



Cloud Computing

Kapitel 1: Kommunikation

Mario-Leander Reimer

mario-leander.reimer@qaware.de

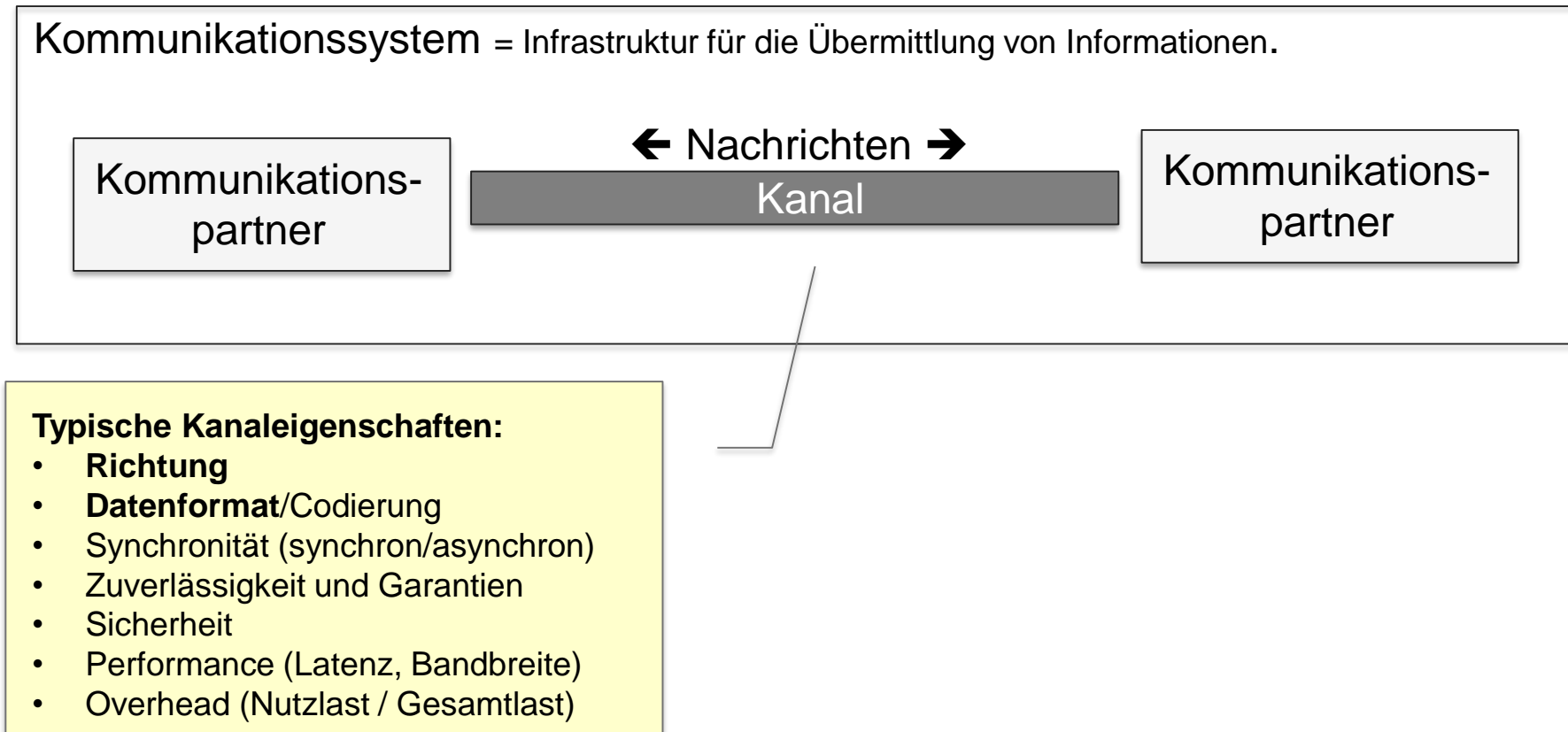
Rosenheim, 16.10.2017



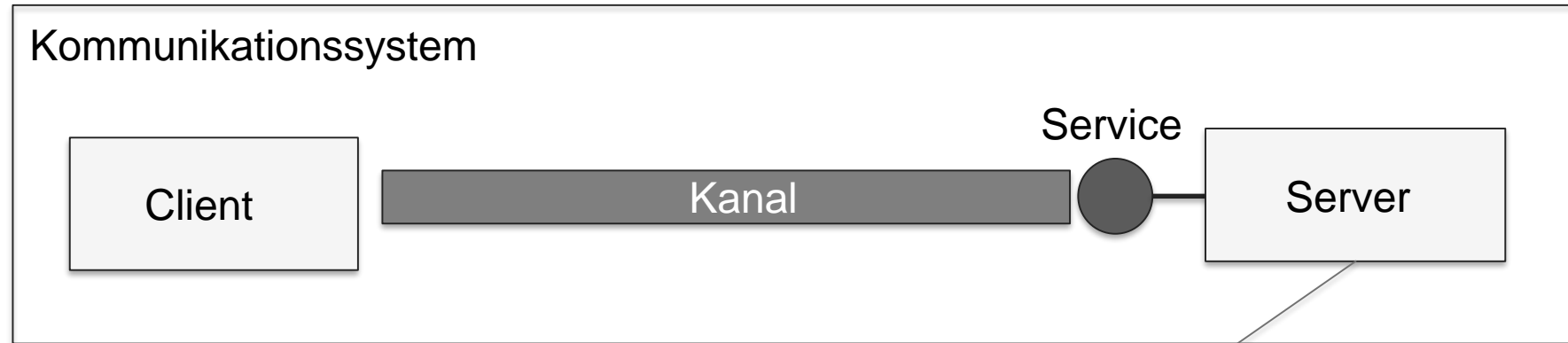
**„Man kann nicht nicht
kommunizieren.“**

Paul Watzlawik

Ein allgemeines Kommunikationsmodell im Internet. Angelehnt an das Modell von Shannon/Weaver.



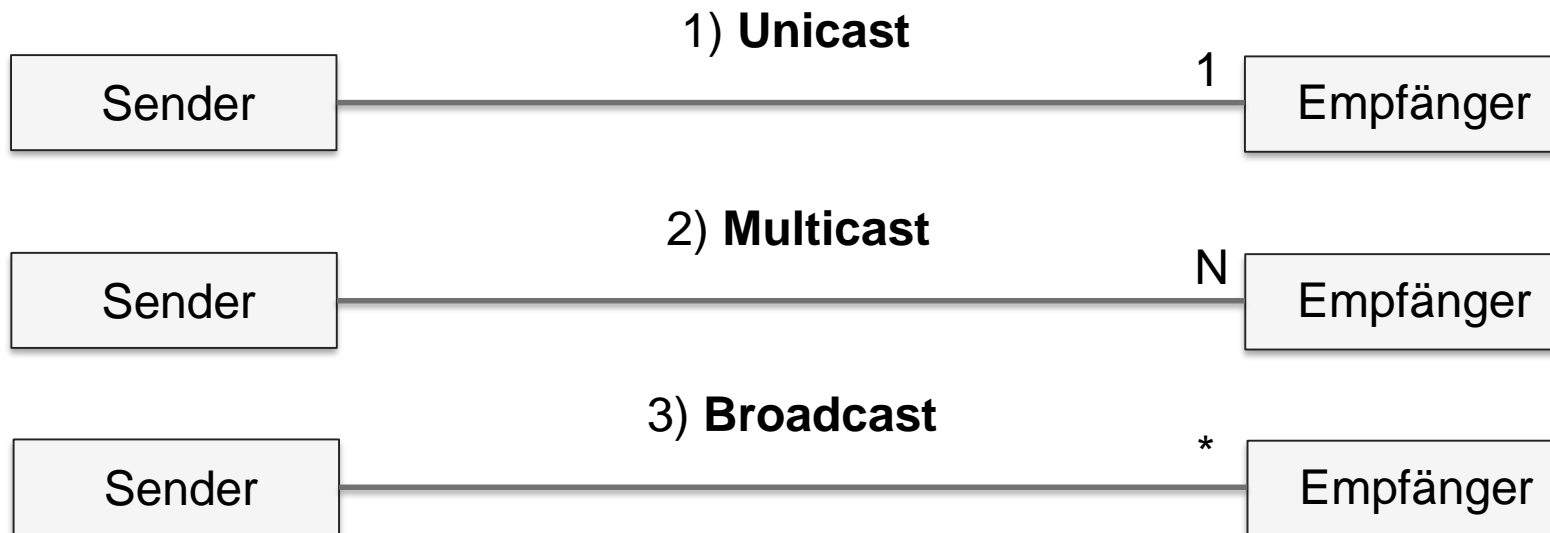
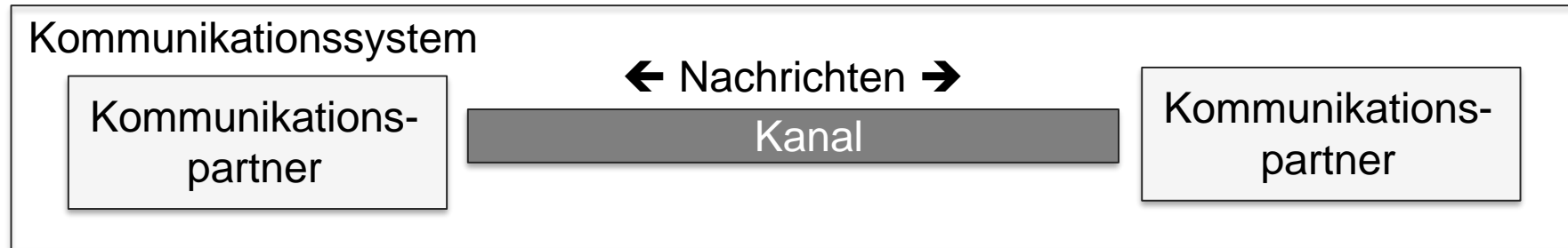
Service-Orientierung in einem Kommunikationssystem.



Ein **Service** ist eine Funktionalität, die über eine definierte Schnittstelle zur Verfügung stellt. Jeder Service ist definiert durch eine **Serviceschnittstelle**.

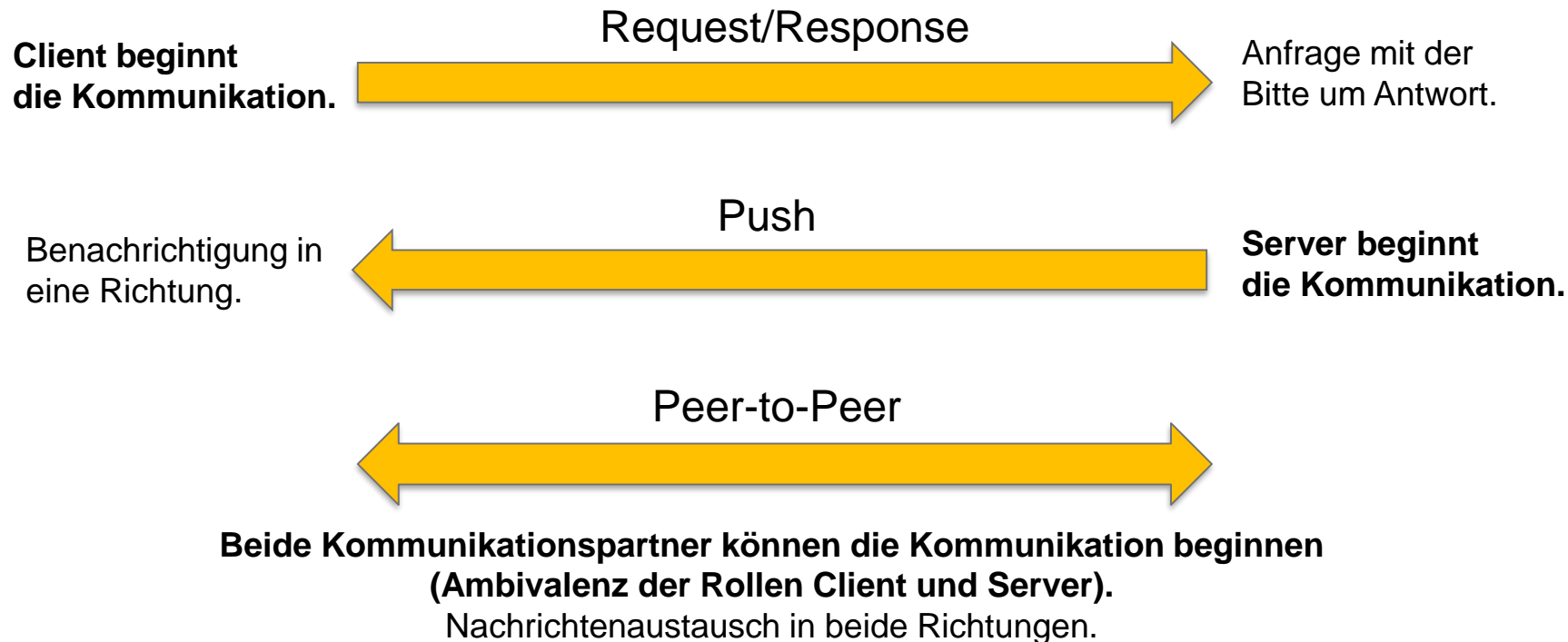
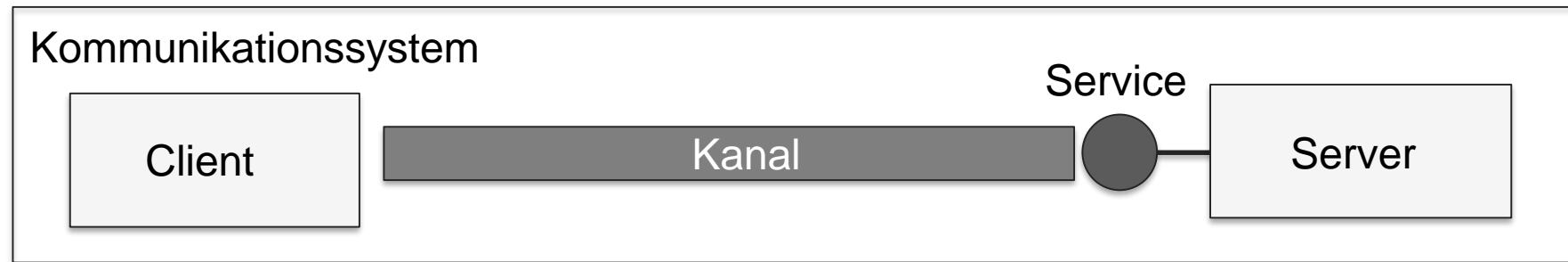
Eine **Serviceschnittstelle** ist ein Vertrag zwischen Nutzer und Anbieter über Syntax und Semantik der Service-Nutzung und enthält optional Zusicherungen in Hinblick auf den **Quality of Service**.

Klassifikation von Kommunikationssystemen: Kardinalität der Empfänger einer Nachricht.



Klassifikation von Kommunikationssystemen:

(B) Wer beginnt mit der Kommunikation?



Basis aller Cloud-Kommunikationstechnologien ist TCP und teilweise HTTP.

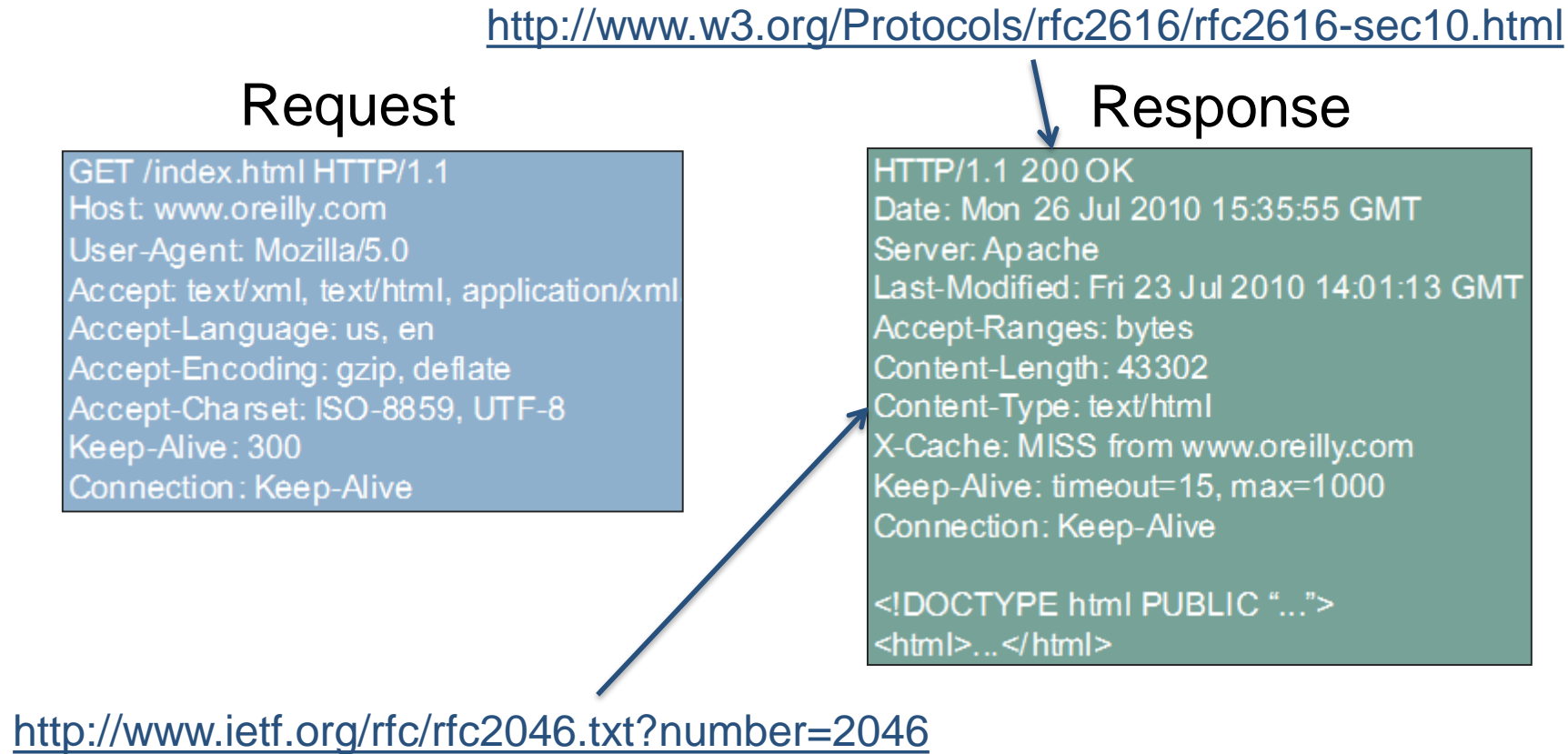
TCP (Transmission Control Protocol)	
Familie:	Internetprotokollfamilie
Einsatzgebiet:	Zuverlässiger bidirektionaler Datentransport
TCP im TCP/IP-Protokollstapel:	
Anwendung	HTTP SMTP ...
Transport	TCP
Internet	IP (IPv4, IPv6)
Netzzugang	Ethernet Token Bus Token Ring FDDI ...
Standards:	RFC 793 ↗ (1981) RFC 1323 ↗ (1992)

- Ab 1973 entwickelt und 1981 standardisiert.
- Zuverlässige Voll-Duplex Ende-zu-Ende Verbindung.
- Ein Endpunkt ist eine IP + Port.

HTTP (Hypertext Transfer Protocol)	
Familie:	Internetprotokollfamilie
Einsatzgebiet:	Datenübertragung, Hypertext u. a.
Port:	80/TCP
HTTP im TCP/IP-Protokollstapel:	
Anwendung	HTTP
Transport	TCP
Internet	IP (IPv4, IPv6)
Netzzugang	Ethernet Token Bus Token Ring FDDI ...
Standards:	RFC 1945 ↗ (HTTP/1.0, 1996) RFC 2616 ↗ (HTTP/1.1, 1999)

- HTTP 1.0: 1989 am CERN entwickelt.
- HTTP 1.1: Connection Pooling / Keepalive, HTTP-Pipelining, Methoden PUT und DELETE.
- HTTP 2.0: Binär-Stream, Multiplexing, Verschlüsselung als Standard, div. Performance-Optimierungen, Push. (siehe <https://http2.github.io>)

Ein Beispiel für eine HTTP-Kommunikation.



Typische Datenformate im Internet: XML

```
<Kreditkarte
  Herausgeber="Xema"
  Nummer="1234-5678-9012-3456"
  Deckung="2e+6"
  Waehrung="EURO">
  <Inhaber
    Name="Mustermann"
    Vorname="Max"
    maennlich="true"
    Alter="42"
    Partner="null">
    <Hobbys>
      <Hobby>Reiten</Hobby>
      <Hobby>Golfen</Hobby>
      <Hobby>Lesen</Hobby>
    </Hobbys>
    <Kinder />
  </Inhaber>
</Kreditkarte>
```

XML = eXtensible Markup Language
(Daten und ihre Beschreibung)

MIME-Types: *text/xml*, *application/xml*

Schema-Sprachen: XML Schema, DTD, Relax NG

Datentypen

- Elemente
- Attribute
- Textknoten
- Listen, Sequenzen, Auswahlen
- 19 primitive Datentypen (string, integer, bool, ...)
- 25 abgeleitete Datentypen (ID, IDREF, URI, ...)

Typische Datenformate im Internet: JSON

Objekt

```
{
  "Herausgeber": "Xema",
  "Nummer": "1234-5678-9012-3456",
  "Deckung": 2e+6,
  "Währung": "EURO",
  "Inhaber": {
    "Name": "Mustermann",
    "Vorname": "Max",
    "männlich": true,
    "Hobbys": [ "Reiten", "Golfen", "Lesen" ],
    "Alter": 42,
    "Kinder": [],
    "Partner": null
  }
}
```

JSON = JavaScript Object Notation
(Daten pur). Auch in Binärcodierung (BSON – Binary JSON).

MIME-Typ: *application/json*

Schema-Sprachen: JSON Schema (<http://json-schema.org>)

Datentypen

- Nullwert: **null**
- bool'scher Wert: **true**, **false**
- Zahl: **42**, **2e+6**
- Zeichenkette: **"Mustermann"**
- Array: **[1, 2, 3]**
- Objekt mit Eigenschaften: **{ "Name": "Mustermann" }**



Service-orientierte Request- Response-Kommunikation mit REST

REST ist ein Paradigma für Anwendungsservices auf Basis des HTTP-Protokolls.

- REST ist eine Paradigma für den Schnittstellenentwurf von Internetanwendungen auf Basis des HTTP-Protokolls.
- Dissertation von Roy Fielding: „Architectural Styles and the Design of Network-based Software Architectures“, 2000, University of California, Irvine.

Grundlegende Eigenschaften:

- **Alles ist eine Ressource:** Eine Ressource ist eindeutig adressierbar über einen URI, hat eine oder mehrere Repräsentationen (XML, JSON, bel. MIME-Typ) und kann per Hyperlink auf andere Ressourcen verweisen. Ressourcen sind, wo immer möglich, hierarchisch navigierbar.
- **Uniforme Schnittstellen:** Services auf Basis der HTTP-Methoden (PUT = erzeugen, POST = aktualisieren oder erzeugen, DELETE = löschen, GET = abfragen). Fehler werden über die HTTP Codes zurückgemeldet. Services haben somit eine standardisierte Semantik und eine stabile Syntax.
- **Zustandslosigkeit:** Die Kommunikation zwischen Server und Client ist zustandslos. Ein Zustand wird im Client nur durch URIs gehalten.
- **Konnektivität:** Basiert auf ausgereifter und allgegenwärtiger Infrastruktur: Der Web-Infrastruktur mit wirkungsvollen Caching- und Sicherheitsmechanismen, leistungsfähigen Servern und z.B. Web-Browser als Clients.



Beispiele für REST-Aufrufsyntax: Schnittstellenentwurf über Substantive.

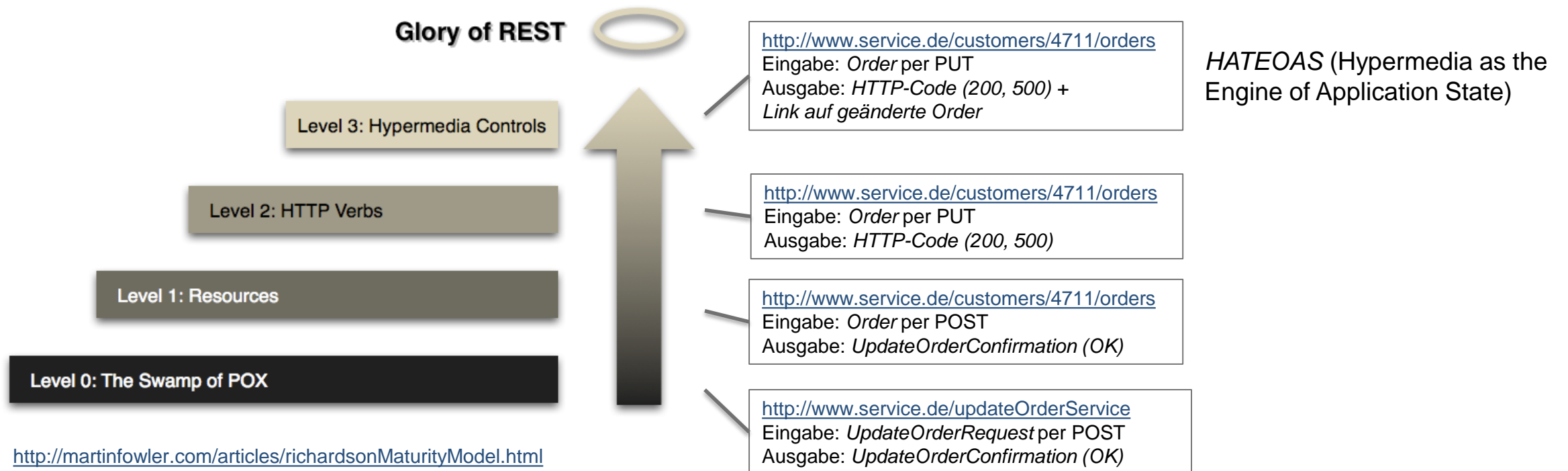
- Produkte aus der Kategorie Spielwaren:
<http://www.service.de/produkte/spielwaren>
- Bestellungen aus dem Jahr 2008
<http://www.service.de/bestellungen/2008>
- Liste aller Regionen, in denen der Umsatz größer als 5 Mio. Euro
<http://www.service.de/regionen/umsatz/summe?groesserAls=5M>
- Gib mir die zweite Seite aus dem Produktkatalog
<http://www.service.de/produkte/2>
- Alle Gruppen, in den der Benutzer „josef.adersberger“ Mitglied ist.
<http://www.service.de/benutzer/josef.adersberger/gruppen>

Gängige Entwurfsregeln:

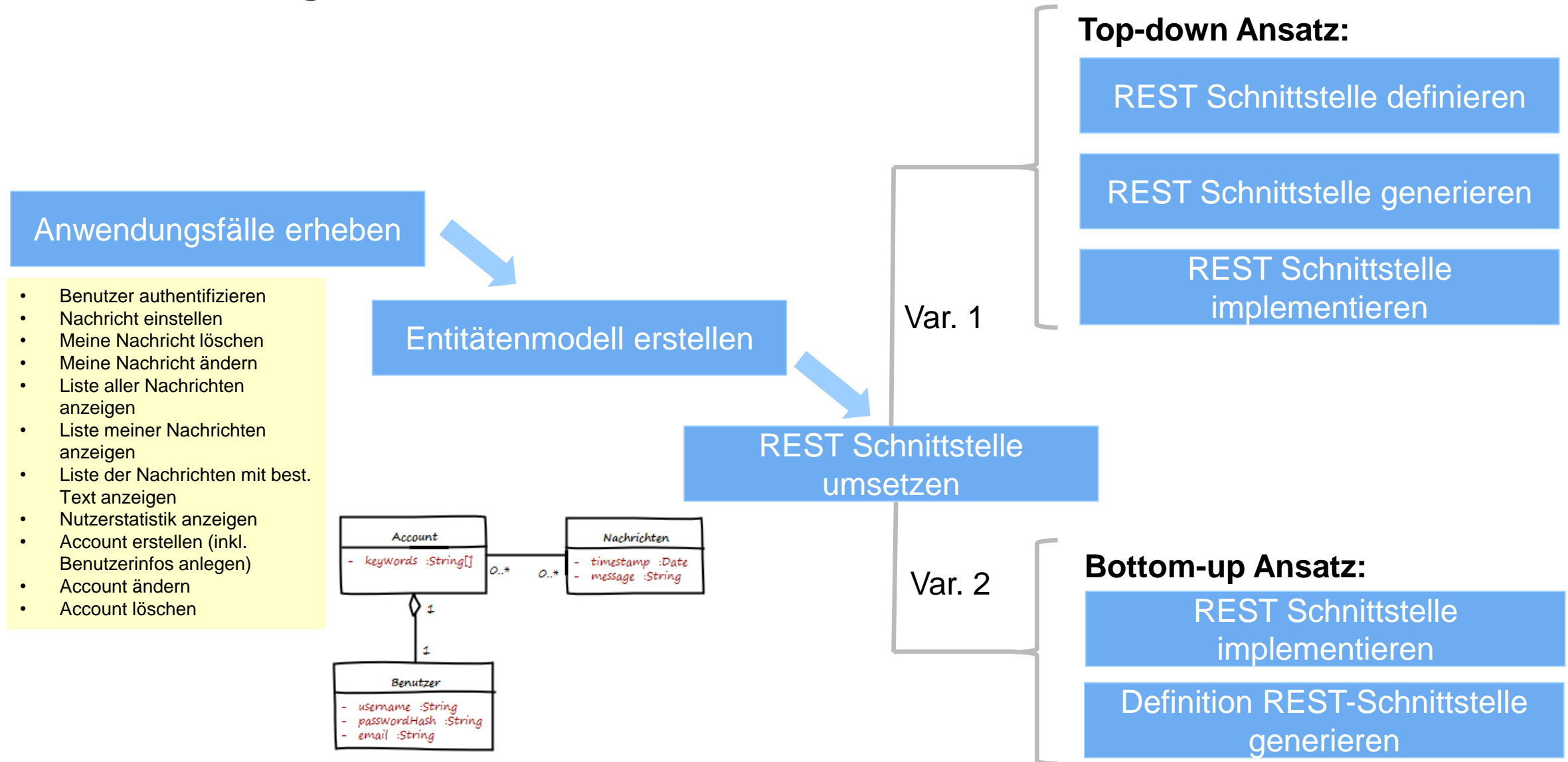
- Plural, wenn auf Menge an Entitäten referenziert werden soll. Sonst singular.
- Pfad-Parameter, wenn Reihenfolge der Angabe wichtig. Sonst Query Parameter.
- Standard Query Parameter einführen (z.B. für Filter und Abfragen sowie seitenweisen Zugriff) und konsistent halten.
- Pfad-Abstieg, wenn Entitäten per Aggregation oder Komposition verbunden sind.
- Pfad-Abstieg, wenn es sich um einen gängigen Navigationsweg handelt.
- Ids als Pfad-Parameter abbilden.
- Fehler und Ausnahmen über Return Codes abbilden. Einen Standard-Code suchen, der von der Semantik her passt.

Siehe auch: <http://codeplanet.io/principles-good-restful-api-design>

Mit dem REST Maturity Model kann bewertet werden, wie RESTful ein HTTP-basierter Service ist.



Entwicklung von REST APIs



REST-Webservices mit JAX-RS.

<http://www.service.de/hello/Josef?salutation=Servus>



Request Path

```
@Path("/hello/{name}")
public class HelloWorldResource {

    @GET, @POST, @PUT, @DELETE
    @GET
    @Produces("application/json")
    public ResponseMessage getMessage(
        @DefaultValue("Hallo") @QueryParam("salutation") String salutation,
        @PathParam("name") String name) throws IOException {
        ResponseMessage response = new ResponseMessage(new Date().toString(), salutation + " " + name);
        return response;
    }
}
```

Analog @Consumes für 1. Parameter

Analog @FormParam bei POST Requests

Die effizienten Alternativen: Binärprotokolle

Binärprotokolle sind eine sinnvolle Alternative zu REST, wenn eine effiziente und programmiersprachennahe Kommunikation erfolgen soll.

- Encoding der Payload als komprimiertes Binärformat
- Separate Schnittstellenbeschreibungen (IDLs, *Interface Definition Languages*) aus denen dann Client- und Server-Code in mehreren Programmiersprachen generiert werden können

Kandidaten

- gRPC / Protocol Buffers
- Apache Avro
- Apache Thrift
- Hessian

Binärprotokolle können auch mit REST kombiniert werden: Als Content-Type und damit als Payload wird eine Binär-Codierung verwendet. Beispiel: Protocol Buffers over REST.

gRPC

- Open-Source-Binärprotokoll von Google auf Basis der Protocol Buffers Binärcodierung (<http://www.grpc.io/docs>)
- Flexibel erweiterbarer Generator (protoc) für Server- und Client-Code (Skeleton und Stubs).

```
syntax = "proto3";

option java_package = "io.grpc.examples";

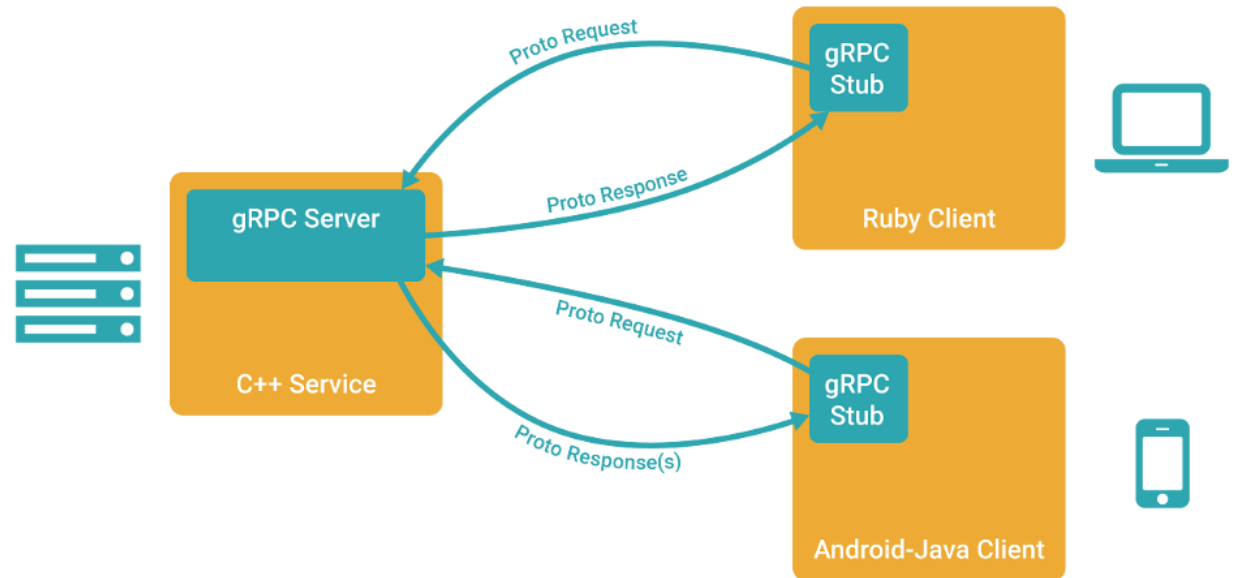
package helloworld;

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

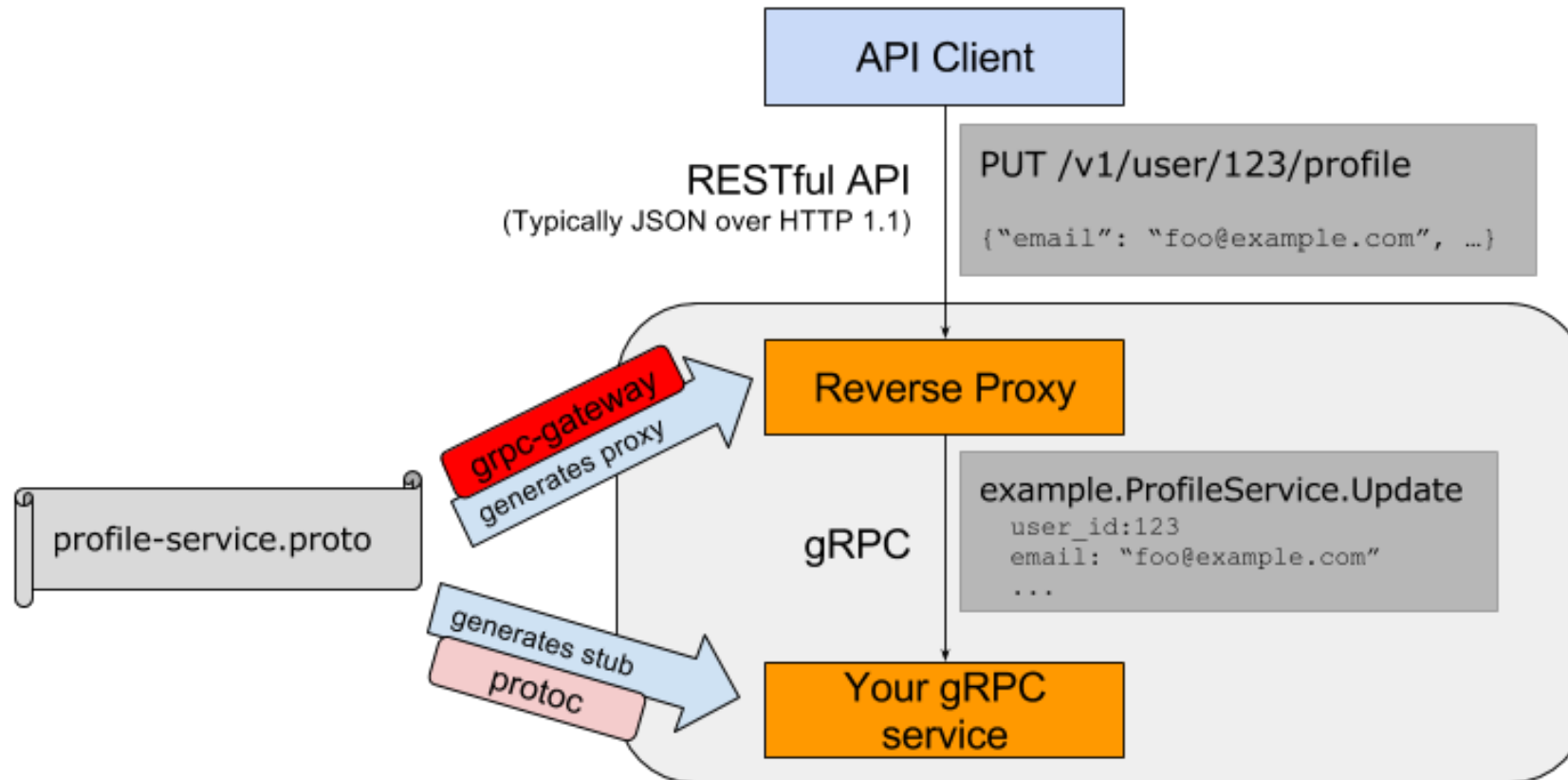
// The request message containing the user's name.
message HelloRequest {
  string name = 1;
}

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

protoc



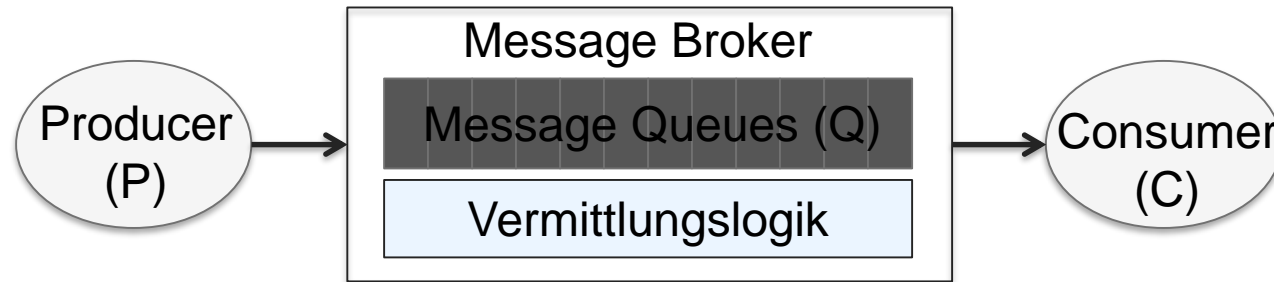
Dank HTTP/2 Multiplexing kann eine Anwendung auf dem selben HTTP-Port ein Binärprotokoll als auch REST anzubieten.





Flexible Kommunikationsmuster mit Messaging

Messaging ist zuverlässiger, asynchroner Nachrichtenaustausch.



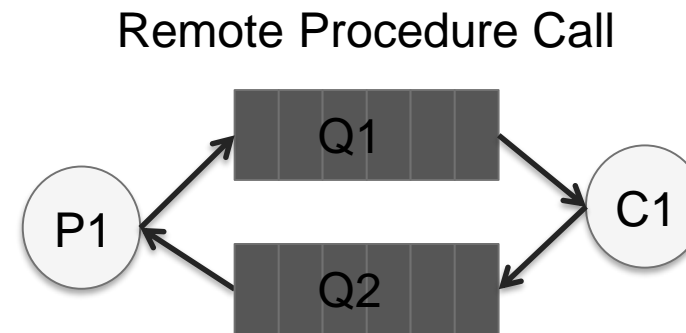
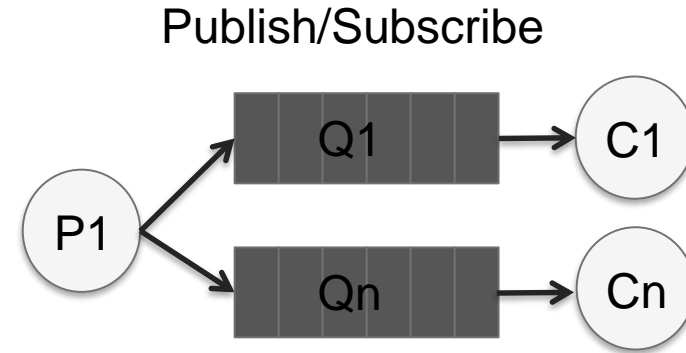
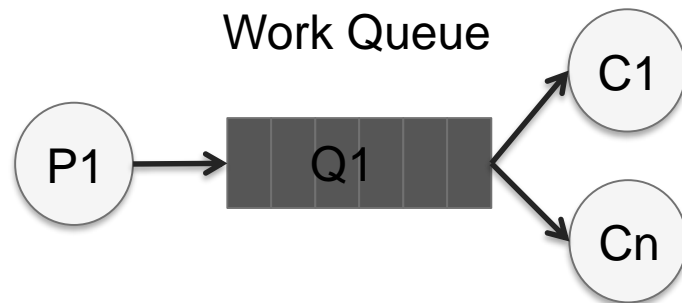
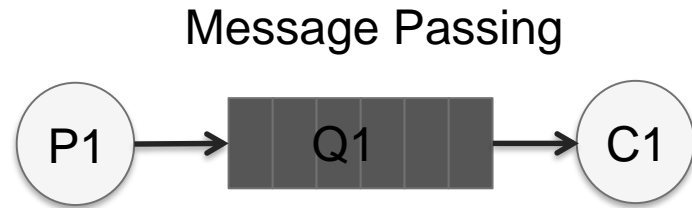
Entkopplung von Producer und Consumer.

Die Serviceschnittstelle ist lediglich das Format der Nachricht. Message Broker machen zum Format keinen Einschränkungen. Sende-Zeitpunkt und Empfangs-Zeitpunkt können beliebig lange auseinander liegen.

Skalierbarkeit. Die Vermittlungslogik entscheidet zentral ...

- ... an wie viele Consumer die Nachricht ausgeliefert wird (horizontale Skalierbarkeit),
 - an welchen Consumer die Nachricht ausgeliefert wird (Lastverteilung),
 - wann eine Nachricht ausgeliefert wird (Pufferung von Lastspitzen),
- auf Basis von konfigurierten Anforderungen an die Vermittlung:
- Maximale Zustelldauer bzw. Lebenszeit der Nachricht
 - Geforderte Zustellgarantie (mindestens 1 Mal, exakt 1 Mal, an alle) und Transaktionalität
 - Priorität der Nachricht
 - Notwendige Einhaltung der Zustellreihenfolge

Messaging ist eine flexible Kommunikationsart, mit der sich vielfältige Kommunikationsmuster umsetzen lassen.



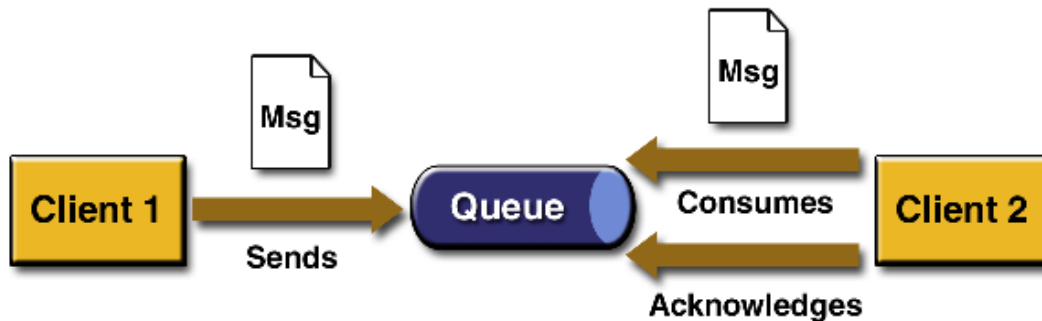
JMS

JMS = Java Messaging Service. Standardisierte API im Rahmen der Java-Enterprise-Edition-Spezifikation. Standardisiert nicht das Messaging-Protokoll.

- 2002-2013: Version 1.1. Sehr stabil und weit verbreitet in der Java-Welt.
- Seit Mai 2013: Version 2.0 als Teil der JEE 7 Spezifikation

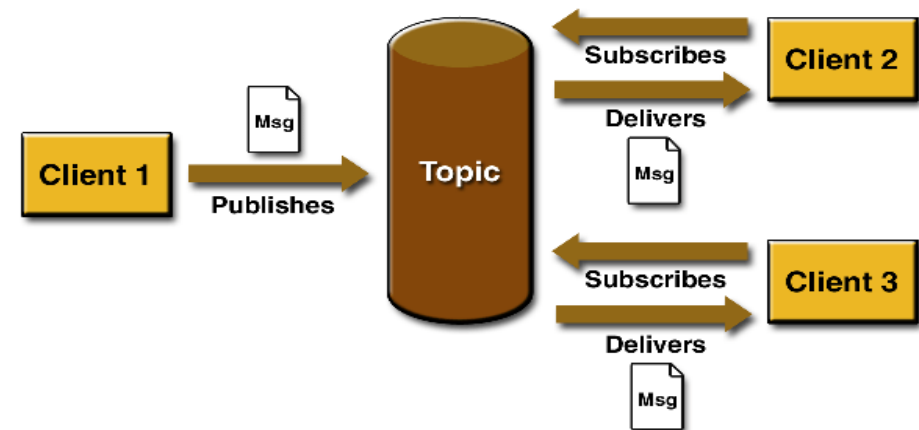
Unterstützte Kommunikationsmuster:

Message Passing:



- Ein Consumer pro Message
- Der Erhalt einer Nachricht wird bestätigt

Publish / Subscribe:



- Mehrere Consumer pro Message

AMQP: Ein Standard-Protokoll für Messaging-Systeme.

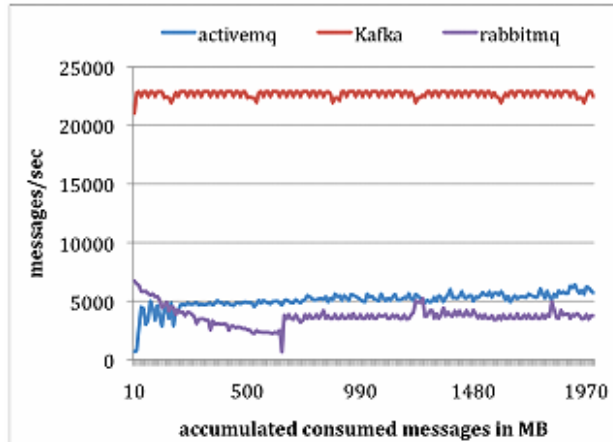
- **Problem:** Message Broker sind intern proprietär aufgebaut (Beispiel: IBM MQSeries mit 80% Marktanteil im kommerziellen Bereich). Sie sind nicht zueinander interoperabel, wie man es z.B. von SNMP-Servern her kennt. Das ist besonders beim Messaging über Firmengrenzen und Technologie-Stacks hinweg ein Problem.



- **Lösung AMQP:** Standardisierung eines interoperablen Protokolls für Messaging-Broker. AMQP steht seit Ende 2011 in der Version 1.0 zur Verfügung.
 - Im Standardisierungsgremium sind u.A. Cisco, Microsoft, Red Hat, Deutsche Börse Systems, IONA, Novell, Credit Suisse, JPMorganChase.
 - Standardisiert ein Netzwerk-Protokoll für die Kommunikation zwischen den Clients und den Message Brokern.
 - Standardisiert ein Modell der verfügbaren APIs und Bausteine für die Vermittlung und Speicherung von Nachrichten (Producer, Exchange, Queue, Consumer).
 - Unterstützung aller bekannter Messaging-Muster.

Kafka

- Entwickelt bei LinkedIn und 2011 als Open Source Projekt veröffentlicht
- Kafka hat sich zum de-facto Standard in der Cloud für Messaging entwickelt, da Kafka hochgradig verteilbar und deutlich schneller als vergleichbare Lösungen ist:



- Kafka ist so schnell, da es Betriebssystem-Mittel intelligent nutzt, ein effizientes Codierungsformat für Nachrichten besitzt und den Auslieferungszustand in den Clients hält.
- Kafka ist in Java und Scala geschrieben. Die Kafka API ist proprietär und orientiert sich an keinem Messaging-Standard.

Kafka basiert auf dem Konzept eines Event-Logs. Jeder Consumer hat einen eigenen Lese-Zeiger im Log.

Alte Events werden gemäß definierter Kriterien gelöscht (z.B. Alter, max. Topic-Größe)

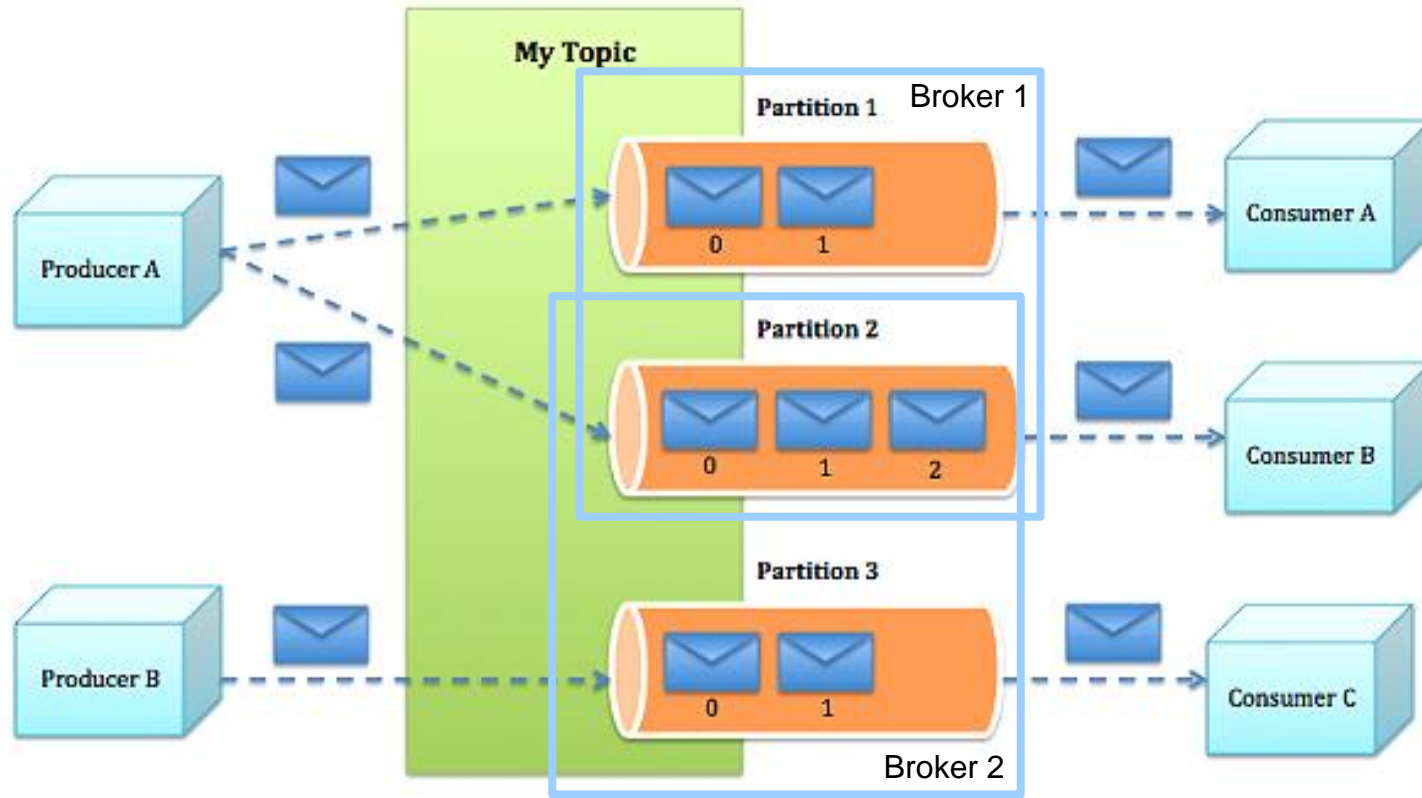


Neue Events werden immer am Ende des Topics angehängt



Zeiger auf letztes gelesenes Event eines Clients (verwaltet der Client selbst).

Der Event-Log in Kafka ist hochgradig verteilt.



- Die Events in einem Topic werden aufgeteilt in Partitionen
- Die Partitionen werden verteilt auf die verfügbaren Broker-Instanzen
- Partitionen werden zur Fehlertoleranz repliziert

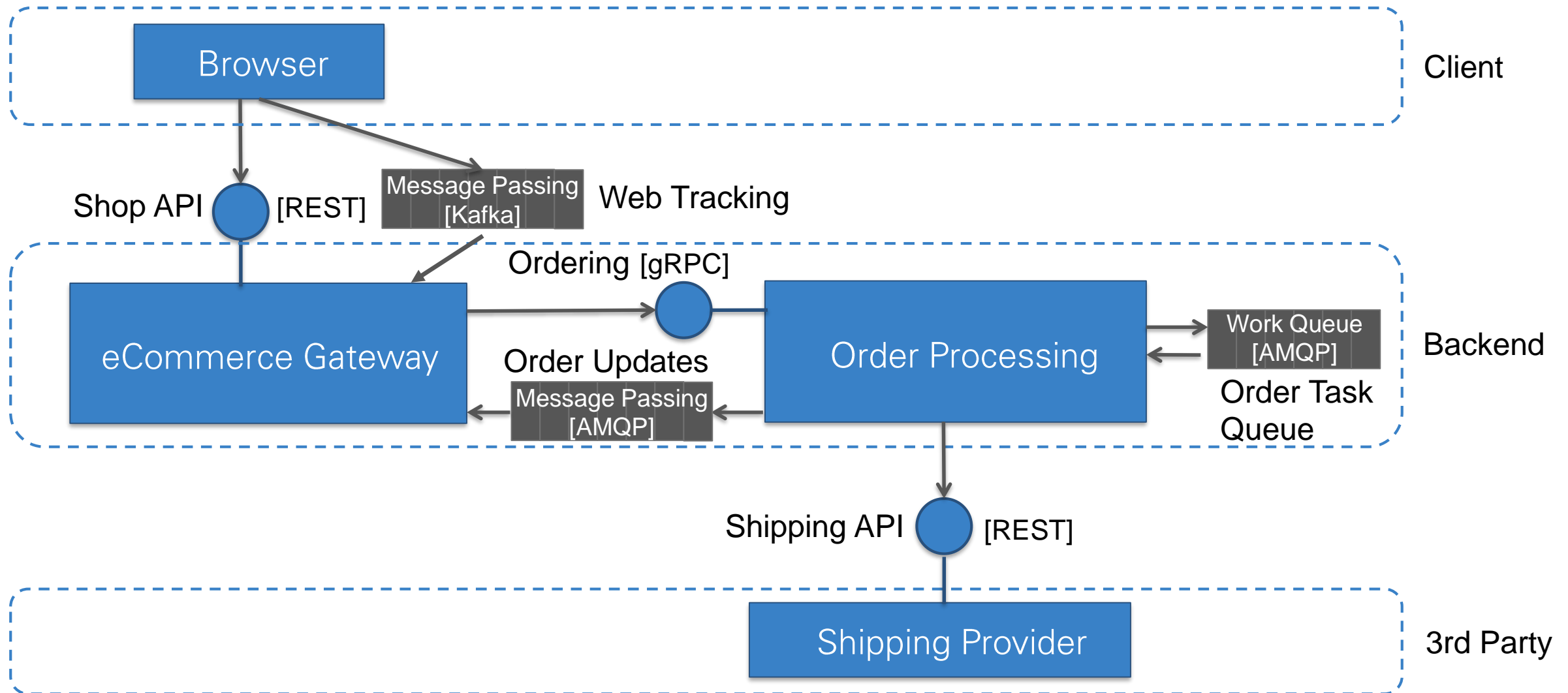
siehe:

- <http://www.michael-noll.com/blog/2013/03/13/running-a-multi-broker-apache-kafka-cluster-on-a-single-node>
- <http://www.infoq.com/articles/apache-kafka>



Architektur Aspekte

Putting it all together...





Literatur

Literatur

Bücher:

- Patterns of Enterprise Application Architecture, Martin Fowler, 2002
- Computer Networks, Andrew Tanenbaum, 2010
- Inter-Process Communication, Hephæstus Books, 2011

Internet:

- Dissertation von Roy Fielding zu REST
http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm
- RESTful Webservices
<http://www.ibm.com/developerworks/webservices/library/ws-restful>



Prolog zur Übung

Technische Basis ist Spring Boot.

SPRING INITIALIZR bootstrap your application now

Generate a Maven Project ▾ with Java ▾ and Spring Boot 1.5.7 ▾

Project Metadata

Artifact coordinates

Group

com.example

Artifact

demo

Dependencies

Add Spring Boot Starters and dependencies to your application

Search for dependencies

Web, Security, JPA, Actuator, Devtools...

Selected Dependencies

Generate Project alt + ↵

Don't know what to look for? Want more options? [Switch to the full version.](#)

REST API Implementierung mit JAX-RS.


```
@Component
@Path("/books")
@Api(value = "/books", description = "Operations about books")
@Produces(MediaType.APPLICATION_JSON)
public class BookResource {

    @Autowired
    private Bookshelf bookshelf;

    @GET
    @ApiOperation(value = "Find books", response = Book.class, responseContainer = "List")
    @ApiResponses(value = {
        @ApiResponse(code = 200, message = "Found all books")
    })
    public Response books(@ApiParam(value = "title to search")
        @QueryParam("title") String title) {
        Collection<Book> books = bookshelf.findByTitle(title);
        return Response.ok(books).build();
    }

    @POST
    @Consumes(MediaType.APPLICATION_JSON)
    @ApiOperation(value = "Create book")
    @ApiResponses(value = {
        @ApiResponse(code = 201, message = "Created the book"),
        @ApiResponse(code = 409, message = "Book already exists")
    })
    public Response create(Book book) {
        boolean created = bookshelf.create(book);
        if (created) {
            return Response.created(URI.create("/api/books/" + book.getIsbn())).build();
        } else {
            return Response.status(Response.Status.CONFLICT).build();
        }
    }
}
```

REST API Dokumentation mit Swagger.

 **swagger**

http://localhost:8080/api/swagger.json

Explore

1.0.1

[Base URL: localhost:8080/http://localhost:8080/api/]
<http://localhost:8080/api/swagger.json>

Schemes

HTTP

books

GET

/books/{isbn}

Find book by ISBN

PUT

/books/{isbn}

Update book by ISBN

DELETE

/books/{isbn}

Delete book by ISBN

GET

/books

Find books

POST

/books

Create book