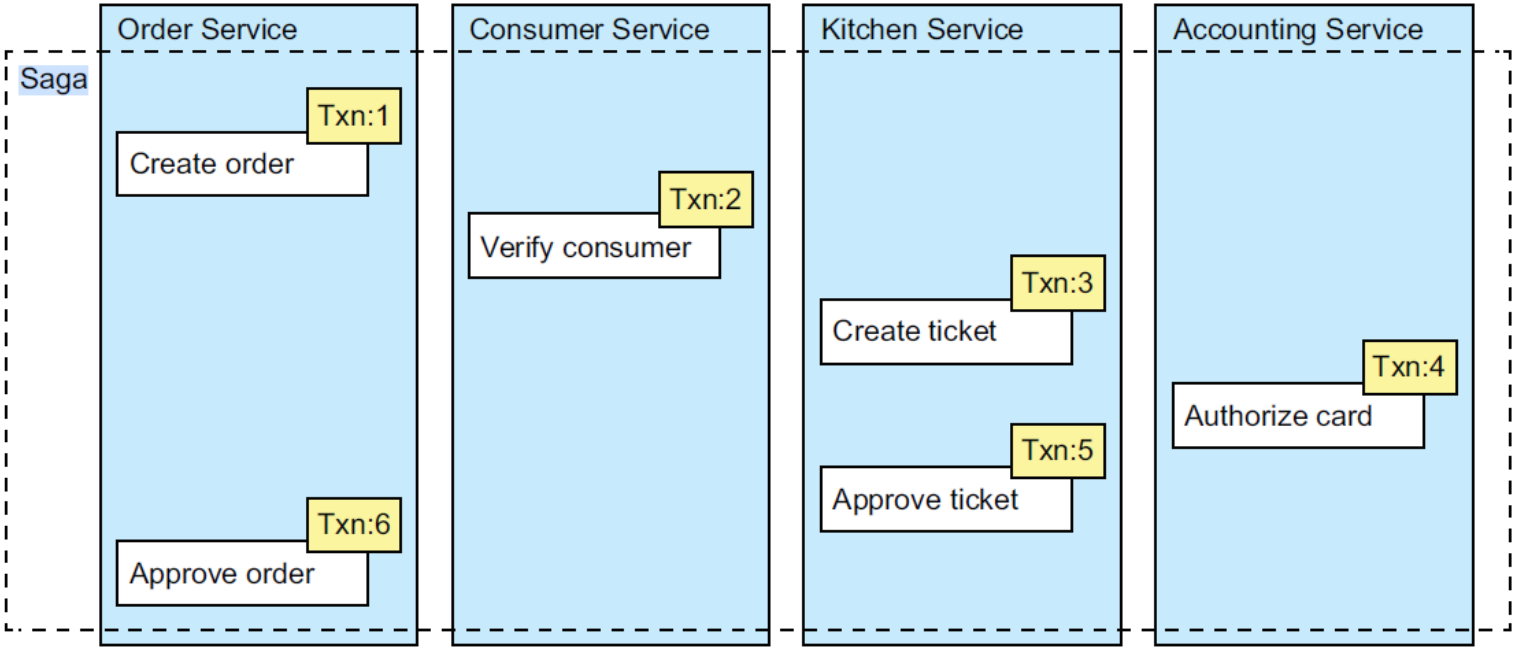
Clustering & Service Discovery

FH University of Applied Sciences TECHNIKUM WIEN

Komponentenaufrufe koordinieren

SAGA

- Ziel: Data Konsistenz wahren (wenn mehrere Services upgedated werden müssen)
- Zu jedem Schritte muss eine Kompensationsoperation existieren



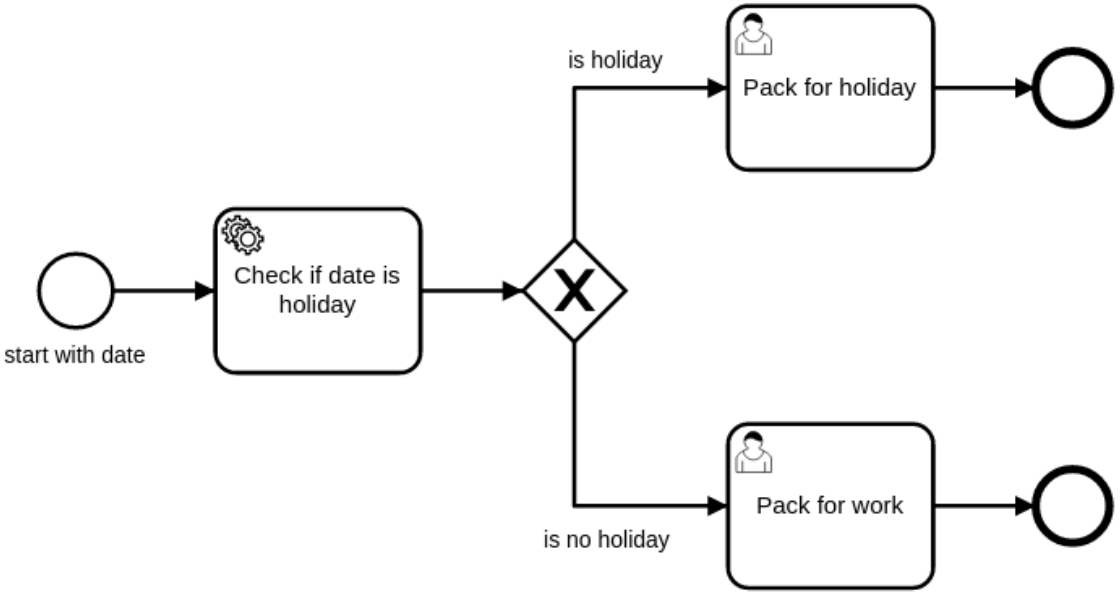
Richardson, C. (2018). Microservices patterns. Manning Publications Company,.

Leader Election

- Mehrere Komponenten machen die gleiche Aufgabe – einer muss die Führung übernehmen
- Ein Beispiel ist das Raft Protokoll
  - Viele Implementierungen https://raft.github.io/
  - E.g. etcd, consul
  - https://learn.hashicorp.com/tutorials/consul/application-leader-elections
- Kein Code-Beispiel → siehe Übung

Cammunda

- Mit Bedacht einsetzen – hat auch seine Nachteile …
- Vorteil: definitiv living documentation des Prozess, Abstraktion für session state, nicht UI only – auch coding möglich



FH University of Applied Sciences TECHNIKUM WIEN

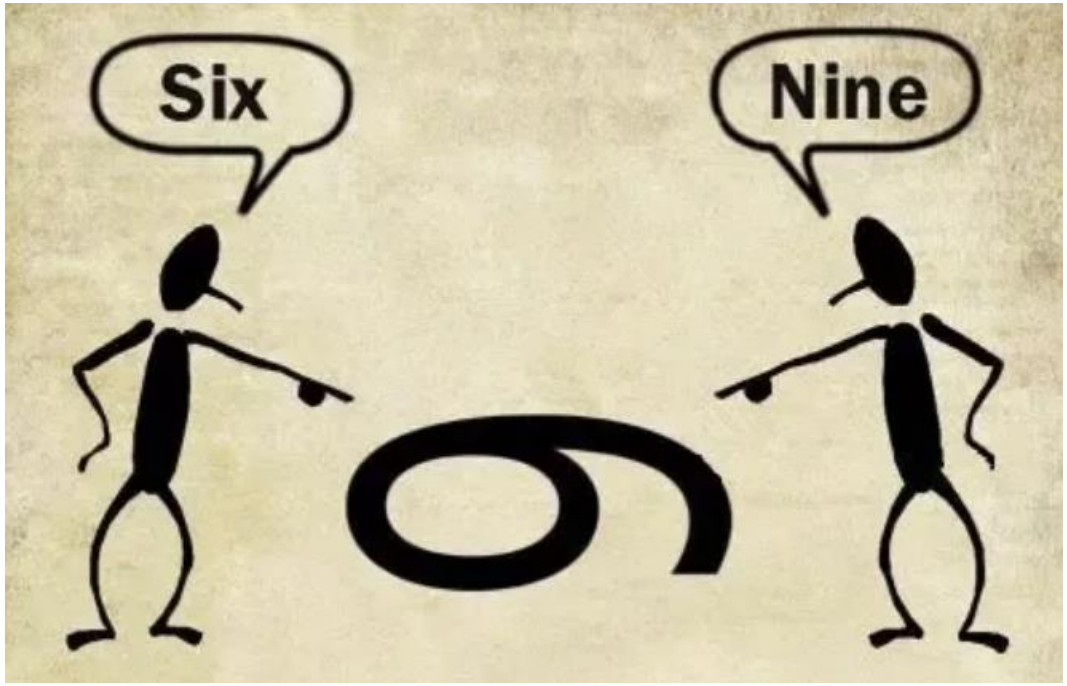
Inter-Komponenten Design

Inter-Komponenten Design

- Hat man festgestellt …
  - welche Komponenten es gibt
  - wie Schnittstellen, Connectoren ausschauen
- … stellt sich die Frage, wie eine Komponente intern ausschaut
- Spannende Themen - und vor allem unerschöpfliche Themenvielfalt – daher 3 Alltagsthemen:
  - Logging
  - Exception-Handling
  - Threading
- Thema sehr komplex → in der Vorlesung ist es das Ziel, einen Überblick zu bekommen

Before we start …

- Zu diesen Themen gibt es viele Meinungen … Und viele Ansichten …
- Wir schauen uns eine davon an

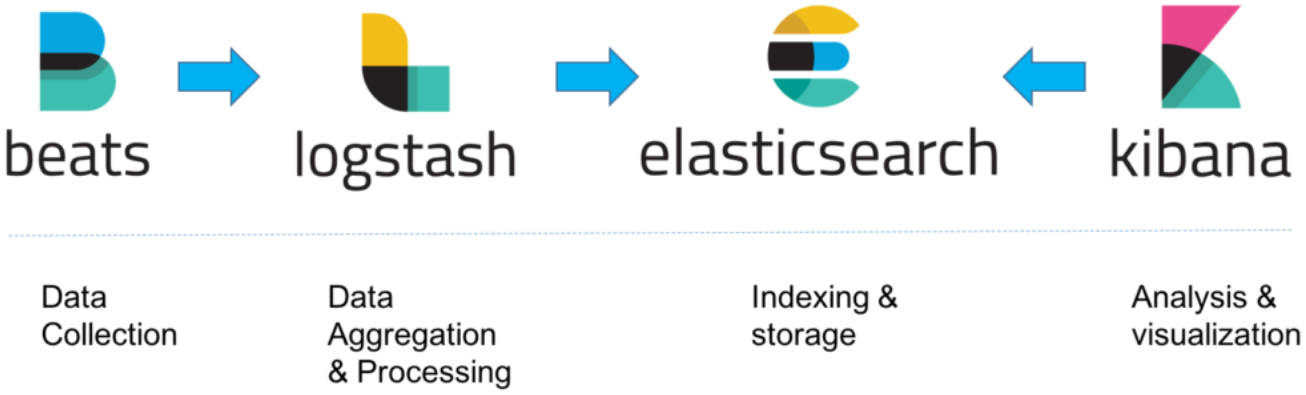


FH University of Applied Sciences TECHNIKUM WIEN

Wie Abläufe und Ereignisse festhalten?

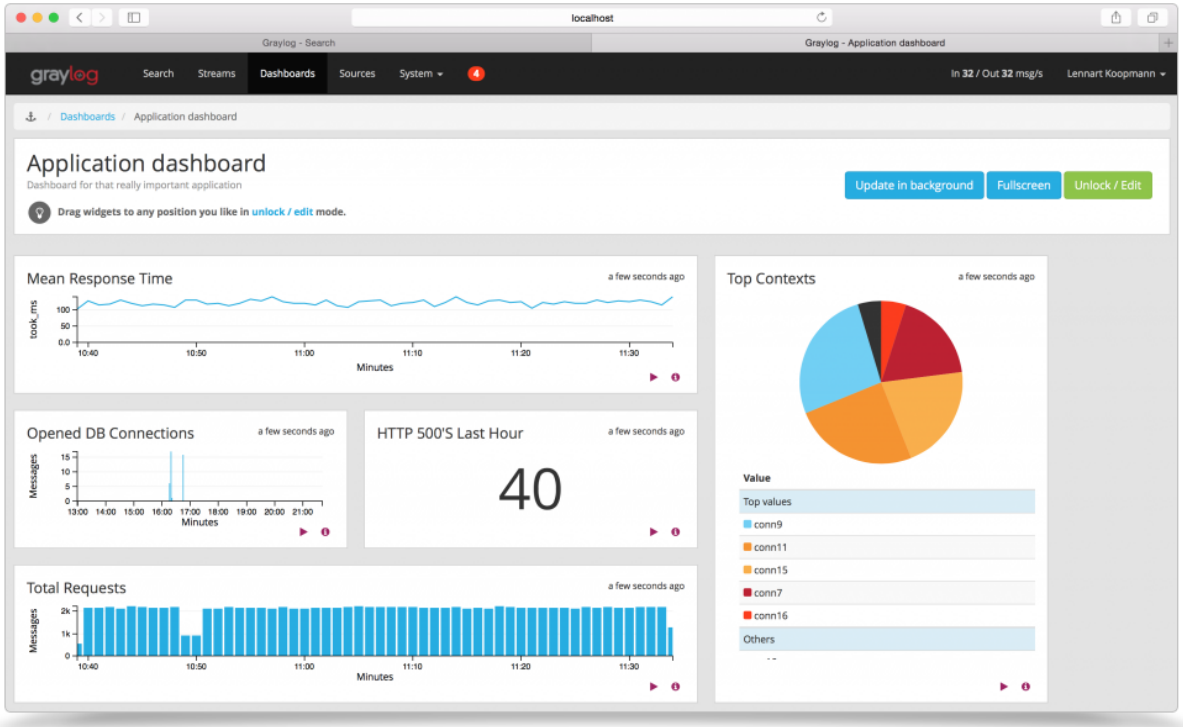
Logging: ELK Stack

- Hat man mal mehrere Komponenten auf unterschiedlichen Server – schaut man jeden Tag auf jedem Server in die Log-Files, ob es „etwas auffälliges“ gibt?
- Nein! Daher automatisieren



Graylog

- Wurde exklusiv für Log-Managemenet entwickelt
- In manchen Bereichen performanter als Logstash
- Einfacher zu konfigurieren



Logging: Semantic Logging mit Serilog

- Textual Logging
  - Meist durch Delimiter (e.g. Leerzeichen) getrennt
  - Wird oft von Menschen gelesen
  - Wenn von Maschinen / Applikationen … – … aufwendiges Parsing-Bingo …!

```
logEntry.Message = String.Format(
  @"Scaling request for role {0} has being successfully submitted.
The requested role instance count is {1}.
The scaling operation was triggered by a rule named '{2}'.
The current role instance count is {3}.",
  request.RoleName,
  request.InstanceCount,
  context.RuleName,
  context.CurrentInstanceCount);
```

Logging: Semantic Logging mit Serilog

- “semantic logging means strongly typed events”

```
MyCompanyEventSource.Log.ScalingRequestSubmitted(
  request.RoleName,
  request.InstanceCount,
  context.RuleName,
  context.CurrentInstanceCount);
```

Logging: Semantic Logging mit Serilog

- Vorteile
  - Konsistenter Weg um e.g. ein `CustomerChanged` Event zu loggen
  - Einfacher für Queries (string search vs. search nach `CustomerChanged`)
  - Einfacher Daten zu korrelieren
- Eine Library in .NET Serilog

FH University of Applied Sciences TECHNIKUM WIEN

Ausnahme-Fehlerbehandlung in Komponenten

Warum das Thema „Exceptions“?

- Richtig mit Fehlern („Ausnahmezuständen“) umgehen
  - Übergang zum Thema „Fault-Tolerance“
- Software „robust“ und transparent bauen → lange Debugging Sessions vermeiden
  - e.g. Fehler hat sich durch alle Komponenten / Services „propagiert“ → erst Kunde meldet den Fehler
  - Fehler nicht mehr nachvollziehbar („Ich kann mir den Fehler in der Komponenten einfach nicht erklären“)
  - Vertrauen in das System sinkt (e.g. durch Unzuverlässigkeit)
  - Kein Konzept bedeutet viel Arbeit für nachkommende Entwickler …
  - uvm.

Exceptions: Was ist eine Exception?

- … a data structure storing information about an **exceptional condition**
- … useful way to signal that a routine **could not execute normally**
- "When implementing your own methods, you should throw an exception when the **method cannot complete its task as indicated by its name**." [CLR via C#]

https://en.wikipedia.org/wiki/Exception_handling#In_software

Welche Exceptions können wir behandeln?
Welche nicht? Was bedeuten sie?

Typen von Exceptions (nach Lippert)

- Für eine bessere Unterscheidung:
  - Fatal exceptions
  - Boneheaded exceptions
  - Vexing exceptions

https://blogs.msdn.microsoft.com/ericlippert/2008/09/10/vexing-exceptions/

Fatal exceptions

- Fatal exceptions **are not your fault**, you **cannot prevent them**, and you **cannot sensibly clean up from them**.
- Out of memory, thread aborted, and so on.
- There is absolutely **no point in catching** these because nothing your puny user code can do will fix the problem. Just let your "finally“ blocks run and hope for the best

Boneheaded Exceptions

- Boneheaded exceptions **are your own darn fault**, you **could have prevented** them and therefore they are **bugs** in your code.
- You **should not catch them**; doing so is **hiding a bug** in your code.
- That argument is null, that typecast is bad, that index is out of range, you're trying to divide by zero

Vexing exceptions

- Vexing exceptions are the result of unfortunate design decisions.
- Vexing exceptions **are thrown in a completely non-exceptional circumstance**, and therefore must be caught and handled all the time.
- • E.g. Int32.Parse → TryParse
- Anmerkung: Sehr oft anzutreffen in der Praxis – viele Entwickler lieben es, dass die Exception nach oben “bubbeln”, da sie sich dann wenig Gedanken über das Software-Design machen müssen

Exogenous Exceptions

- Exogenous exceptions appear to be somewhat like vexing exceptions except that they **are not the result of unfortunate design choices.**
- Beispiel: Man kann mit `File.Exists` prüfen, ob die Datei existiert. Die Racecondition zwischen `Exists` und `OpenFile` wird aber immer bleiben

```
try
{
  using ( File f = OpenFile(filename, ForReading) )
  {
    // Blah blah blah
  }
}
catch (FileNotFoundException)
{
  // Handle filename not found
}
```

Exkurs: Wenn Exceptions nicht immer Sinn machen – was gibt es noch?

Exkurs: Excpetions vs. Return-Codes

```
if(doSomething())
{
   if(doSomethingElse())
   {
      if(doSomethingElseAgain())
      {
          // etc.
      }
      else
      {
         // react to failure of doSomethingElseAgain
      }
   }
   else
   {
      // react to failure of doSomethingElse
   }
}
else
{
   // react to failure of doSomething
}
```

```
try
{
   doSomething() ;
   doSomethingElse() ;
   doSomethingElseAgain() ;
}
catch(const SomethingException & e)
{
   // react to failure of doSomething
}
catch(const SomethingElseException & e)
{
   // react to failure of doSomethingElse
}
catch(const SomethingElseAgainException & e)
{
   // react to failure of doSomethingElseAgain
}
```

Quelle: Stackoverflow

Exkurs: Excpetions vs. Return-Codes

- Bei constructors, operator overloads, und properties hat der Entwickler keine andere Wahl als Exceptions
- *"Exceptions integrate well with object-oriented languages. … For example, in the case of* ***constructors, operator overloads, and properties, the developer has no choice in the return value****. For this reason, it is not possible to standardize on return-value-based error reporting for object-oriented frameworks. An error reporting method, such as exceptions, which is out of band of the method signature is the only option.“* [Framework Design Guidelines]

Fazit Exceptions

- Man sollte sich im Vorfeld damit beschäftigen …
  - … wie ich mit Exceptions umgehe (Logging ist schön – aber wer liest die Logs?)
  - … welchen Einfluss sie auf das System haben können (was bedeutet ein non-recoverable State? Kann ich die Komponente alleine „retten“? Oder sind mehrere Komponenten betroffen?)
  - … Entwicklern „Defensive programming“ eintrichtern
- Viele Aufgaben für den Architekten …
- Kleine Hilfestellung: Fehlerbehandlung und Business-Logik sollte nicht unbedingt in einer Komponente passieren → **dedizierte Komponenten**, welche andere überwachen und im Fehlerfall agieren

Fazit Exceptions cont.

- Business Anforderungen lassen sich in vielen Situationen mit Return-Codes besser abbilden („Wenn X dann Y“ → Intention im Code besser lesbar)
- Wann Exception fangen? “However, the problem with catching System.Exception and allowing the application to continue running is that state may be corrupted”
  - … dass ich keinen Bug “hide”
  - … dass ich den State in einen konsistenten Zustand bringen kann
  - … dass ich den Benutzer meiner API nicht zwinge die Exception zu fangen → „Flow Control Smell“
  - … dass es nicht anders geht (siehe Exogenous Exceptions)
- Logging / Monitoring kommt noch im „Maintenance“ SDLC-Phase
- Docker etc. spielen bei diesem Thema auch mit …

Tools für Exceptions: Polly

- Polly is a .NET resilience and transient-fault-handling library that allows developers to express policies such as Retry, Circuit Breaker, Timeout, Bulkhead Isolation, and Fallback in a fluent and thread-safe manner. From version 6.0.1, Polly targets .NET Standard 1.1 and 2.0+.
- Nimmt sehr viel Arbeit rund um Exception Handling ab (Erinnerung: wir fangen Excpetions **nur**, wenn wir den State 100% recovern können!)

Tools für Exceptions: Polly - Retry

FH University of Applied Sciences TECHNIKUM WIEN

```
// Retry multiple times, calling an action on each retry
// with the current exception and retry count
Policy
    .Handle<SomeExceptionType>()
    .Retry(3, onRetry: (exception, retryCount) =>
    {
        // do something
    });
```

```
// Retry a specified number of times, using a function to
// calculate the duration to wait between retries based on
// the current retry attempt (allows for exponential backoff)
// In this case will wait for
//  2 ^ 1 = 2 seconds then
//  2 ^ 2 = 4 seconds then
//  2 ^ 3 = 8 seconds then
//  2 ^ 4 = 16 seconds then
//  2 ^ 5 = 32 seconds
Policy
  .Handle<SomeExceptionType>()
  .WaitAndRetry(5, retryAttempt =>
        TimeSpan.FromSeconds(Math.Pow(2, retryAttempt))
  );
```

Tools für Exceptions: Polly - Circuit Breaker

```
// Break the circuit after the specified number of consecutive exceptions
// and keep circuit broken for the specified duration.
Policy
    .Handle<SomeExceptionType>()
    .CircuitBreaker(2, TimeSpan.FromMinutes(1));
```

```
// Break the circuit after the specified number of consecutive exceptions
// and keep circuit broken for the specified duration,
// calling an action on change of circuit state.
Action<Exception, TimeSpan> onBreak = (exception, timespan) => { ... };
Action onReset = () => { ... };
CircuitBreakerPolicy breaker = Policy
    .Handle<SomeExceptionType>()
    .CircuitBreaker(2, TimeSpan.FromMinutes(1), onBreak, onReset);
```

“What's worse if you have many callers on a unresponsive supplier, then you can run out of critical resources leading to cascading failures across multiple systems. ”
https://martinfowler.com/bliki/CircuitBreaker.html

Tools für Exceptions: Polly - Fallback

FH University of Applied Sciences TECHNIKUM WIEN

```
// Provide a substitute value, if an execution faults.
Policy<UserAvatar>
    .Handle<FooException>()
    .OrResult(null)
    .Fallback<UserAvatar>(UserAvatar.Blank)
```

Wie können Komponenten das Maximum aus der Umgebung herausholen?

Nebenläufigkeit

- Sollte ein Architekt auch immer im Blick haben → sehr oft essentiell zum Einhalten der nicht-funktionalen Anforderungen
- Kann auch auf Design Ebene betrachtet werden (siehe Map-Reduce e.g.)
- Fokus hier: Komponente intern

Warum das Thema?

- Entwickler bekommen Aufaben
- Ausgangsszenario: 5 Jahre alter Legacy Code:
  - `new Thread` an vielen Codestellen
  - der Entwickler davor hat schon mal einen `Mutex` und einen `Semaphore` eingebaut um sich zu behelfen
- Wie hoch ist die Wahrscheinlichkeit, dass neue Aufaben es besser machen?
- Siehe auch Clean-Code (https://de.wikipedia.org/wiki/Broken-Windows-Theorie)
- Daher: Von Anfang an ein Konzept vorgeben, wie mit parallelen Code umzugehen ist
  - Es muss nicht immer der „neuste“ Ansatz sein → aber sollte sich konsistent in der Komponente durchziehen

Wo ist der Herausforderung?

- Moderne CPUs haben viele Kerne
- Jeder Core hat (je nach Architektur) eigene
  - ALU (Arithmetisch-logische Einheit)
  - Register
  - L1 Cache
  - Fetch Decode Unit
- Anzahl der Kerne?
  - Bei Spielekonsole: bekannt
  - Bei „Standard“ Software: **unbekannt** ☹
- Anmerkung: Auch eine Aufgabe des Architekten: Skaliert mein Zielsystem? Viele Kunden bieten Softwarelieferanten oft nur virtuelle Maschinen an – aber **ohne SLA** …

Exkurs: Concurrency vs. Paralellism

- **Concurrency**: mehrere Threads welche Zeitscheiben (Times Slices) bekommen → geht auf Single Core Maschine
- **Parallelism**: Threads werden zur selben Zeit ausgeführt
- Trotzdem gemeinsame Herausforderungen:
  - Deadlocks
  - Raceconditions
  - …
- Vermeiden Sie um jeden Preis diese „Herausforderungen“!

Fazit

- Der Architekt sollte sich im Vorfeld damit auseinander setzen
  - Zielsystem
  - Paralellism in der Komponenten / zwischen Komponenten (wenn Aufgaben parallel abgearbeitet werden)
- Gut dokumentieren und die Intention im Code klar ausdrücken (damit es andere Entwickler 100% verstehen und das Konzept fortsetzen können)
- Und: Ein Entwickler sollte **immer** wissen, wie es um den Paralellism steht, wenn ich auf eine Codestelle oder eine Komponente zeige → im Vorfeld planen!

Tools für Threading: PostSharp

```
[Actor]
class AverageCalculator
{
    float sum;
    int count;

    [Reentrant]
    public async Task AddSample(float n)
    {
        this.count++;
        this.sum += n;
    }

    [Reentrant]
    public async Task<float> GetAverage()
    {
        return this.sum / this.count;
    }
}
```

- Diese Implementierung ist Thread-Safe!
- Reentrant: means that other methods can be invoked during awaiting

ProtoActor .NET

- Actor Konzept in Erlang populär geworden
- In Java: AKKA (bzw. AKKA.NET → momentan mehr Entwicklungsaktivität als protoactor?!)
- Alternative: https://github.com/AsynkronIT/protoactor-dotnet (selbes Konzept)
- Frameworks helfen:
  - Concurrency / Paralleism mit einem Konzept zu handeln
  - Fehlerbehandlung strukturiert durchzuführen
  - Aber auch einige Helferlein für Distributed-Challenges (out-of-topic)
- Es handelt sich um einen Message-Based-Approach
  - Messages sollten immer immutable sein (https://en.wikipedia.org/wiki/Immutable_object#C#)

ProtoActor .NET

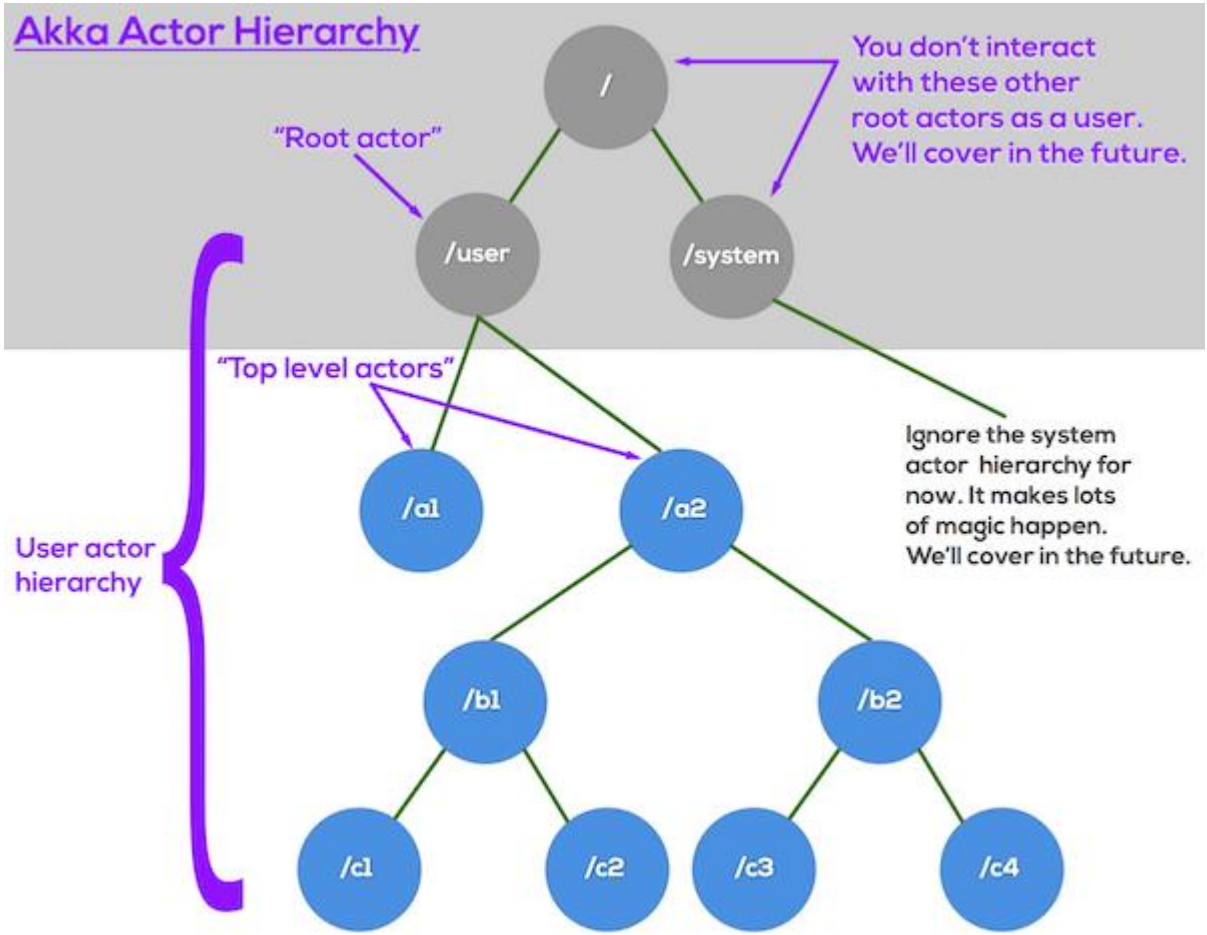
```
internal class HelloActor : IActor
{
    public Task ReceiveAsync(IContext context)
    {
        var msg = context.Message;
        if (msg is Hello r)
        {
            Console.WriteLine($"Hello {r.Who}");
        }
        return Actor.Done;
    }
}
```

```
var context = new RootContext();
var props = Props.FromProducer(() => new HelloActor());
var pid = context.Spawn(props);

context.Send(pid, new Hello
{
    Who = "Alex"
});
```

ProtoActor .NET Exceptions

- Supervisor Hierachy (out-of-scope)
- Was mitnehmen:
  - Trennen von Logik und Fehlerbehandlung
  - /b1 tut sich leichter Rückschlüsse zu ziehen, weil es mehr Context als /c1 hat (geht /c2 noch oder muss ich alles neu starten?)
- https://github.com/AsynkronIT/protoactor-dotnet/blob/dev/examples/Supervision/Program.cs

ProtoActor .NET Exceptions



FH University of Applied Sciences TECHNIKUM WIEN

Komponentenzuverlässigkeit

Themen rund um nicht-funktionale Anforderungen

Software-Aging

- Auch Software / Komponenten altern …
- Wenn Software läuft …
  - … kann es zu Memory-Leaks kommen
  - … Threads können hängen bleiben (Deadlocks, …)
  - … Festplatten können volllaufen
  - … Race-Conditions können Software in falschen Zustand bringen
- Helfen Langzeit / Performancetests immer?
  - Eher nein: sie geben eher Tendenzen an – e.g. +10% mehr Last oder längerer Test könnten u.U. die Komponenten in einen Schiefstand bringen

Software-Aging

- Daher: Rejuvenation
- Proaktive Maßnahme
  - Ressourcen freigeben
  - Interne Daten neu initialisieren
  - Reboot jeden Tag um 02:00
- Daher: Energie in Failover investieren (man braucht es meist so oder so → nicht funktionale Anforderungen)

https://en.wikipedia.org/wiki/Software_aging

Weiter führende Themen zu Software Aging

- https://netflix.github.io/chaosmonkey/
- https://www.golem.de/news/stresstest-netflix-chaos-monkey-zerstoert-effektiver-infrastruktur-1610-123938.html
- Units of Mitigation Pattern

Exkurs: Sicherheit vs. Zuverlässigkeit

- Oft fälschliche Annahme, dass Sicherheit (Safety) durch Zuverlässigkeit (Reliability) erhöht werden kann
- So kann ein System e.g. zuverlässig, aber unsicher sein. D.h., dass die Komponenten nicht versagt, aber trotzdem unsicher gehandelt haben.
- Beispiel Mars Polar Lander: Ruck am Landesensor → frühzeitigen Abschalten der Bremstriebwerke → Absturz
- Alle Komponenten haben funktioniert → aber unsicher
- **Component Interaction Accident**
  - durch die immer steigende Systemkomplexität ein steigendes Interesse an diesem Thema

Nicht-funktionalen Anforderungen für Zuverlässige Software

- **Coverage** gibt die Wahrscheinlichkeit an, welche beschreibt, dass sich ein System im Fehlerfall in einem gegebenen Zeitintervall, automatisch rettet (engl. recover).
- **Reability**: Wahrscheinlichkeit, dass ein System ohne Fehler in einem gewissen Zeitintervall arbeitet. Ein Attribut ist e.g. Mean Time To Failure (MTTF).
- **Availability** beschreibt in Prozent jene Zahl, welche die Zeit wiederspiegelt, in der das System laut Spezifikation funktioniert.


APACHE kafka
A distributed streaming platform
GET STARTED
DOCS
Fault Tolerance
Kafka Streams builds on fault-tolerance
partitions are highly available and replic
redis.io/topics/cluster-spec
The readonly state of the connection can be cleared
Fault Tolerance
Heartbeat and gossip messages
Redis Cluster nodes continuously exchange ping an
structure, and both carry important configuration in

Definition: Fault Tolerance

- Von Software Fault Tolerance spricht man, wenn Software weiter funktioniert – auch wenn Teile von ihr nicht mehr korrekt funktionieren
- drei Begriffe: **Fault**, **Error** und **Failure**
- Wenn Spezifikation (e.g. 99,999% Verfügbarkeit) nicht eingehalten → **Failed (Failure)**
- Ein Failure wird von einem **Error** (e.g. Race conditions, Endlosschleifen, Protokollfehler, …) hervorgerufen
  - sind wichtig, da sie erkannt werden können, bevor sie zu einem Failure führen
- **Fault** (Umgangssprachlich Bug) ist ein Defekt in der e.g. Software (e.g. falsch gesetzte Klammern bis hin zu falschen Schleifenabbruchbedingungen)

Warum das Thema? Cont. Phase III

- Wir werden es schwer schaffen, fehlerlose Komponenten zu bauen
- Aber: durch ein paar Strategien können wir lernen, mit Fehlern gut umzugehen
- Durch Sensibilisierung von Entwicklern im Team kann man viele Probleme präventiv vermeiden
- Welche **Fehlertypen** gibt es? Viele – ein paar gängige:
  - Integration Points
  - Chain Reactions
  - Attacks of Self-Denial
  - Slow Responses
  - Unbounded Result Sets

Integration Points

- Komponenten integrieren gegen Datenbanken, kommunizieren übers Netzwerk
- Beispiel: Datenbank „dead-connections“ – braucht Connection-Pool auf → passende Detektion erforderlich
- Netzwerk: TCP vs. SCTP Thema half-opened-connections
  - https://www.ibm.com/developerworks/library/l-sctp/
- Fazit: Jeder Integration-Point kann versagen → man sollte drauf vorbereitet sein
- Hilfemittel:
  - Decoupling Middleware (mehr dazu später)
  - Circuit Breaker

Chain Reactions

- Load-Balancer → Server fällt aus → restliche Last auf andere Server → diese sterben auch
- Wahrscheinlichkeit wenn ein Server einen Bug hat (Memory-Leak, Race-Condition, Dead-Lock, …) werden es auch die anderen haben
- Hilfesmittel:
  - Bulkheads

Attacks of Self-Denial

- Immer, wenn Menschen Teil des Systems sind
- Amazon 2006 Angebot 1000 XBOXes für $100
  - Viele Leute haben Webseite refreshed
  - 1000 Konsolen wurden in 5 Minuten verkauft
  - In dieser Zeit wurde sonst nichts verkauft → Serverauslastung

Slow Responses

- Es ist besser, wenn die eine Komponente auf die Anfrage einer anderen Komponenten mit einem Fehler antwortet, als diese warten zu lassen (Ressourcen …)
- User fangen an den Reload Button zu drücken …
- Hilfsmittel:
  - Fail Fast

Unbounded Result Sets

- Komponenten mit realistischen Datenmengen entwerfen und testen
  - dafür brauchen wir natürlich gute nicht funktionale Anforderungen
- Hilfsmittel:
  - Paging (SQL Server OFFSET / FETCH)
  - Limits immer einbauen (SQL TOP usw.)
    - Aber auch bei JSON, GRPC oder sonstigen Protokollen

FH University of Applied Sciences TECHNIKUM WIEN

Was können wir dagegen tun?

Patterns für Komponenten-Stabilität

- Es gibt einige Patterns zur Verbesserung der Stabilität von Komponenten
- Vor allem aber reduzieren sie das Kopfweh und den Aufwand der Fehlersuche
- Es gibt viele – einige interessante:
  - Timeouts
  - Circuit Breaker
  - Bulkheads
  - Steady State
  - Fail Fast
  - Maintenance Interface

Timeouts

- Vor allem für Remote-Calls (von einer anderen Komponenten abhängig)
- Timeouts aber bei erneuten Versuchen – es macht keinen Sinn, die andere Komponenten mit Anfragen zu bombardieren (eventuell auch randomisierte Timeouts, damit nicht alle gleichzeitig wieder probieren)
- Wir erinnern uns an den Connector zurück
  - Dort befindet sich die Logik um mit einer anderen Komponenten zu kommunizieren → normalen „Business-Code“ freihalten von Kommunikations-Timeouts

Timeouts in C# mit async

C#

```
static async Task Main()
{
    Console.WriteLine("Application started.");

    try
    {
        s_cts.CancelAfter(3500);

        await SumPageSizesAsync();
    }
    catch (TaskCanceledException)
    {
        Console.WriteLine("\nTasks cancelled: timed out.\n");
    }
    finally
    {
        s_cts.Dispose();
    }

    Console.WriteLine("Application ending.");
}
```

https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/async/cancel-async-tasks-after-a-period-of-time

Circuit Breaker

- Wenn die Nummer der Fehler einen Treshold überschreiten → fliegt die Sicherung
- Ist der Circuit Breaker offen, schlagen Aufrufe sofort fehl (ohne die andere Komponenten noch zu belasten)
- Implementierungsdetail: half-open: Es wird nach einer gewissen Zeit nochmals probiert – wenn es wieder fehschlägt, wird er sofort wieder geschlossen
- „When there's a difficulty with Integration Points, stop calling it!"
- Am Besten zusammen mit Timouts nutzen
- Wichtig: Wenn ein Circuit Breaker öffnet → sollte es irgendwo aufscheinen → kritische Situation!

Circuit Breaker

```
CircuitBreakerPolicy breaker = Policy
  .Handle<HttpRequestException>()
  .CircuitBreaker(
    exceptionsAllowedBeforeBreaking: 2,
    durationOfBreak: TimeSpan.FromMinutes(1)
  );
```

https://github.com/App-vNext/Polly/wiki/Circuit-Breaker

Bulkheads

- Im Schiffbau: Unterteilung des Schiffs in mehrere „Compartments“
- „Save part of the ship”: Teilfunktionalität sicherstellen und nicht das ganze System “sterben” lassen
- Partitionierung ist oft die Challenge: Wie teile ich mein System in sinnvolle „Compartments“ ein?
  - Ganze Server?
  - Memory?
  - CPU?

Bulkheads

```
// Restrict executions through the policy to a maximum of twelve concurrent actions.
Policy
  .Bulkhead(12)

// Restrict executions through the policy to a maximum of twelve concurrent actions,
// with up to two actions waiting for an execution slot in the bulkhead if all slots are taken.
Policy
  .Bulkhead(12, 2)
```

- Achtung: Auch auf Architekturebene wichtig
- https://www.haproxy.com/de/blog/four-examples-of-haproxy-rate-limiting/
- https://docs.microsoft.com/en-us/azure/architecture/patterns/bulkhead

https://github.com/App-vNext/Polly/wiki/Bulkhead

Steady State

- Wenn Menschen am Server „herumschrauben“ (e.g. Konfiguration im OS) → eine Wahrscheinlichkeit für Fehler entstehen
- D.h. menschliche Interaktionen sollten vermieden werden → Automatisieren was geht
- Database Maintaince, Memory Cache limitieren (TTL, Total Limit)
- Roll Logs: Nicht loggen bis die Festplatte voll ist …

Fail Fast

- Das schlimmste was passieren kann, wenn ich eine andere Komponenten etwas frage:
  - eine langsame Antwort
  - und dann auch noch falsch!
- Wenn ich die Anfrage nicht erfüllen kann → sofort rückmelden
- Wenn Ressourcen knapp sind und es ungewiss ist, ob ich die Anfrage erledigen kann → lieber gleich Fehler (e.g. Circuit Breaker vom Komponenten B zur Datenbank ist schon offen → frühestmöglich Komponenten A sagen)
- E.g. HTTP Status Code 503 - Service Unavailable
- Siehe auch Thema Back Pressure

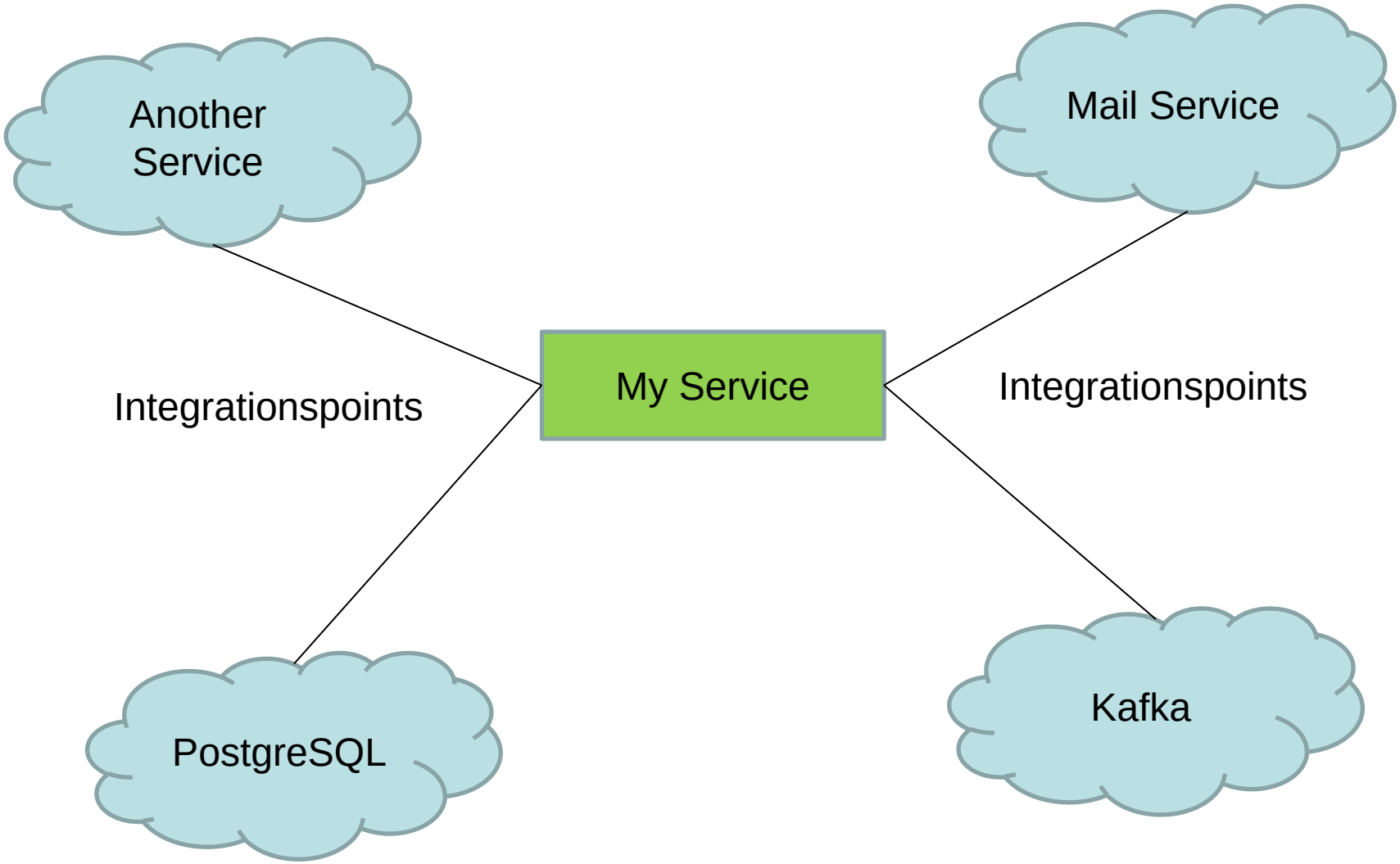
Maintenance Interface

- Software braucht Wartung – vor allem, wenn es mal ein Problem gibt
- Wichtig: Immer genug Luft lassen, damit Wartungs-Request durch gehen (e.g. Connection-Limit ändern, Cache leeren, Komponenten neustarten, Logs / Metriken lesen)
- Gibt nichts schlimmeres, wenn ein ausgelastetes System nicht mehr für Maintenance responsive ist

FH University of Applied Sciences TECHNIKUM WIEN

Komponenten / Services überwachen und verstehen

Example


Another Service
Mail Service
Integrationspoints
My Service
Integrationspoints
PostgreSQL
Kafka

Wo kracht bei Services sehr oft?

- Bei Integrationspunkten!
  - Datenbank, Message Broker, andere Service, …
- Es ist daher wichtig folgende 3 Punkte automatisiert zu überwachen:
  - Verstehe ich, was die andere Komponente sagt? E.g. Fehler durch Protokoll (e.g. Versionskonflikt)
  - Antwortet die andere Komponente in einem von mir erwarteten Zeitrahmen? (e.g. Expectations festlegen → Timouts, Integrationspunkt ausgelastet → Backpressure)
  - Last but not least: ist der Integrationspunkt überhaupt erreichbar? (e.g. down, Firewall Fehlkonfiguration)

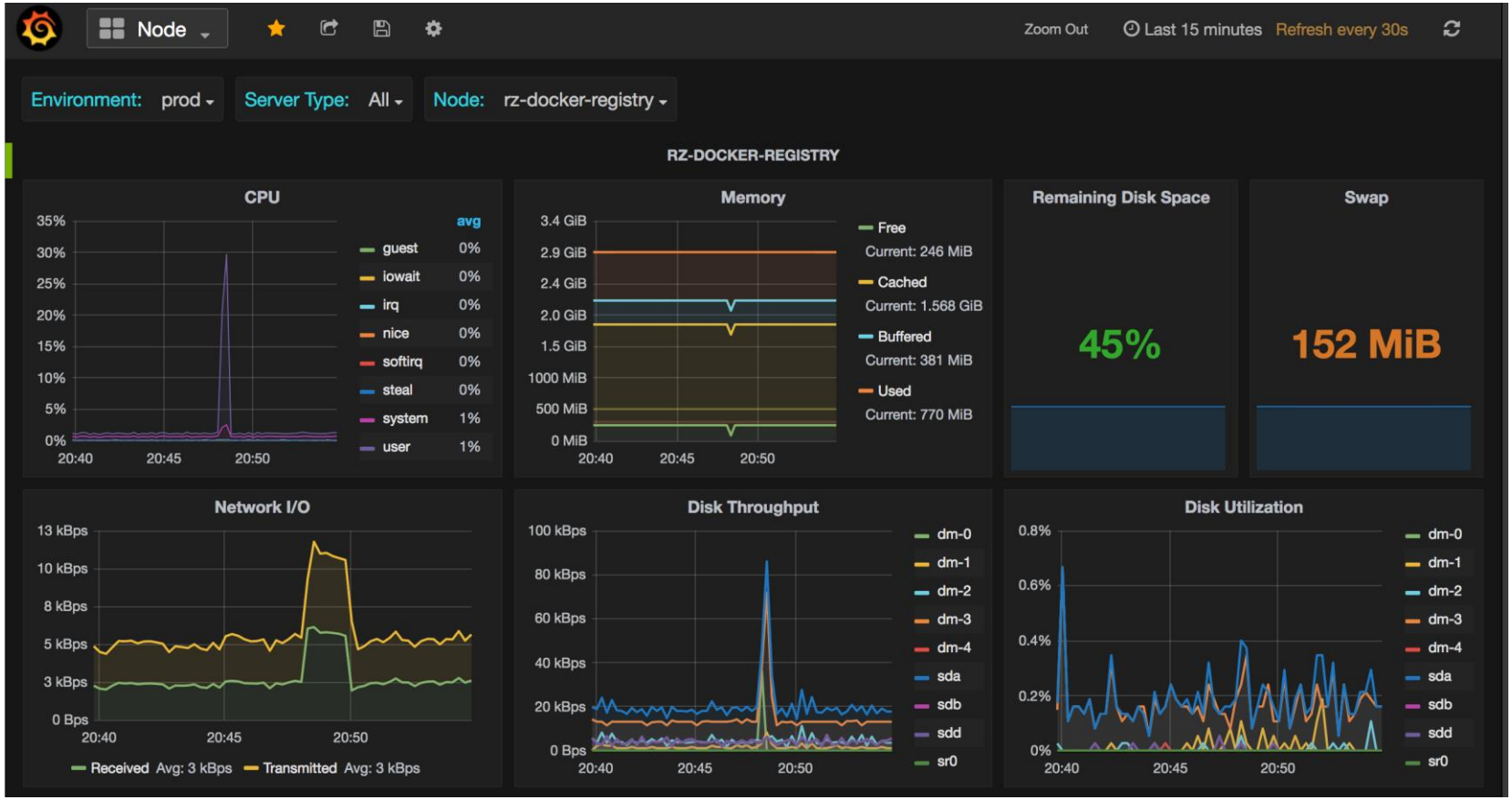
Mögliche Lösung: Health Check API

- https://microservices.io/patterns/observability/health-check-api.html
- `/health` gibt den Status des Integrationspunkte zurück
- Was das Leben eher schwer macht:
  - 103 Mal in die LogFile „DatabaseConnectionException“ schreiben …
  - Wie zuverlässig interpretieren? Hilft maximal beim Debuggen
- Was schon hilfreich sein kann: Metriken
  - Mehr dazu jetzt

Service Metriken: Prometheus

- Zum Event-Monitoring (e.g. Request / second) und zum erstellen von Alarmen (e.g. zu viele Requests)
- Optimiert für Time-Series (Datenbank speziell für Events mit Zeitstempeln)
- PromQL (Prometheus Query Language) zum Abfragen von Zeitdaten. Auch interessant: Operatoren zum aggregieren
- Daten werden gepollt (Exports) – d.h. Prometheus holt sich aktiv Daten von Endpunkten, welche es überwachen soll: https://prometheus.io/docs/instrumenting/exporters/

Service Metriken: Prometheus

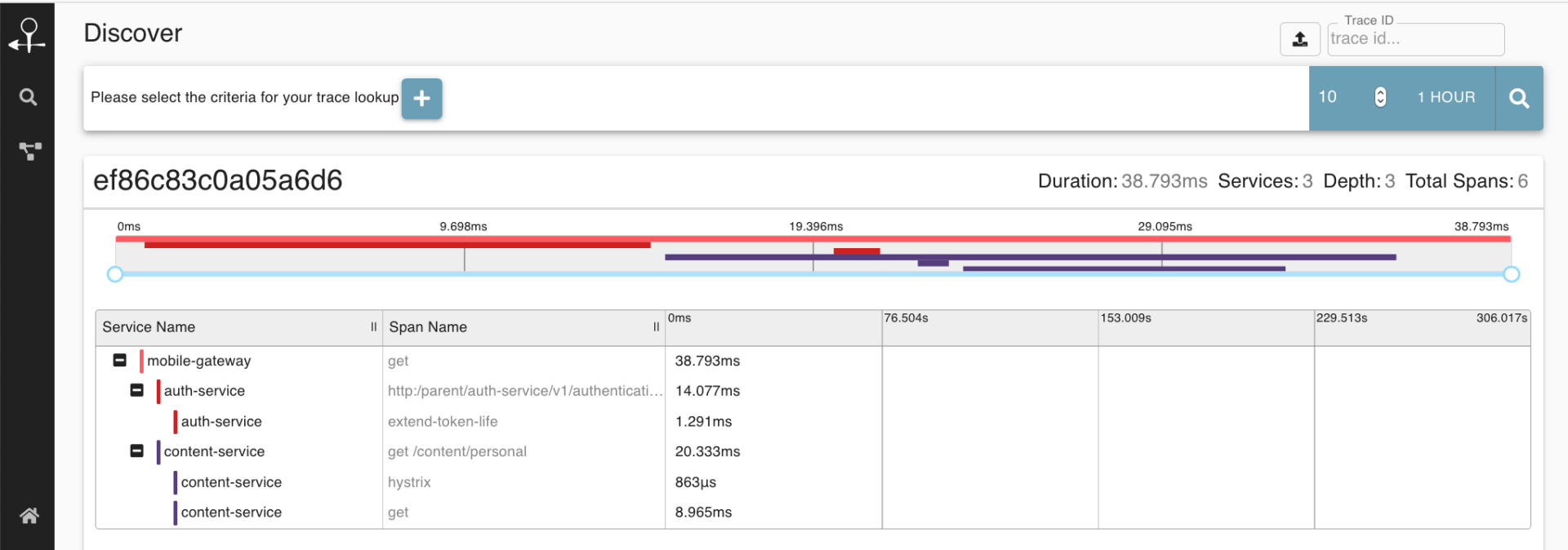


Anwendungsbeispiele

- Ressourcen überwachen (Exhaustion erkennen – e.g. Speicher oder Datenbankverbindungen)
- Anzahl der Reqeusts / Sekunde
- Anzahl der Fehler / Sekunde
- uvm.

Distributed Tracing: Zipkin

- Reporter sendet daten zu Zipkin (traceId hat mehrere spanIds)



FH University of Applied Sciences TECHNIKUM WIEN

Daten in Komponenten persistieren

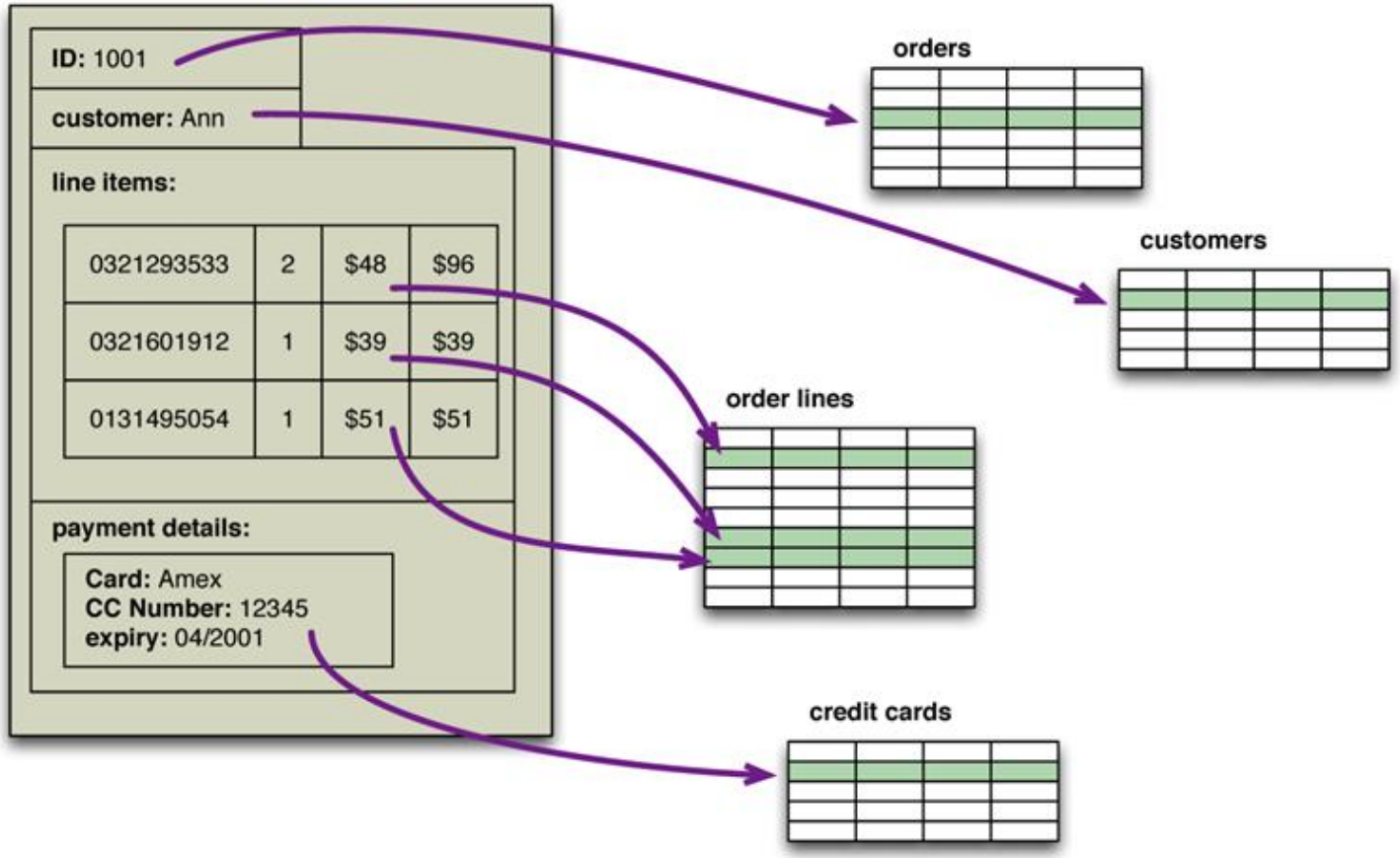
Persistenz …

- … das Cafe-Küchen Thema schlechthin …
- Wir schauen uns zwei Themen an
  - NoSQL
  - RDBMS
- Ziel: Daten persistieren

NoSQL

- Kein Ersatz für RDBMS → NoSQL und RDBMS können nebeneinander existieren … (
  - Sogar innerhalb eines Projekts (https://en.wikipedia.org/wiki/Polyglot_persistence)
- Impedance mismatch: Konflikt zwischen relationalen Datenbankmodel und Objektorientierten In-Memory Datenstruktur
- Use the right tool for the right job!

Impedance mismatch



Sadalage, P. J., & Fowler, M. (2013). NoSQL distilled: a brief guide to the emerging world of polyglot persistence. Pearson Education.

NoSQL

- Begriff wurde 2009 bei einer Konferenz verwendet – rund um die damals bekannten Datenbanken „Google BigTable“ und „Amazon Dynamo“
- Vereinfacht gesagt geht es um **nicht-relationale** Datenbanken
- Haupt Eigenschaften:
  - Kein relationales Model
  - Cluster-orientiert
  - Schemaless
- Es wird meist in Aggregates gedacht (siehe DDD) → zusammengehörige Dinge (e.g. ein Kunde, eine Bestellung)

Beispiel eines Aggregates

```
{
    "customerId": 12,
    "dateOfPurchase": "2020-11-06T14:06:37Z",
    "items": [
        {
            "id": "HSDH23",
            "quantity": 2,
            "singlePrice": 10.00,
            "sum": 20.00
        },
        {
            "id": "KKSDU3",
            "quantity": 1,
            "singlePrice": 19.00,
            "sum": 19.00
        }
    ],
    "paymanetDetails": {
        "type": "PAYPAL"
    }
}
```

- Aggregate-Boundary finden nicht trivial
- Was ist wenn ich die verkaufte Menge von Artikel „HSDH23“ im aktuellen Monat wissen will?
- → man müsste alle Aggregate durchlaufen
- Vorteile von Aggregates: wir wissen, welche Daten miteinander geändert werden (→ vor allem im Clusterbetrieb hilfreich)

NoSQL Ausprägungen

- Key-Value
- Document
- Column-Family
- Graph
- https://hostingdata.co.uk/nosql-database/

Key-Value

- Unterschiedliche Ausprägungen / Features (siehe e.g. https://redis.io/topics/data-types)
- Manche Implementierungen unterstützen Buckets
  - Erlaubt Untergliederung in „Domains“ um Konflikte zu vermeiden
- Bekannte Implementierungen: Redis, Riak

Key-Value: Beispiel Redis

```
7  var redisManager = new RedisManagerPool("localhost:6379");
8  var redis = redisManager.GetClient();
9
10 public class Todo
11 {
12     public long Id { get; set; }
13     public string Content { get; set; }
14     public int Order { get; set; }
15     public bool Done { get; set; }
16 }
17
18 var redisTodos = redis.As<Todo>();
19 var newTodo = new Todo
20 {
21     Id = redisTodos.GetNextSequence(),
22     Content = "Learn Redis",
23     Order = 1,
24 };
25
26 redisTodos.Store(newTodo);
27 Todo savedTodo = redisTodos.GetById(newTodo.Id);
28
29 savedTodo.Done = true;
30 redisTodos.Store(savedTodo);
31
32 var updatedTodo = redisTodos.GetById(newTodo.Id);
```

Document

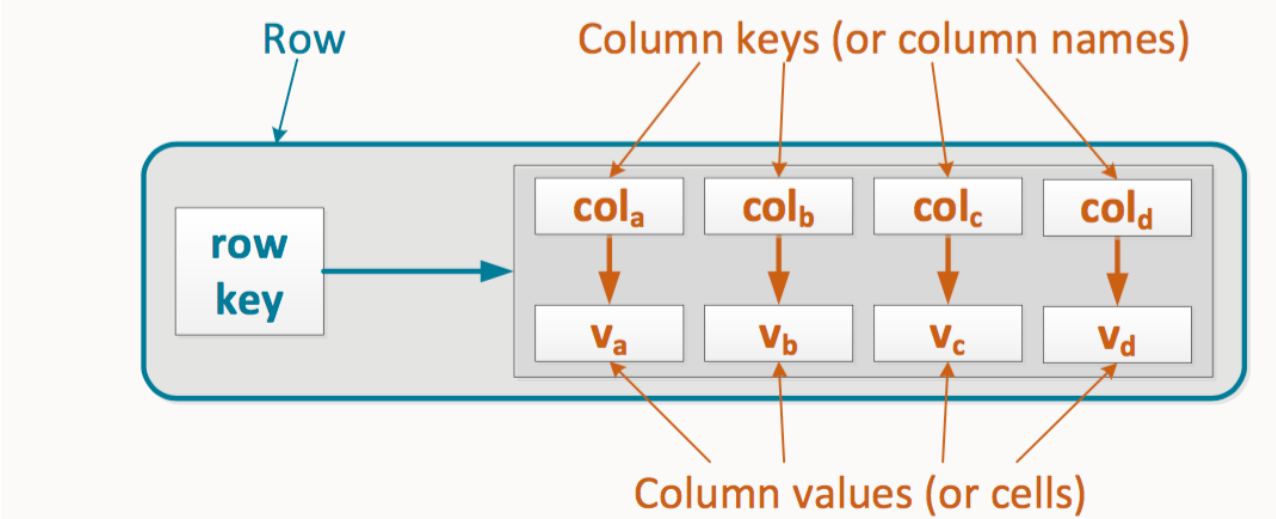
- Beispiel MongoDB
- Meist in XML, JSON, BSON, … (Semistrukturierte / self describing)
- Transaktionen meist nur auf document-level
  - Ausnahmen e.g. ab Version 4.0 MongoDB https://www.mongodb.com/blog/post/mongodb-multi-document-acid-transactions-general-availability
- MongoDB hat collections (ähnlich tables)
- Bekannte Implementierungen: MongoDB, CouchDB

Document: Beispiel MongoDB

```
var connectionString = "mongodb://localhost";
var client = new MongoClient(connectionString);
var server = client.GetServer();
var database = server.GetDatabase("test");
var collection = database.GetCollection<Entity>("entities");

var entity = new Entity { Name = "Tom" };
collection.Insert(entity);
var id = entity.Id;

var query = Query<Entity>.EQ(e => e.Id, id);
entity = collection.FindOne(query);

entity.Name = "Dick";
collection.Save(entity);

var update = Update<Entity>.Set(e => e.Name, "Harry");
collection.Update(query, update);

collection.Remove(query);
```

Column-Family

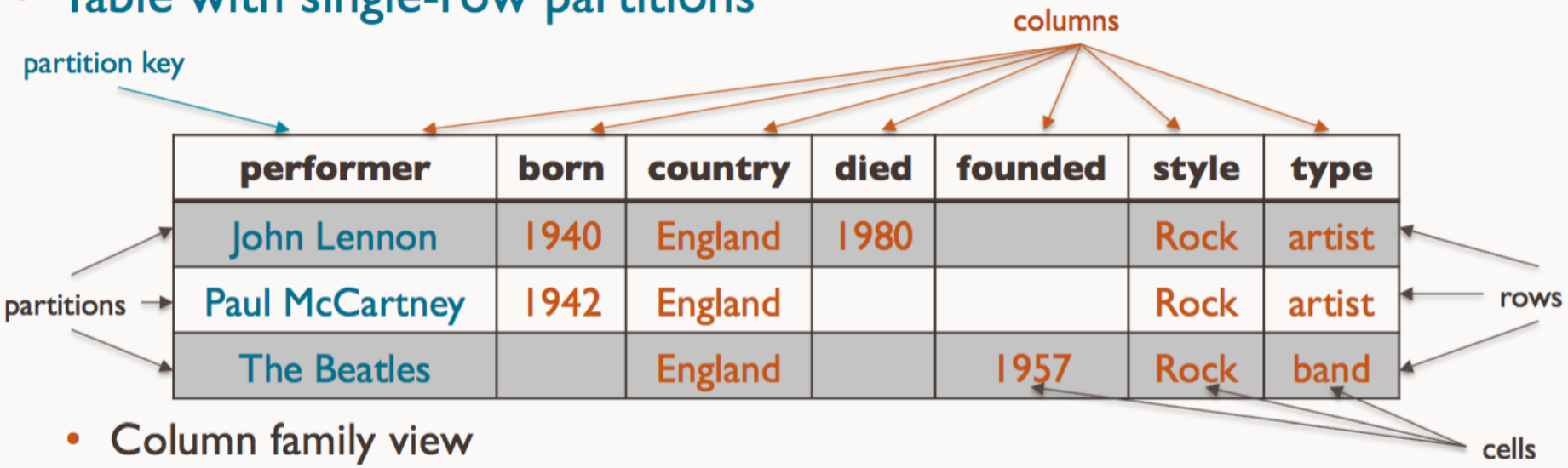
- Column families sind schemaless
  - Daher dynamischer als normale RDBMS rows
- Schnelle beim Schreiben (Distributed, zuerst Memory → dann Disk) → Partition Key
- SQL ähnlich (CQL)
- Bekannte Implementierungen: Cassandra



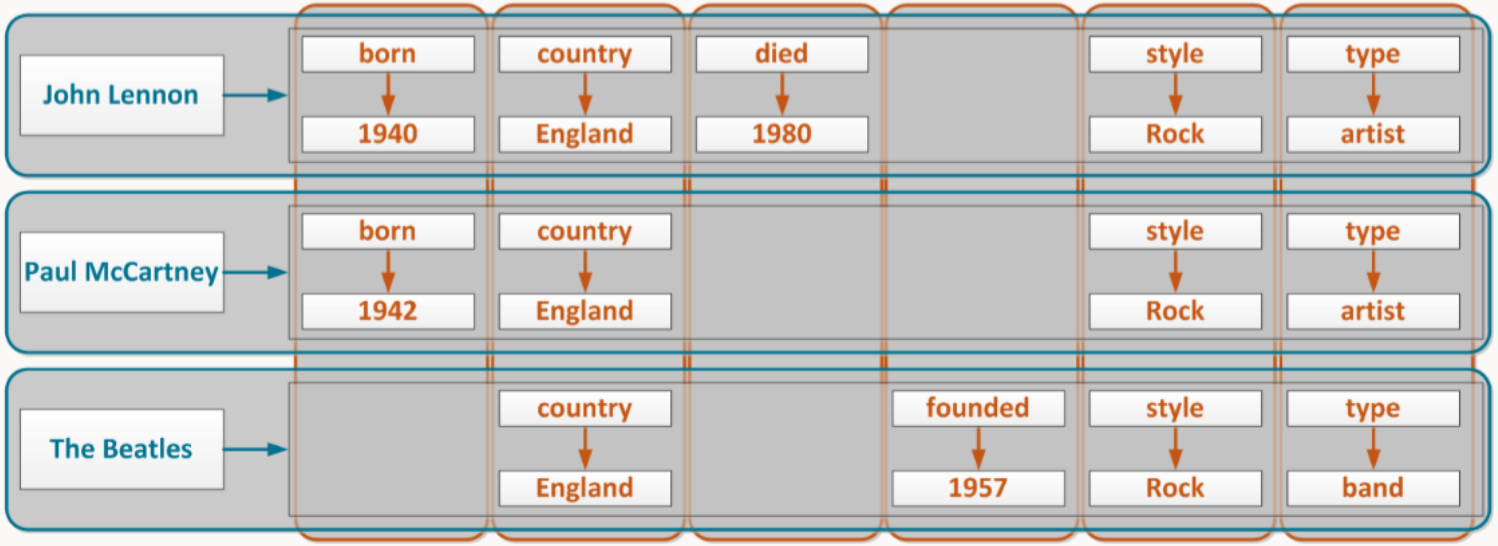
Quelle: https://studio3t.com/

Beispiel column family

- Table with single-row partitions



- Column family view



Quelle: https://studio3t.com/

Achtung Use-Cases

- Nicht trivial …
  - Writes exceed reads by a large margin.
  - Data is rarely updated and when updates are made they are idempotent.
  - Read Access is by a known primary key.
  - Data can be partitioned via a key that allows the database to be spread evenly across multiple nodes.
  - There is no need for joins or aggregates.
- Quelle: https://blog.pythian.com/cassandra-use-cases/

Column-Family: Beispiel Cassandra

FH University of Applied Sciences TECHNIKUM WIEN

You can execute multiple statements (prepared or unprepared) in a batch to update/insert several rows atomically even in different column families.

```
// Prepare the statements involved in a profile update once
var profileStmt = session.Prepare("UPDATE user_profiles SET email=? WHERE key=?");
var userTrackStmt = session.Prepare("INSERT INTO user_track (key, text, date) VALUES (?, ?, ?)");
// ...you should reuse the prepared statement
// Bind the parameters and add the statement to the batch batch
var batch = new BatchStatement()
  .Add(profileStmt.Bind(emailAddress, "hendrix"))
  .Add(userTrackStmt.Bind("hendrix", "You changed your email", DateTime.Now));
// Execute the batch
session.Execute(batch);
```

Quelle: https://github.com/datastax/csharp-driver

RDBMS

- Durch Frameworks versucht man „Gap“ zwischen Objektorientierten Welt und Relationaler Welt zu schließen
- Man unterscheidet zwischen
  - ORM
    - Queries bleiben meist verborgen
    - Man merkt die relationale Datenbank fast gar nicht
  - Micro ORM
    - Low-Level: Queries selber schreiben
    - Hilfe nur beim Mapping relational → objektorieniert

ORM: Entity Framework

```
using (var db = new BloggingContext())
{
    // Create
    Console.WriteLine("Inserting a new blog");
    db.Add(new Blog { Url = "http://blogs.msdn.com/adonet" });
    db.SaveChanges();

    // Read
    Console.WriteLine("Querying for a blog");
    var blog = db.Blogs
        .OrderBy(b => b.BlogId)
        .First();

    // Update
    Console.WriteLine("Updating the blog and adding a post");
    blog.Url = "https://devblogs.microsoft.com/dotnet";
    blog.Posts.Add(
        new Post
        {
            Title = "Hello World",
            Content = "I wrote an app using EF Core!"
        });
    db.SaveChanges();

    // Delete
    Console.WriteLine("Delete the blog");
    db.Remove(blog);
    db.SaveChanges();
}
```

ORM-Micro: Dapper

```
public class Dog
{
    public int? Age { get; set; }
    public Guid Id { get; set; }
    public string Name { get; set; }
    public float? Weight { get; set; }

    public int IgnoredProperty { get { return 1; } }
}

var guid = Guid.NewGuid();
var dog = connection.Query<Dog>("select Age = @Age, Id = @Id", new { Age = (int?)null, Id = guid });
```

```
var count = connection.Execute(@"insert MyTable(colA, colB) values (@a, @b)",
    new[] { new { a=1, b=1 }, new { a=2, b=2 }, new { a=3, b=3 } }
```