

Kapitel 1: Kommunikation



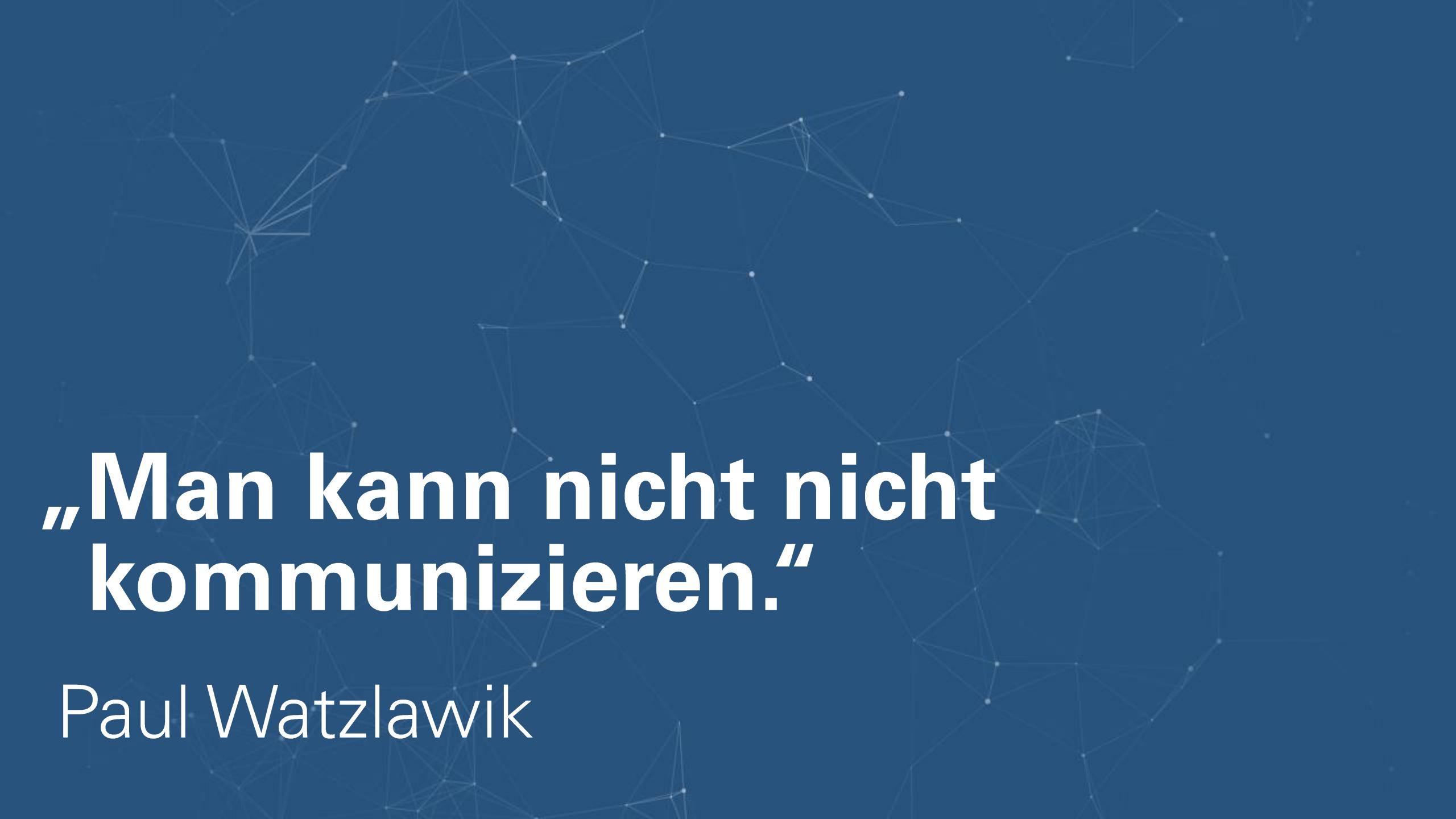
vorlesung

**CLOUD
COMPUTING**

Reminder:

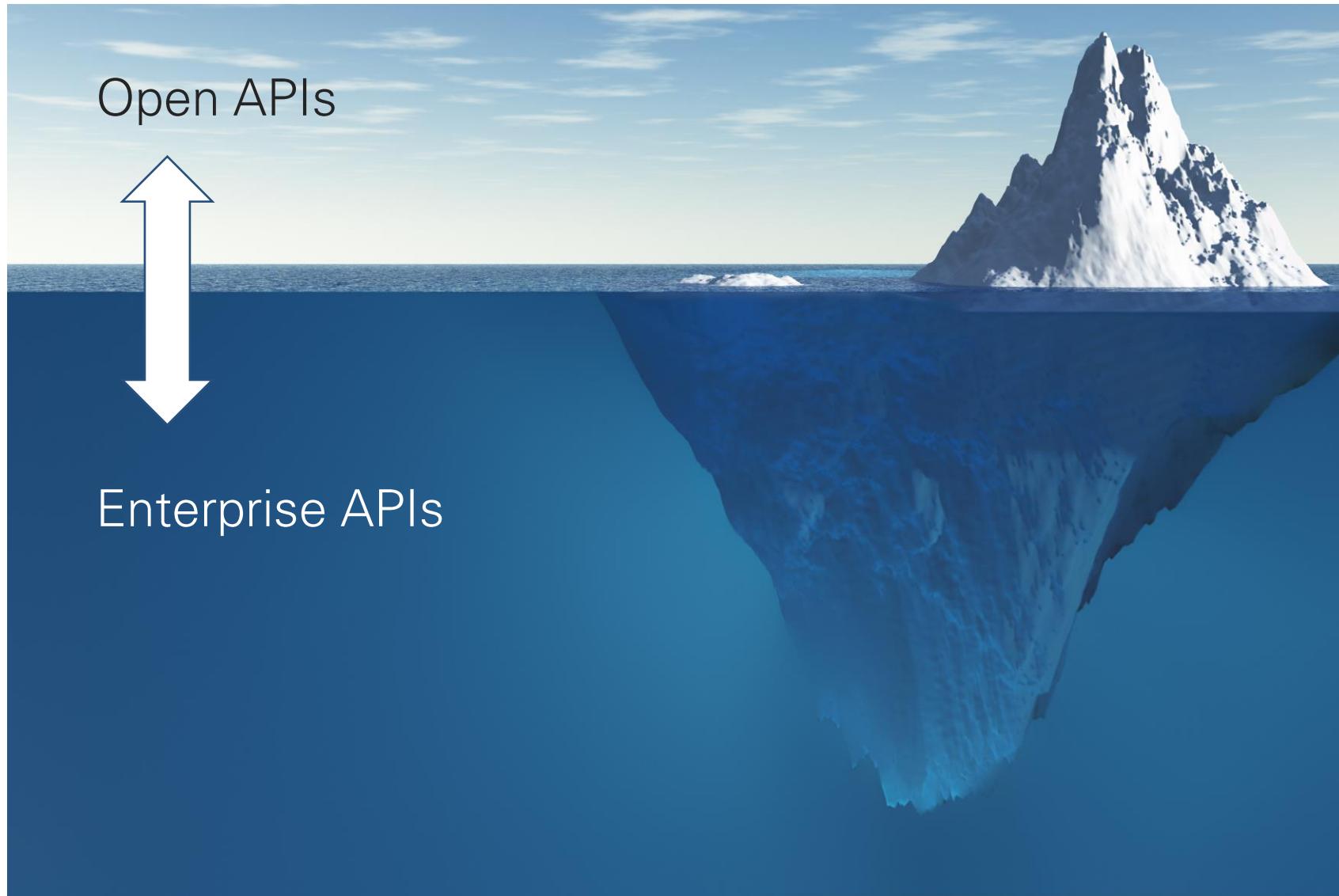
Github:

<https://github.com/qaware/cloud-computing-rosenheim-2021.git>

The background of the slide features a complex, abstract network structure composed of numerous small, semi-transparent white dots connected by thin white lines, forming a web-like pattern across the entire dark blue surface.

„Man kann nicht nicht
kommunizieren.“

Paul Watzlawik



<https://www.gettyimages.de/detail/foto/tip-of-the-iceberg-lizenzfreies-bild/157509282>

- Nur ein Bruchteil der bestehenden APIs ist öffentlich aufrufbar.
- Mit APIs lässt sich Geld verdienen.
- Bestimmte APIs müssen veröffentlicht werden.
- Nutzer öffentlicher APIs entwickeln (mit etwas Glück) innovative Produkte.

Beispiel: Mercedes Benz Developer Portal

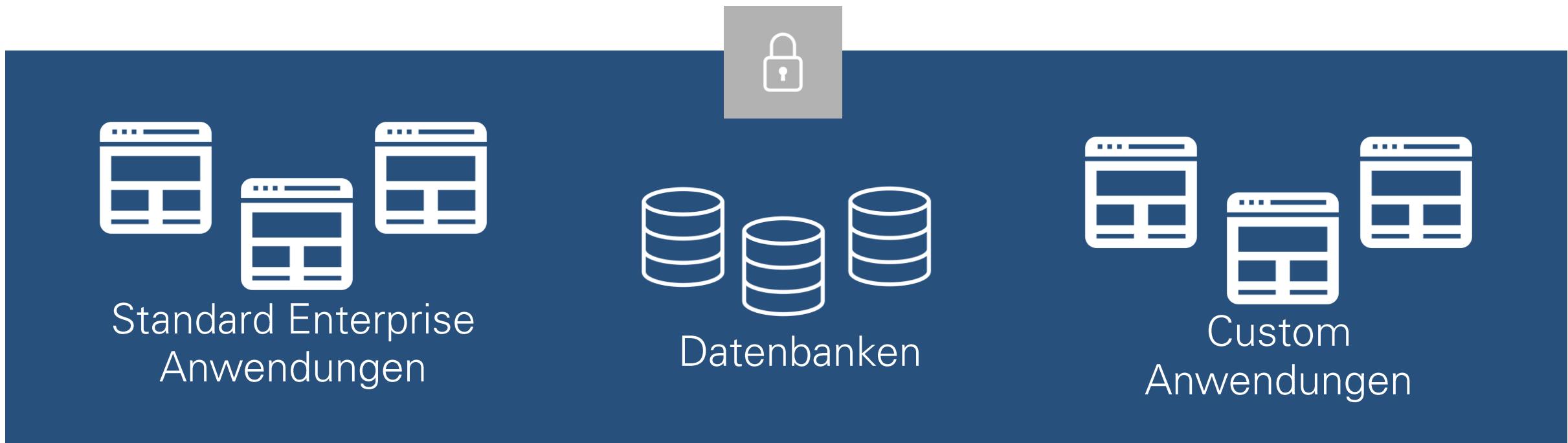
The screenshot shows the homepage of the Mercedes-Benz Developer Portal. At the top left is the Mercedes-Benz logo. To its right is the URL [/developers](#). The top navigation bar includes links for [APIs](#), [SDKs](#), [SUPPORT](#), [INSPIRE](#), [WHAT'S NEW](#), and [LOGIN](#). The main headline reads "We move the world by data." Below it, a welcome message says "Welcome to Mercedes-Benz /developers!" followed by a descriptive paragraph about API products and business models. A large, stylized blue circle graphic is positioned on the right side of the text. In the bottom right corner, there is a small icon of three interconnected circles.

<https://developer.mercedes-benz.com/>

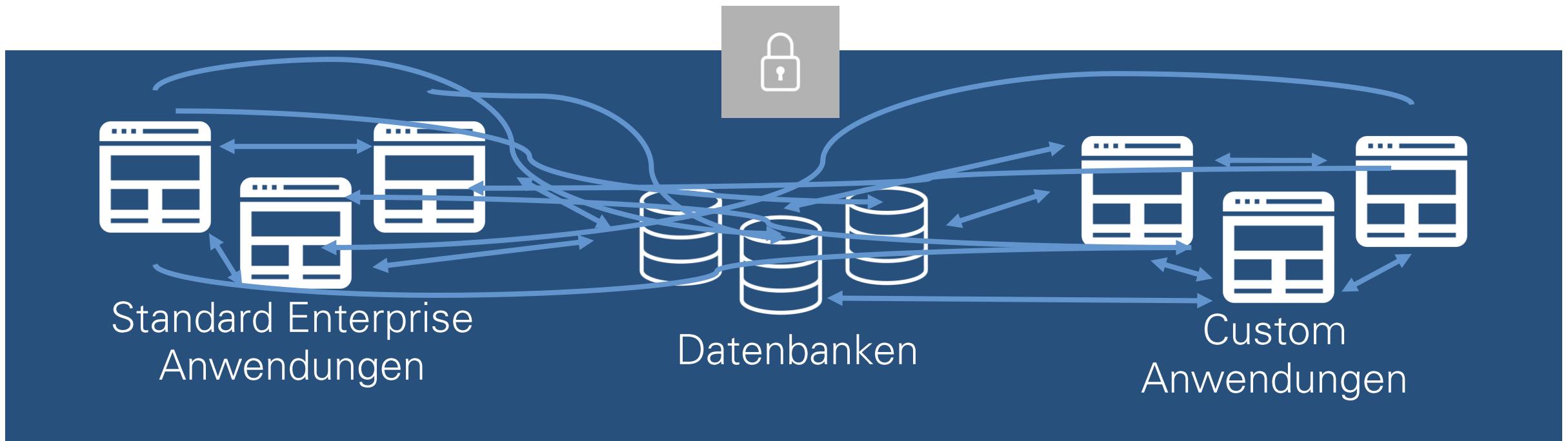
The background of the slide features a complex, abstract network structure composed of numerous small, semi-transparent white dots connected by thin white lines, forming a web-like pattern across the dark blue background.

Was hat das mit Cloud Computing zu tun?

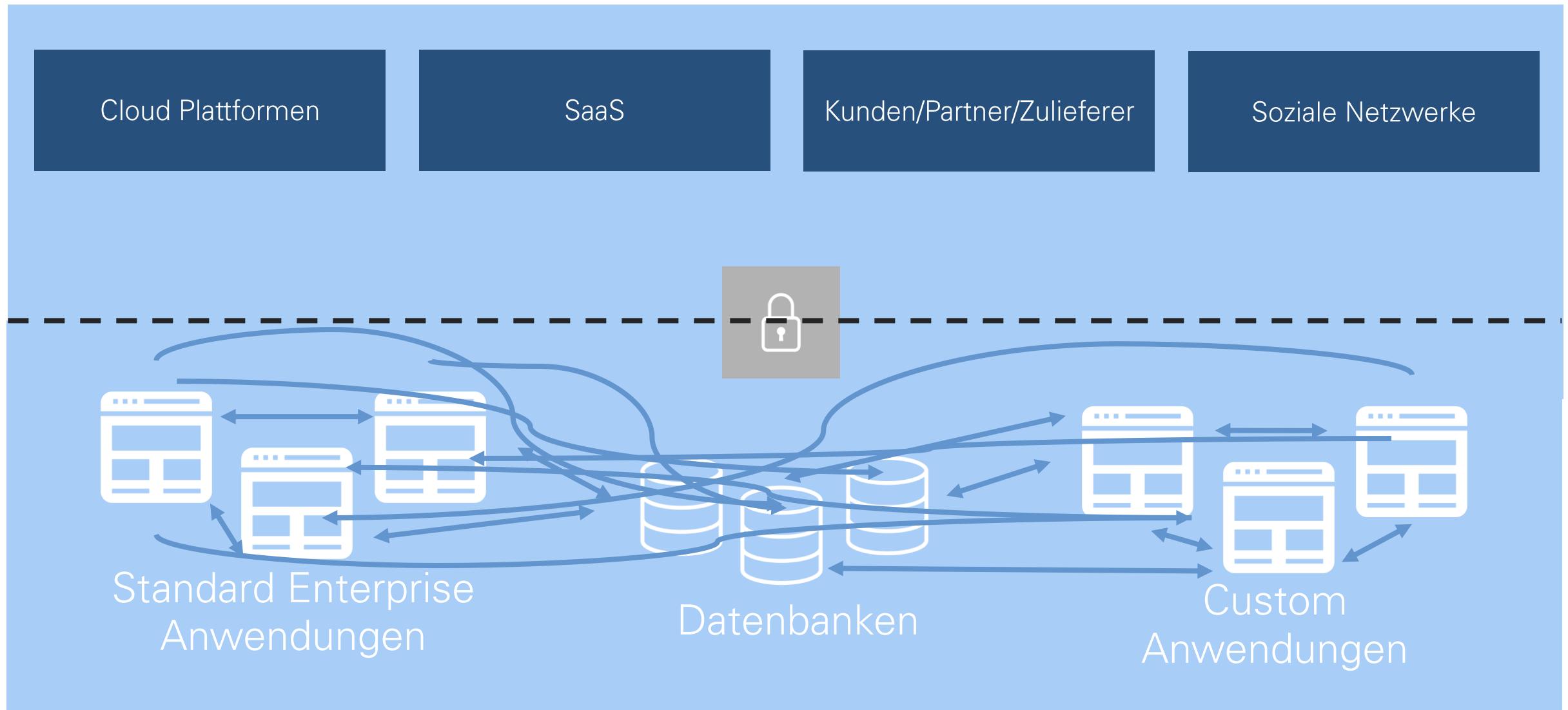
„Früher“ lief Enterprise Software hinter einer Firewall.



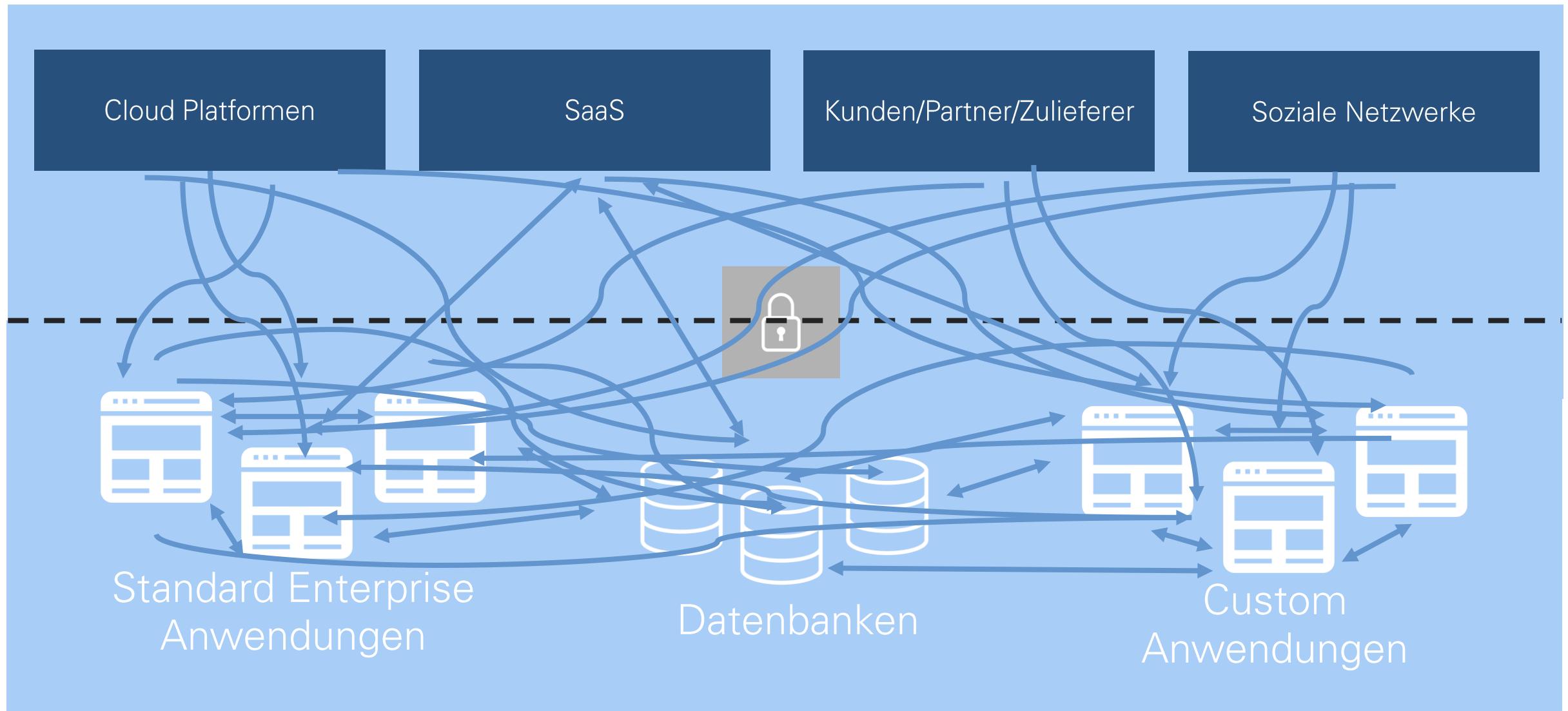
„Früher“ lief Enterprise Software hinter einer Firewall.
Die Komponenten kommunizieren intern.



Heute geht die Vernetzung noch deutlich weiter, auch über die Grenzen des Unternehmensnetzwerks hinaus:

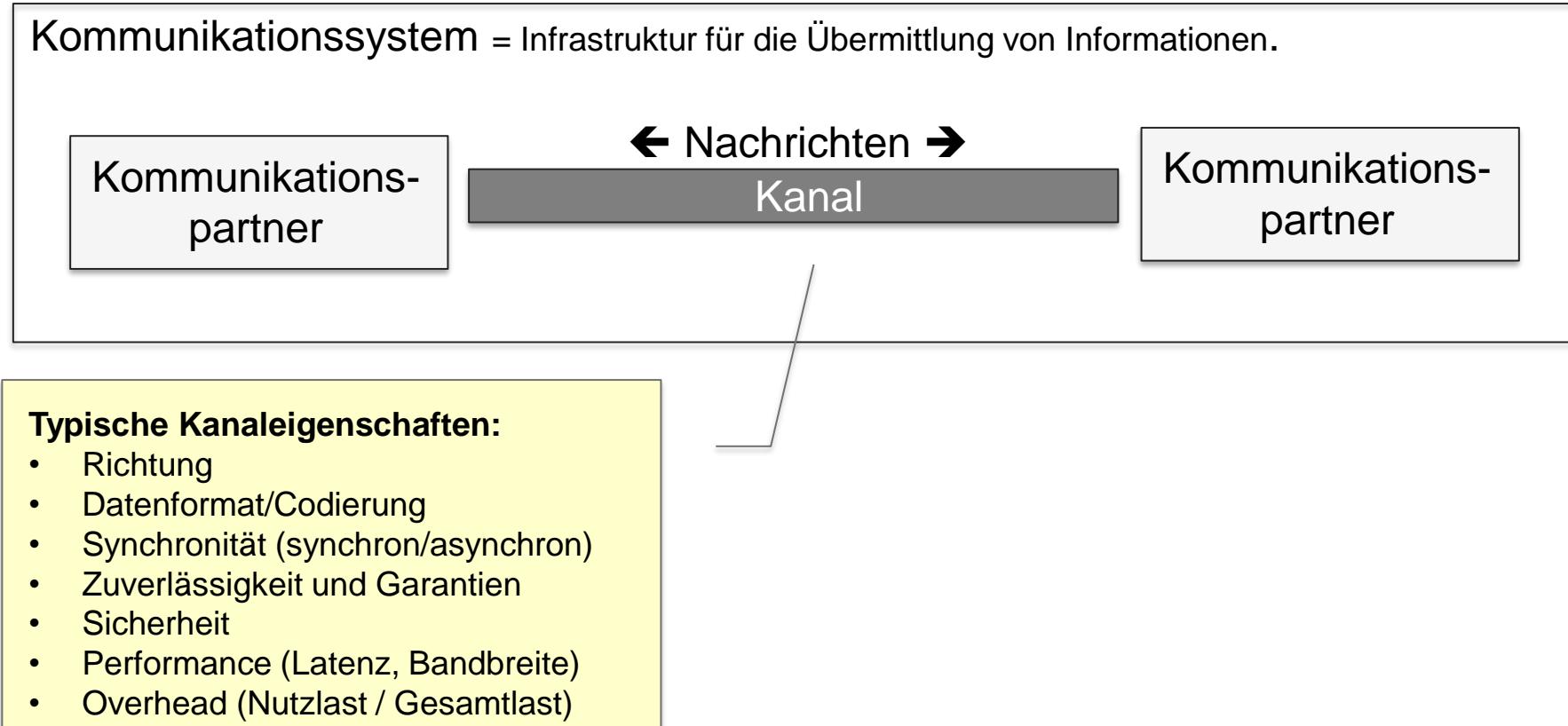


Heute geht die Vernetzung noch deutlich weiter, auch über die Grenzen des Unternehmensnetzwerks hinaus:

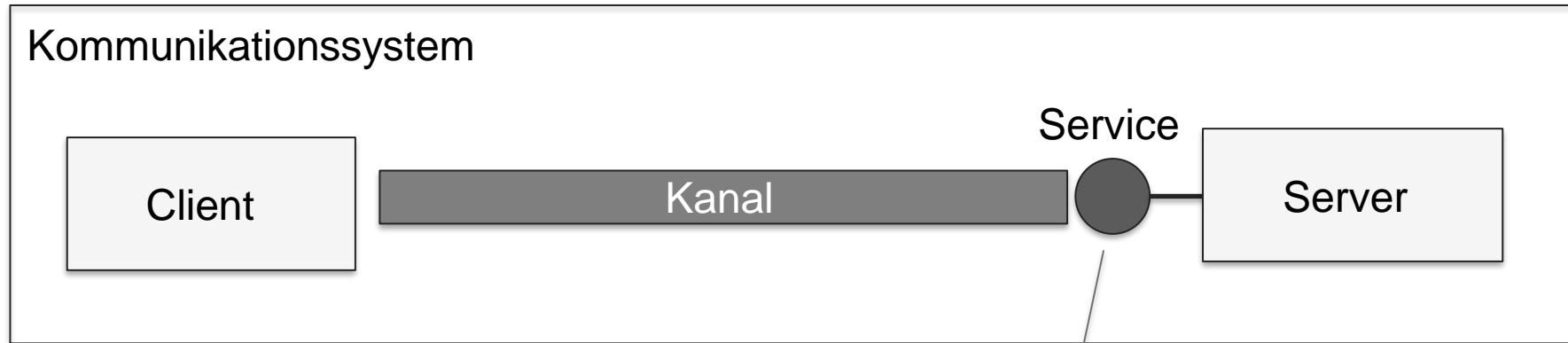


Kommunikationsmodelle und -kanäle

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



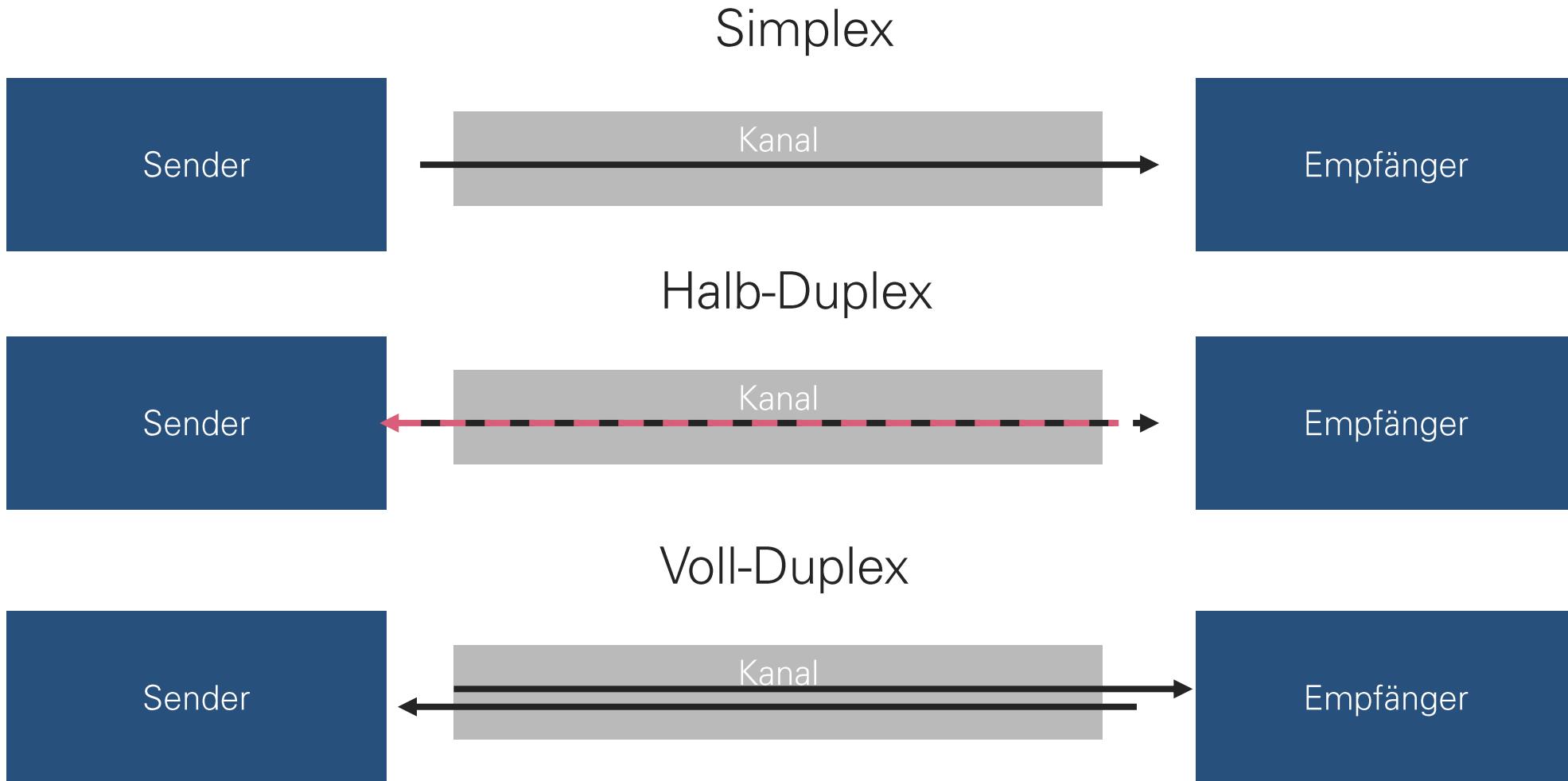
Service-Orientierung in einem Kommunikationssystem: Client-Server-Kommunikation über Services



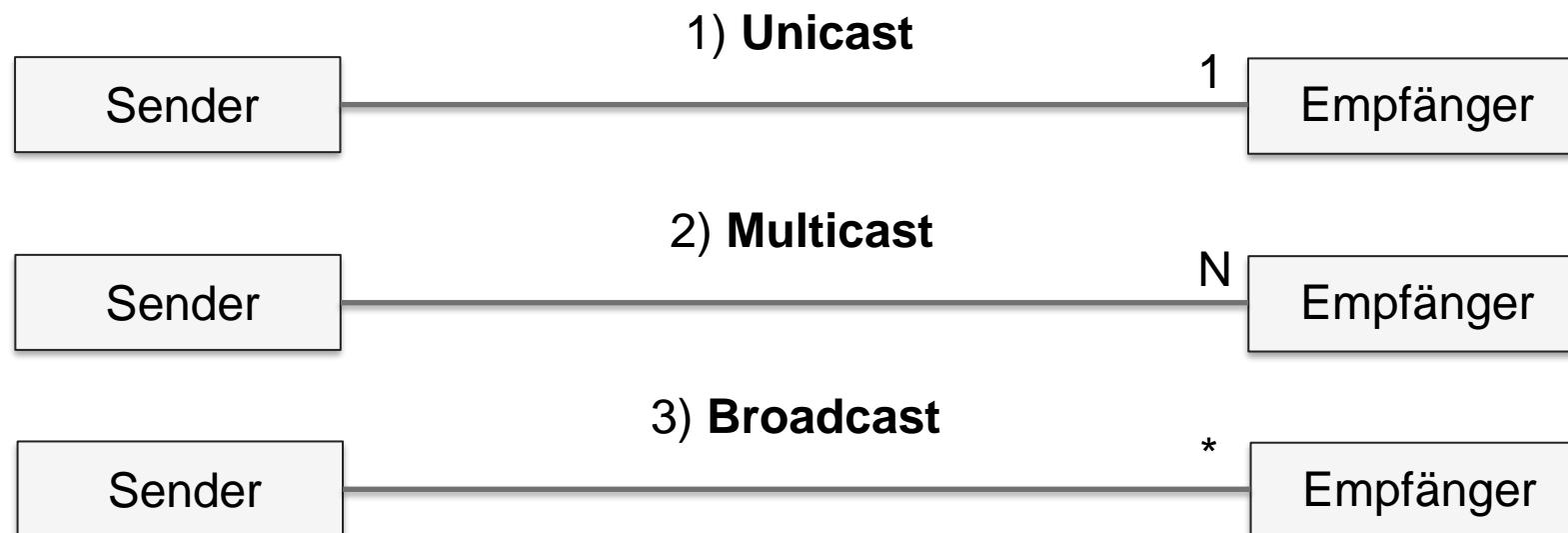
Ein **Service** ist eine Funktionalität, die über eine definierte Schnittstelle zur Verfügung gestellt wird. Jeder Service ist definiert durch eine **Serviceschnittstelle**.

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**.

Nutzungsmuster des Kanals

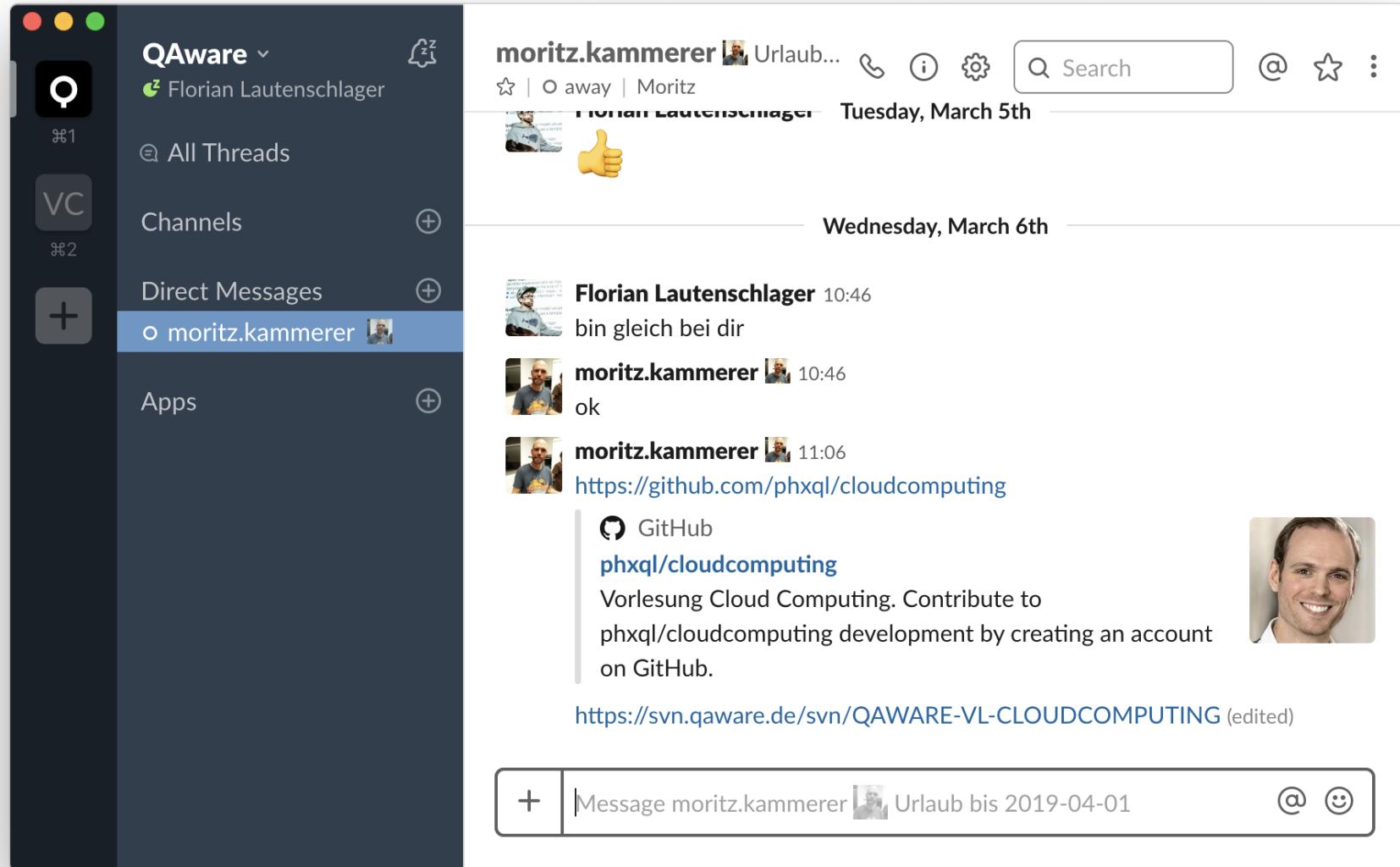


Klassifikation von Kommunikationssystemen: Kardinalität der Empfänger einer Nachricht.



Slack Beispiel :

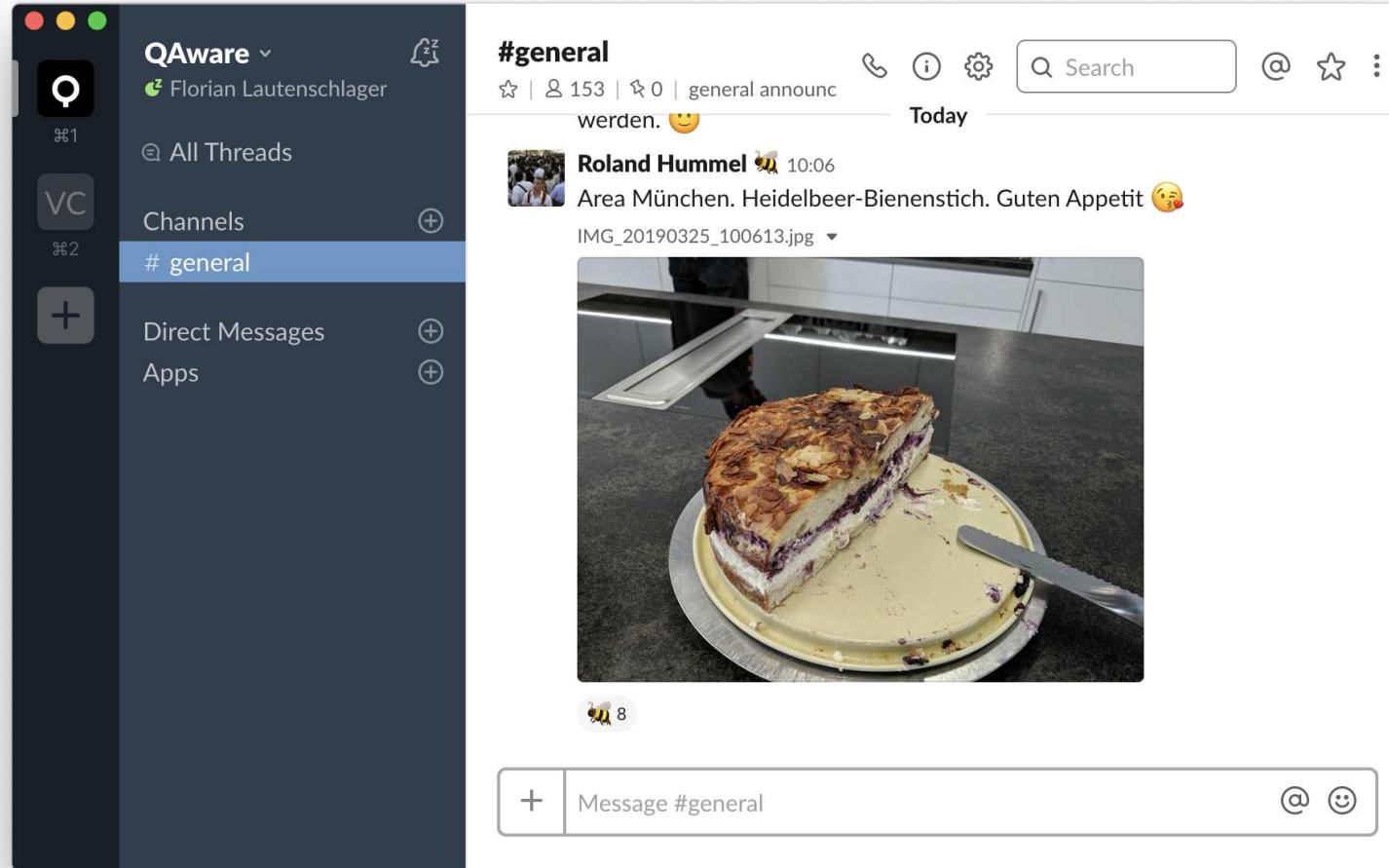
Unicast: Direct Message zwischen Moritz und Florian.



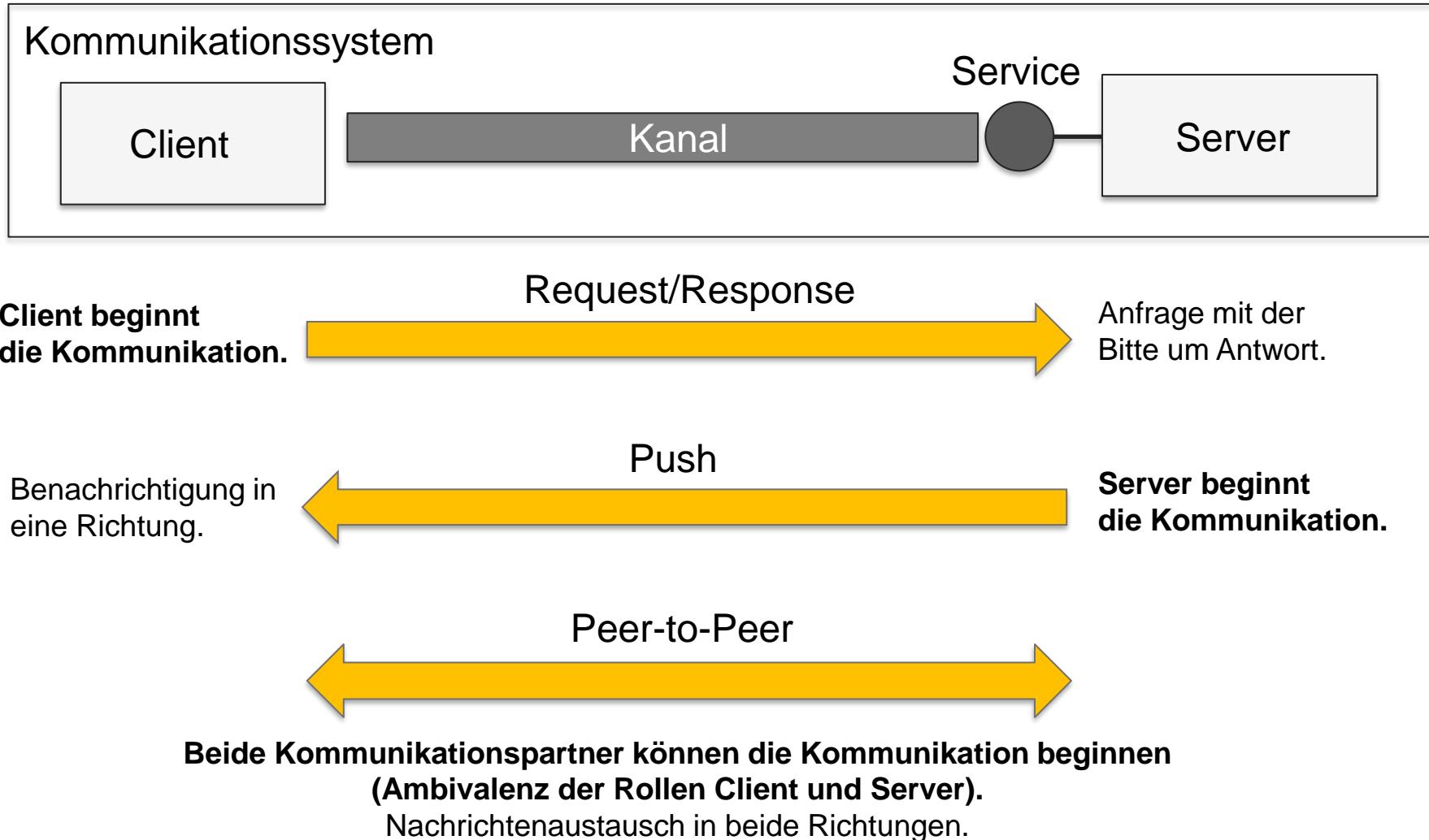
Slack Beispiel: Multicast: N Studenten in Cloud Computing

The screenshot shows the Slack interface. On the left is the sidebar with a dark purple background. It displays a list of channels and users. The channel **# cloud-computing-2019** is highlighted with a blue selection bar at the top of the list. Other visible channels include **# allgemein**, **# zufällig**, and **Direktnachrichten**. Below the channels is a list of users: Florian Lautenschlager, Andreas Pabst, Artem Polovyi, Kevin Richter, mitja, Moritz Kammerer, Nils Engelbrecht, and Samantha Klier. On the right is the main workspace for the **# cloud-computing-2019** channel. The header includes the channel name, a star icon, a user count of 11, a message count of 0, a 'Thema hinzufügen' button, and a search bar labeled 'Suchfunk...'. Below the header, a message from **Moritz Kammerer** is shown, stating they created the channel on March 4th. A timestamp 'Montag, 4. März' is displayed. At the bottom is a message input field with a placeholder 'Nachricht an #cloud-computing-2019' and a button with a plus sign.

Slack Beispiel: Broadcast: In *#general* sind alle Mitarbeiter*



Klassifikation von Kommunikationssystemen: Wer beginnt mit der Kommunikation?



Basis aller Cloud-Kommunikationstechnologien ist TCP und teilweise HTTP.

TCP (Transmission Control Protocol)	
Familie:	Internetprotokollfamilie
Einsatzgebiet:	Zuverlässiger bidirektonaler Datentransport
TCP im TCP/IP-Protokollstapel:	
Anwendung	HTTP SMTP ...
Transport	TCP
Internet	IP (IPv4, IPv6)
Netzzugang	Ethernet Token Token FDDI ... Bus Ring
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 Token FDDI ... Bus Ring
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.

<http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html>

Request

```
GET /index.html HTTP/1.1
Host: www.oreilly.com
User-Agent: Mozilla/5.0
Accept: text/xml, text/html, application/xml
Accept-Language: us, en
Accept-Encoding: gzip, deflate
Accept-Charset: ISO-8859, UTF-8
Keep-Alive: 300
Connection: Keep-Alive
```

<http://www.ietf.org/rfc/rfc2046.txt?number=2046>

Response

```
HTTP/1.1 200 OK
Date: Mon 26 Jul 2010 15:35:55 GMT
Server: Apache
Last-Modified: Fri 23 Jul 2010 14:01:13 GMT
Accept-Ranges: bytes
Content-Length: 43302
Content-Type: text/html
X-Cache: MISS from www.oreilly.com
Keep-Alive: timeout=15, max=1000
Connection: Keep-Alive

<!DOCTYPE html PUBLIC "...>
<html>...</html>
```

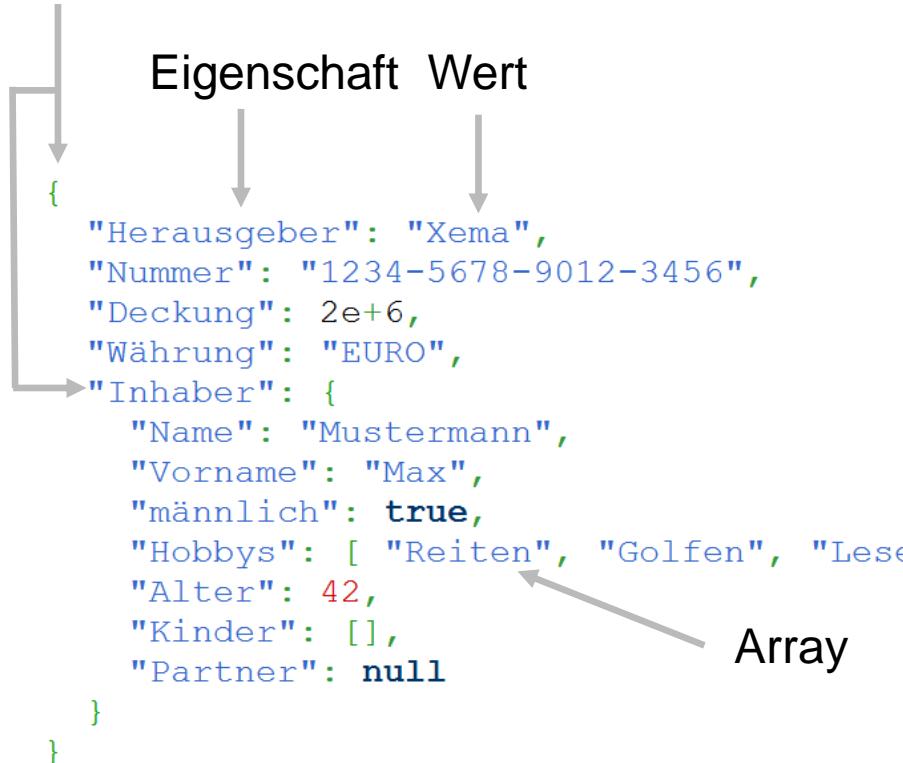
Typische Datenformate im Internet: XML

```
<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, Auswählen
 - 19 primitive Datentypen (string, integer, bool, ...)
 - 25 abgeleitete Datentypen (ID, IDREF, URI, ...)

Typische Datenformate im Internet: JSON

Objekt



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 (Verben).
- 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.

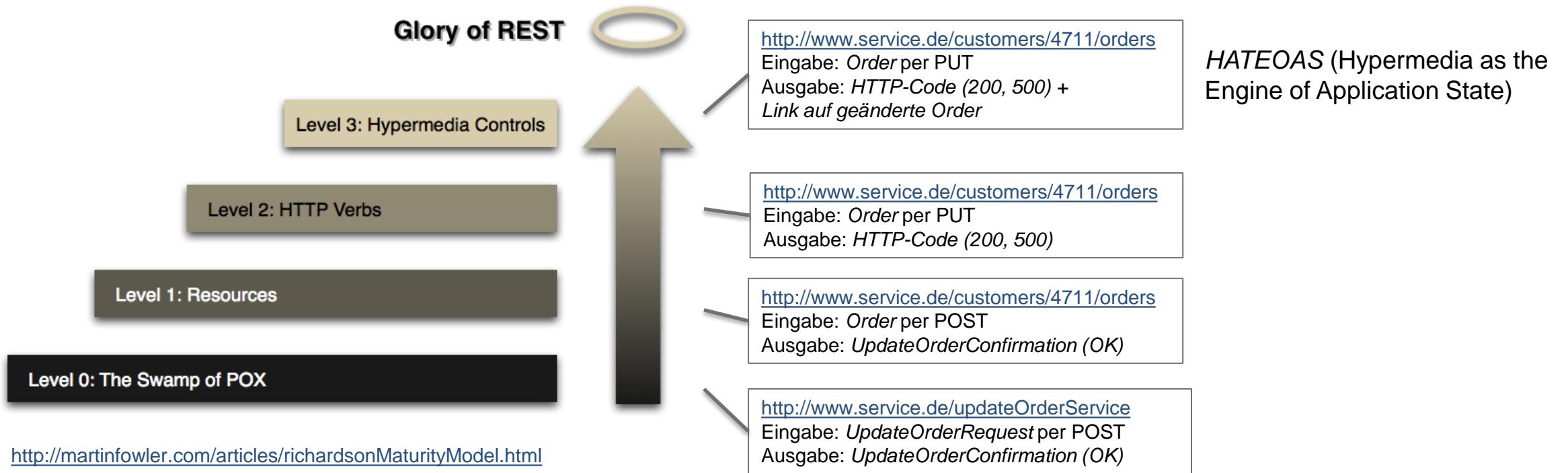
- Produkte aus der Kategorie Spielwaren:
<http://www.example.com/produkte/spielwaren>
- Bestellungen aus dem Jahr 2008
<http://www.example.com/bestellungen/2008>
- Liste aller Regionen, in denen der Umsatz größer als 5 Mio. Euro
<http://www.example.com/regionen/umsatz/summe?groesserAls=5M>
- Gib mir die zweite Seite aus dem Produktkatalog
<http://www.example.com/produkte/2>
- Alle Gruppen, in den der Benutzer „josef.adersberger“ Mitglied ist.
<http://www.example.com/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: <https://thomashunter.name/posts/2013-12-31-codeplanet-principles-of-good-restful-api-design>

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



Level 0

REQUEST POST <http://www.example.com/api/vehicle/requestVehicleHistory>

```
<requestHistoryOverview xmlns="http://www.example.com/SAM" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    refSchema=„history.xsd" "version="01.02.03">
    <historySearchArguments>
        <vehicle identification=„ABC12345B39676543"/>
    </historySearchArguments>
    <dealer identification=„12345"/>
</requestHistoryOverview>
```

RESPONSE

```
<requestVehicleHistoryResponse refSchema=„history.xsd" version="01.02.03" {...} >
    <historyId> ABC12345B39676543_20191008_001</historyId>
    <header>
        <dealer identification=„76543"/>
        <number>231714</number>
    ...

```

Level 2

vehicle identification

REQUEST: GET <http://www.example.com/api/vehicle/{ABC12345B39676543}/history>

RESPONSE

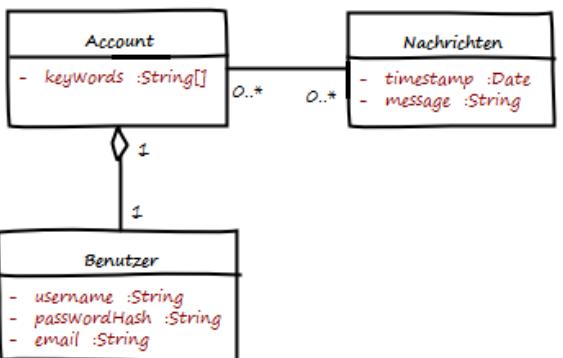
```
{"brand": "BMW",
"entries": [
  {
    "date": "2018-10-25T00:00:00",
    "dealer": "ABC-1234567 Test Dealer",
    "dealerNumber": "987654321"
  }
]}
```

Entwicklung von REST APIs

Anwendungsfälle erheben

- Benutzer authentifizieren
- Nachricht einstellen
- Meine Nachricht löschen
- Meine Nachricht ändern
- Liste aller Nachrichten anzeigen
- Liste meiner Nachrichten anzeigen
- Liste der Nachrichten mit best. Text anzeigen
- Nutzerstatistik anzeigen
- Account erstellen (inkl. Benutzerinfos anlegen)
- Account ändern
- Account löschen

Entitätenmodell erstellen



REST Schnittstelle umsetzen

Var. 1

Top-down Ansatz:

REST Schnittstelle definieren

REST Schnittstelle generieren

REST Schnittstelle implementieren

Var. 2

Bottom-up Ansatz:

REST Schnittstelle implementieren

Definition REST-Schnittstelle generieren

REST-Webservices mit JAX-RS.

<http://www.example.com/hello/Josef?salutation=Servus>



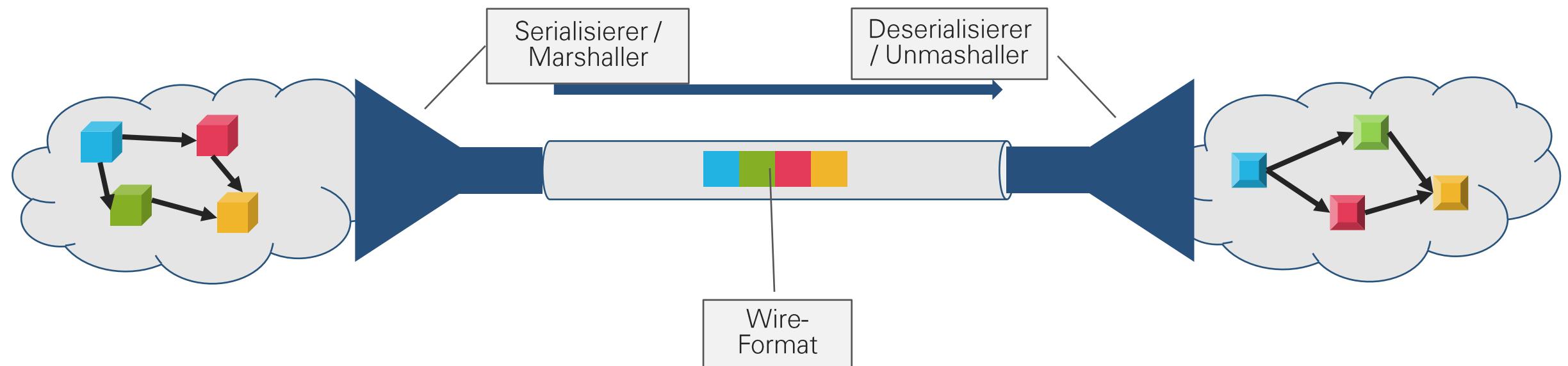
```
Request Path  
@Path("/hello/{name}")  
public class HelloWorldResource {  
    @GET, @POST, @PUT, @DELETE  
    @GET  
    @Produces("application/json")  
    public ResponseMessage getMessage(  
        @DefaultValue("Hallo") @QueryParam("salutation") String salutation,  
        @PathParam("name")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

Annotations and their descriptions:

- Request Path**: Points to the path parameter in the URL (`{name}`) and the `@PathParam("name")` annotation in the code.
- @GET, @POST, @PUT, @DELETE**: Points to the method annotations in the code.
- @GET**: Points to the `@GET` annotation in the code.
- Analog @Consumes für 1. Parameter**: Points to the `@Produces("application/json")` annotation in the code.
- Analog @FormParam bei POST Requests**: Points to the `@QueryParam("salutation")` annotation in the code.

Für XML und JSON Schnittstellen benötigen wir immer Serialisierung und Deserialisierung.

Die **Serialisierung** ist [...] eine Abbildung von strukturierten Daten auf eine sequenzielle Darstellungsform. Serialisierung wird hauptsächlich für die Persistierung von Objekten in Dateien und für die Übertragung von Objekten über das Netzwerk bei verteilten Softwaresystemen verwendet.



Marshalling (englisch *marshal*, ‚aufstellen‘, ‚ordnen‘) ist das Umwandeln von strukturierten oder elementaren Daten in ein Format, das die Übermittlung an andere Prozesse ermöglicht

- Wikipedia

JSON-B und JSON-P

Was ist JSON-B?

Datenformat	Text (JSON)
Aufwand	Keiner
Objektgraph	Zyklusfrei

„JSON-B is a standard binding layer for converting Java objects to/from JSON messages“

- In Java EE 8 enthalten
- Einfache API zum Konvertieren von Java Objekten nach JSON und umgekehrt
- In den allermeisten Fällen: „good enough“

Roundtrip Beispiel

```
@Data // Lombok-Annotation. Liefert Getter, Setter und Konstruktor
public class Book {
    private String id;
    private String title;
    private String author;
}
```

```
Book book = new Book("ABCD-1234", "Fun with Java", "Alex Theedom");
```

```
Jsonb jsonb = JsonbBuilder.create();
```

```
String json = jsonb.toJson(book);
```

```
Book book = jsonb.fromJson(json, Book.class);
```

```
{
    "author": "Alex
Theedom",
    "id": "ABCD-1234",
    "title": "Fun with Java"
}
```

Und was ist dann JSON-P?

Datenformat	Text (JSON)
Aufwand	Manuell
Objektgraph	Zyklusfrei

„JSON Processing (JSON-P) is a Java API to process [...] JSON messages. It produces and consumes JSON text in a streaming fashion [...] and allows to build a Java object model for JSON text [...].“

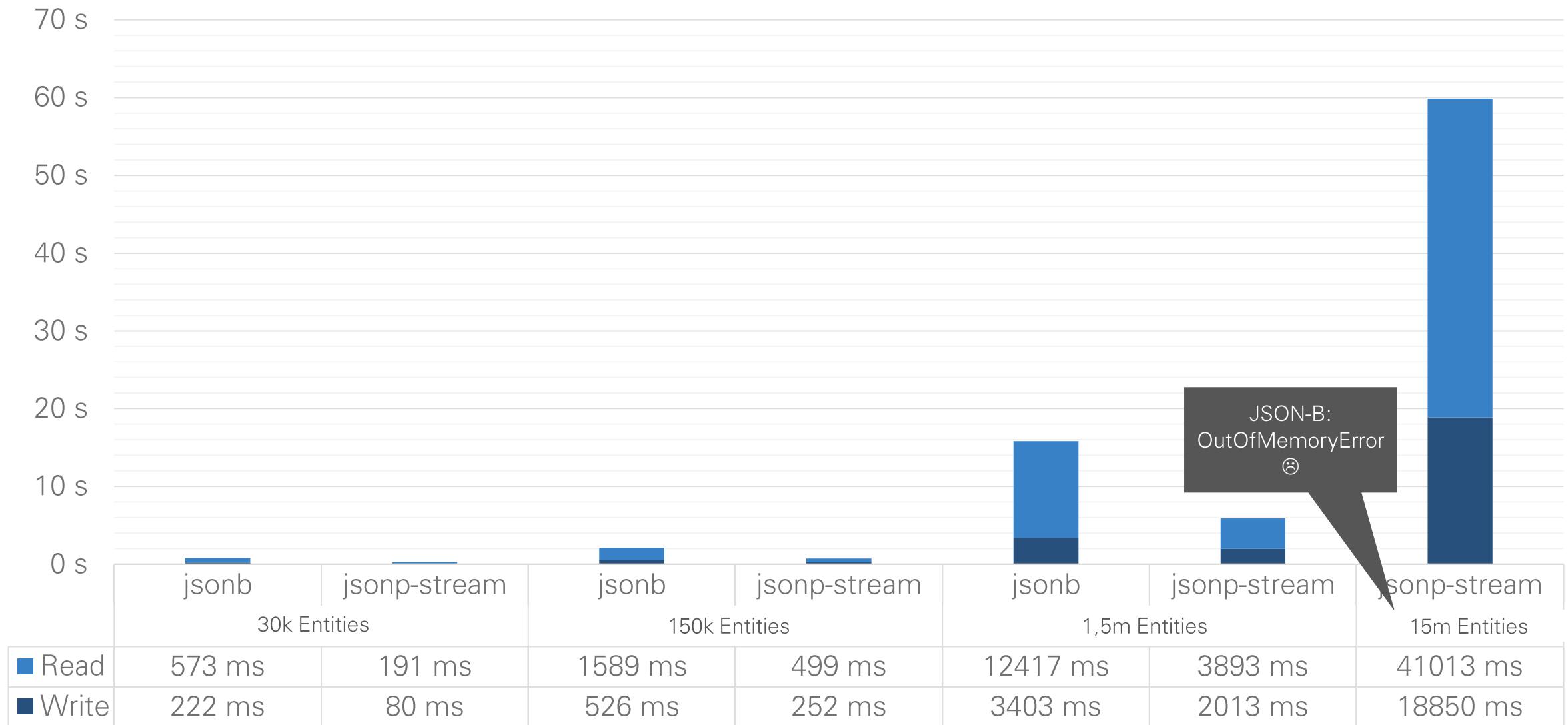
- Ebenfalls in Java EE enthalten
- Mehr low-level als JSON-B

```
{  
    "name": "Rex",  
    "race" : "dog",  
    "age": 3,  
    "biteable": false  
}
```

```
JsonObject json = Json.createObjectBuilder()  
    .add("name", "Rex")  
    .add("race", "dog")  
    .add("age", BigDecimal.valueOf(3))  
    .add("biteable", Boolean.FALSE).build();  
String result = json.toString();
```

```
JsonParser parser = Json.createParser(new  
StringReader(result));  
String key = null, value = null;  
while (parser.hasNext()) {  
    final Event event = parser.next();  
    switch (event) {  
        case KEY_NAME:  
            key = parser.getString();  
            System.out.println(key);  
            break;  
        case VALUE_STRING:  
            value = parser.getString();  
            System.out.println(value);  
            break;  
    }  
}  
parser.close();
```

Performance JSON-B und JSON-P (streaming)



Binärprotokolle

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";

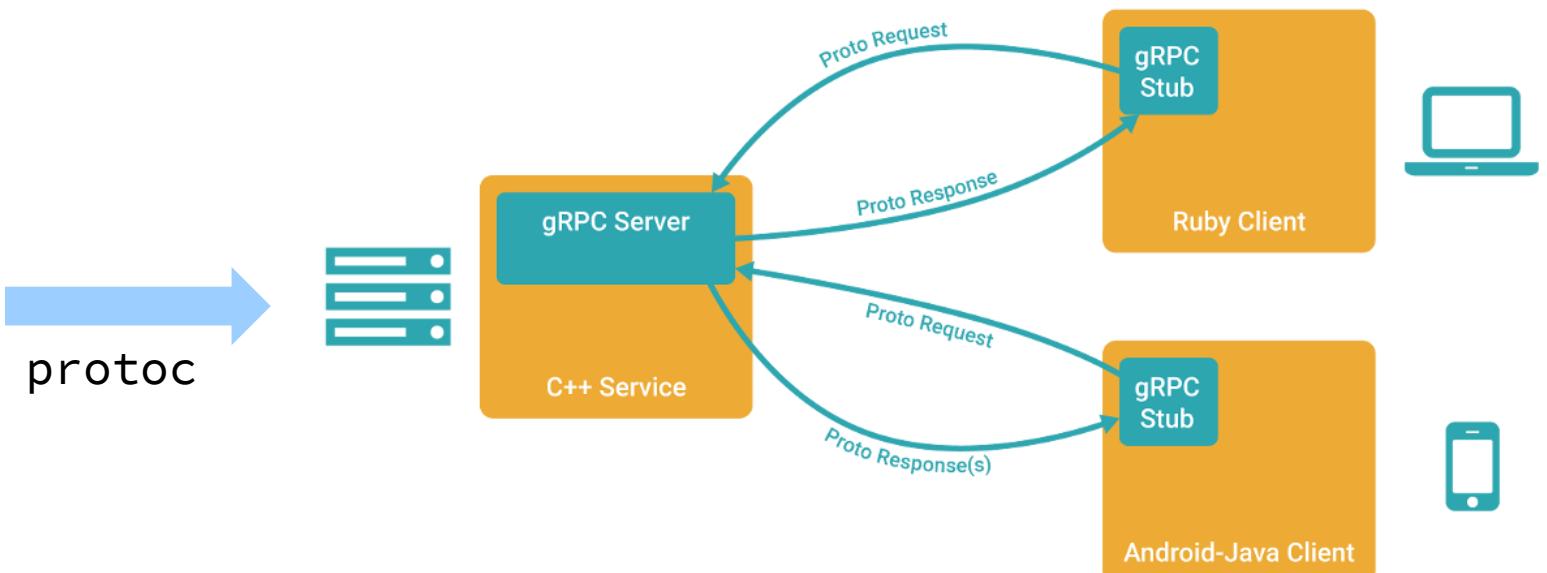
option java_package = "io.grpc.examples";

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;
}

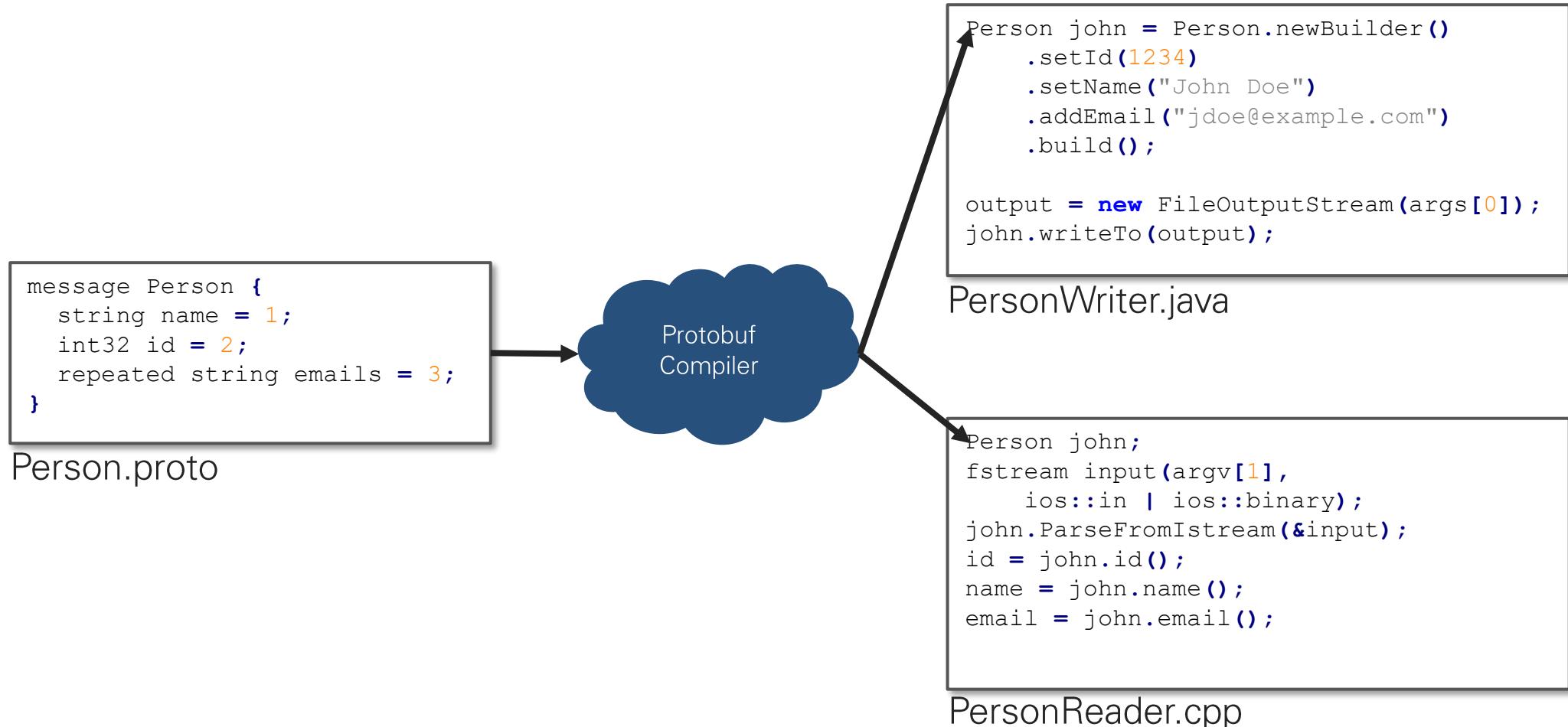
    // The response message containing the greetings
message HelloReply {
    string message = 1;
}
```



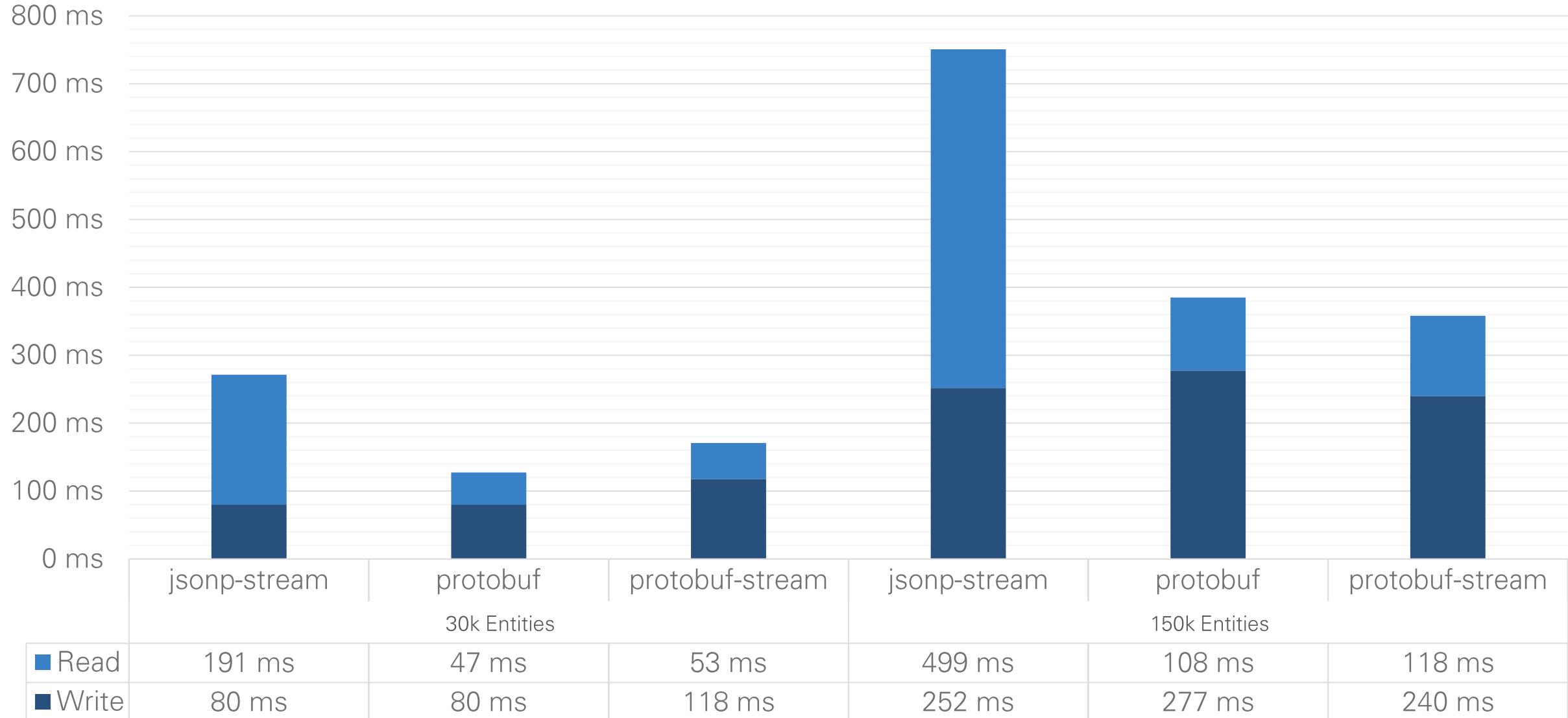
Protobuf 101

Datenformat	Binär, sprachneutral
Aufwand	Schema
Objektgraph	Zyklusfrei

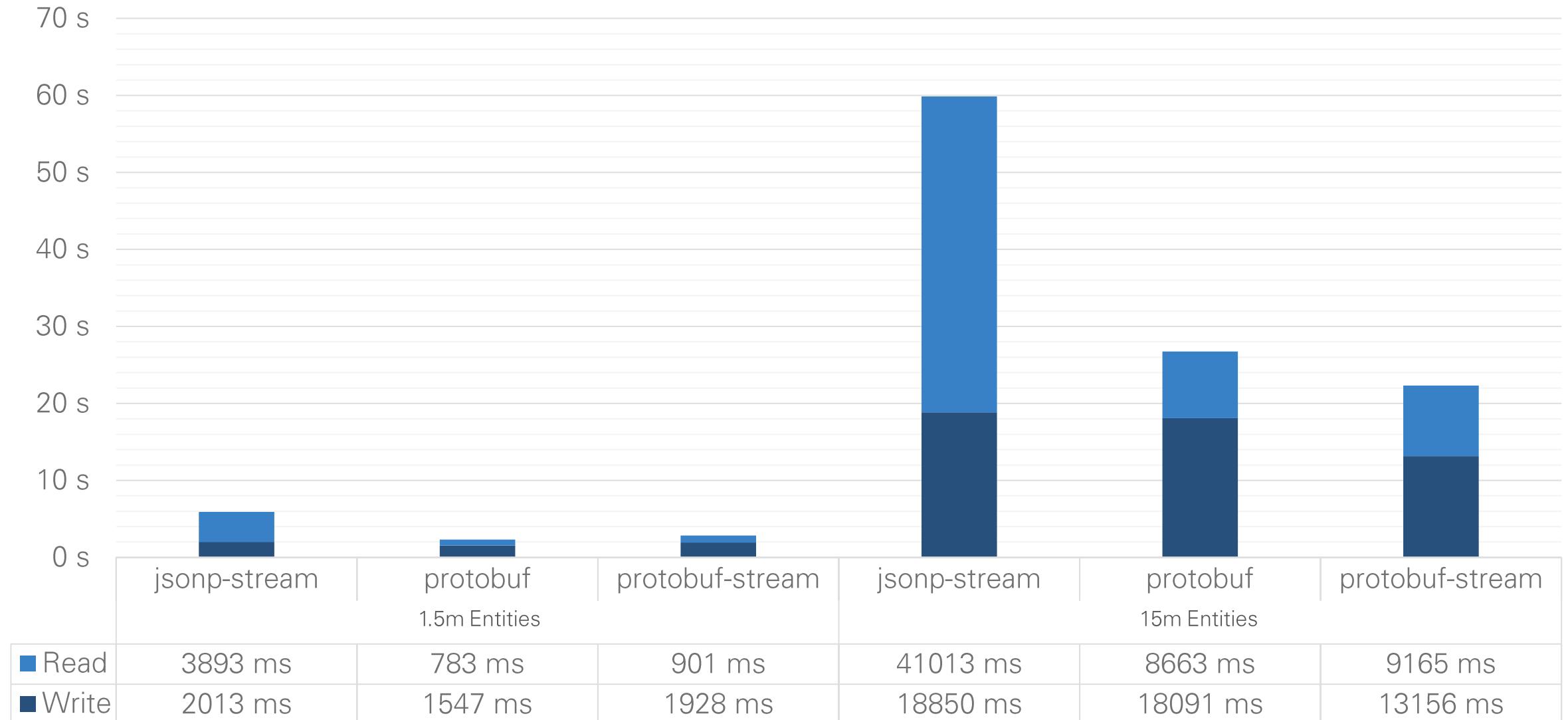
"Protocol buffers are Google's language-neutral, platform-neutral, extensible mechanism for serializing structured data – think XML, but smaller, faster, and simpler."



Performancevergleich – kleine Requests



Performancevergleich – große Requests



Flexible Kommunikationsmuster mit Messaging

Messaging ist zuverlässiger, asynchroner Nachrichtenaustausch.



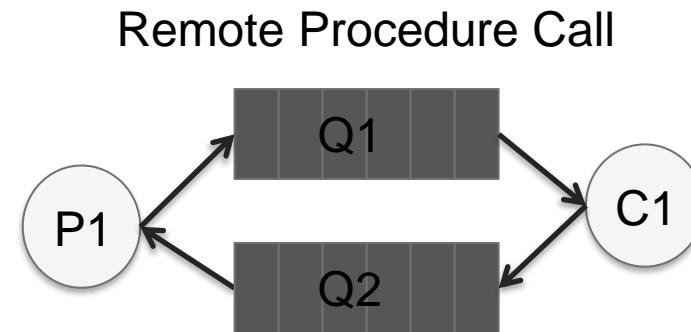
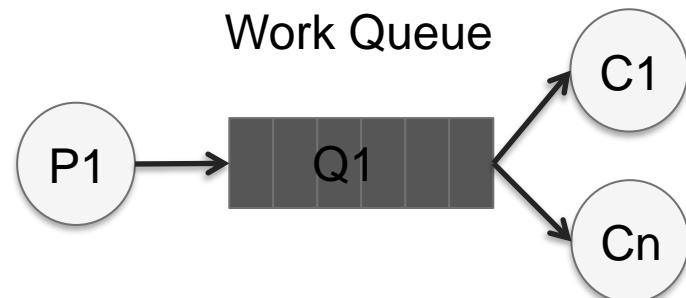
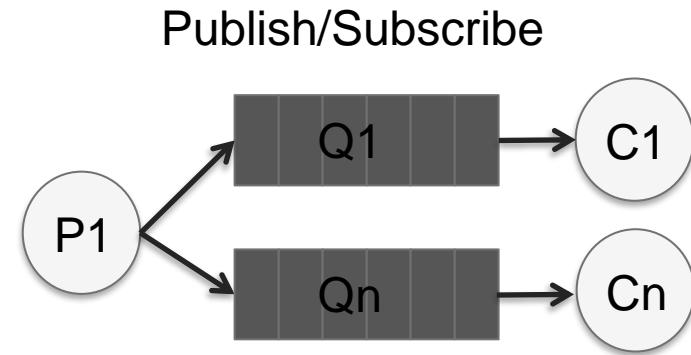
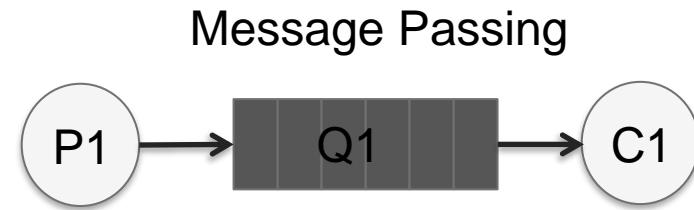
Entkopplung von Producer und Consumer.

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.



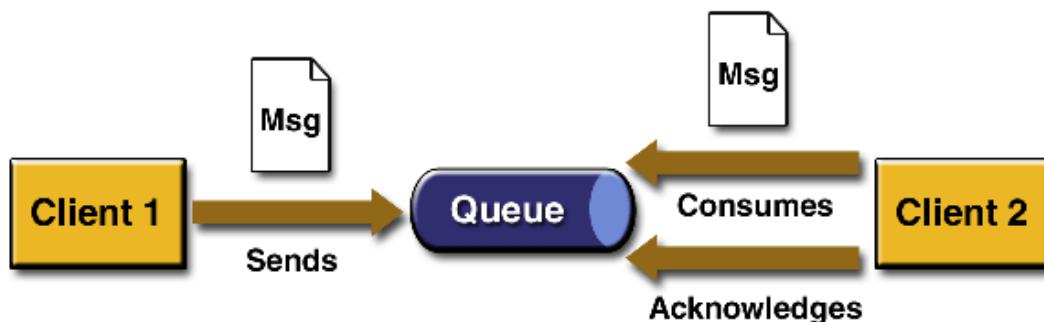
JMS

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

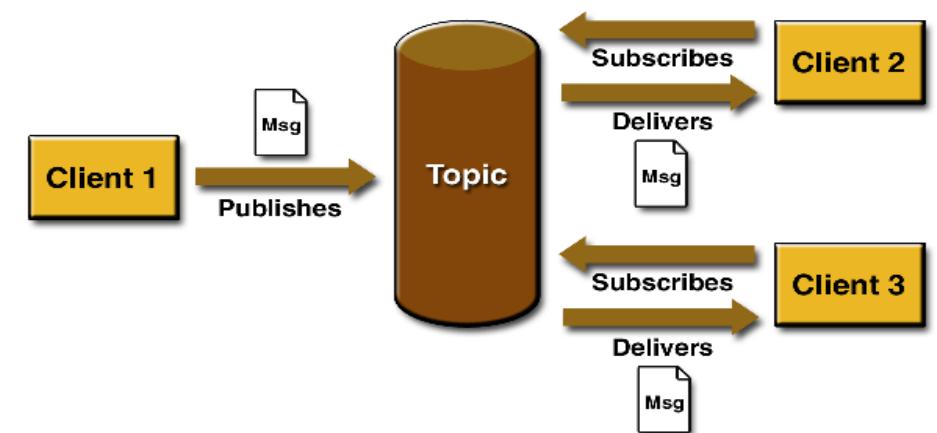
- 2002-2013: Version 1.1. Sehr stabil und weit verbreitet in der Java-Welt.
- Seit Mai 2013: Version 2.0 als Teil der JEE 7 Spezifikation

Unterstützte Kommunikationsmuster:

Message Passing:



Pub/Sub:



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

- Mehrere Consumer pro Message

AMQP: Ein Standard-Protokoll für Messaging-Systeme.

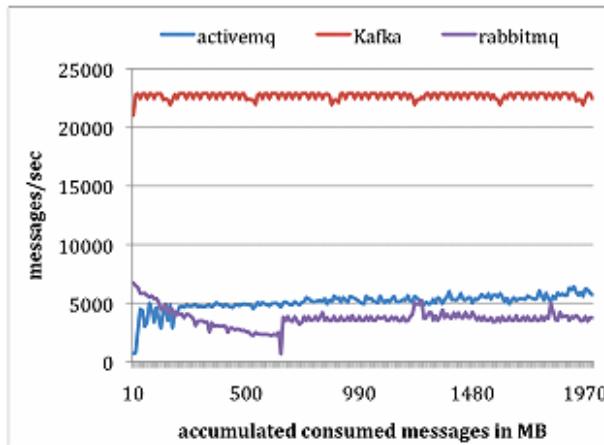
- **Problem:** Message Broker sind intern proprietär aufgebaut (Beispiel: IBM MQSeries mit 80% Marktanteil im kommerziellen Bereich). Sie sind nicht zueinander interoperabel, wie man es z.B. von SNMP-Servern her kennt. Das ist besonders beim Messaging über Firmengrenzen und Technologie-Stacks hinweg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

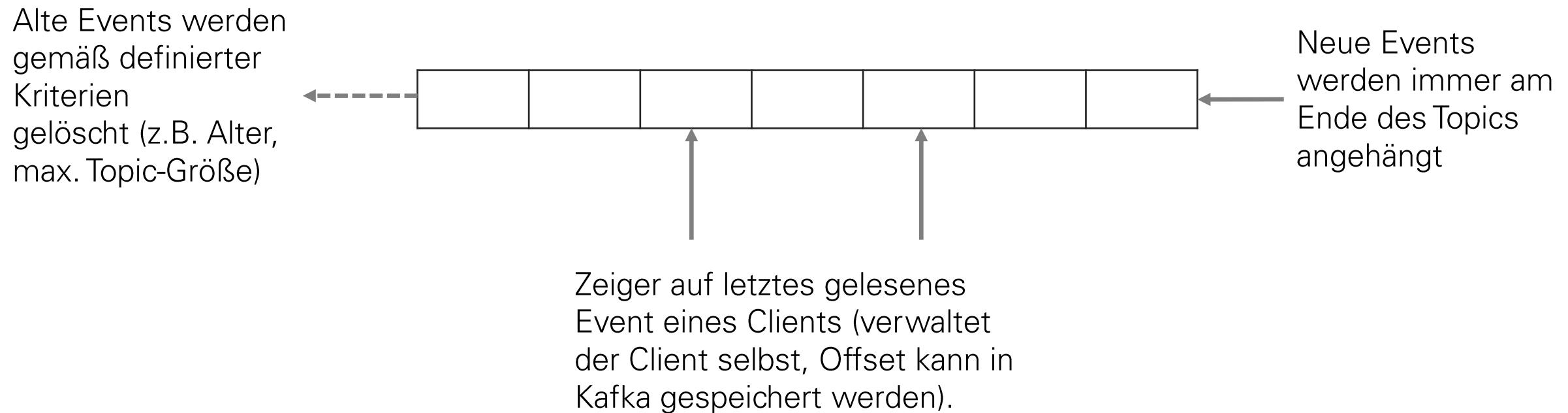
Kafka

- 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:

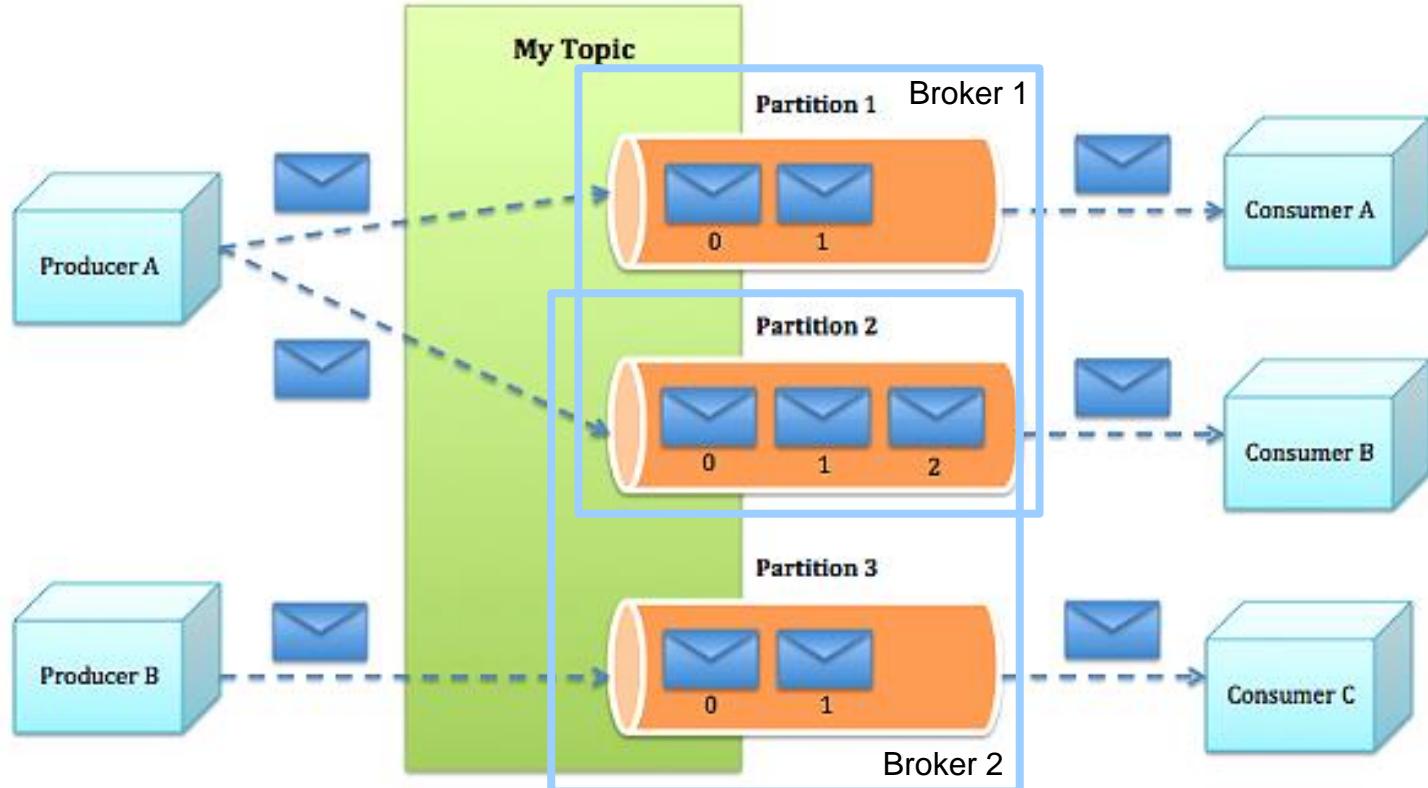


- Kafka ist so schnell, da es Betriebssystem-Mittel intelligent nutzt, ein effizientes Codierungsformat für Nachrichten besitzt und den Auslieferungszustand in den Clients hält.
- Kafka ist in Java und Scala geschrieben. Die Kafka API ist proprietär und orientiert sich an keinem Messaging-Standard.

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



Der Event-Log in Kafka ist hochgradig verteilt.



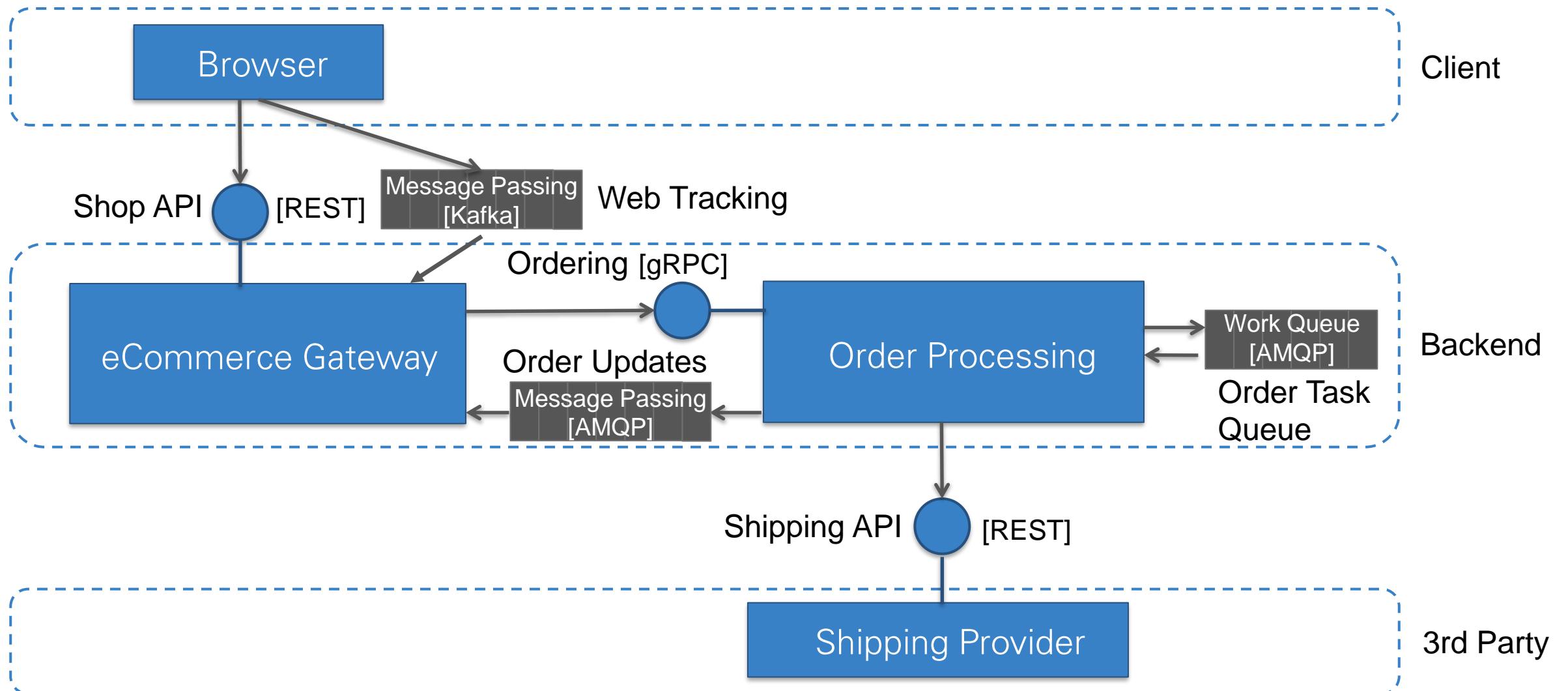
- Die Events in einem Topic werden aufgeteilt in Partitionen
- Die Partitionen werden verteilt auf die verfügbaren Broker-Instanzen
- Partitionen werden zur Fehlertoleranz repliziert

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>

Architekturaspekte

Putting it all together...



Literatur

Literatur

Bücher:

- Patterns of Enterprise Application Architecture, Martin Fowler, 2002
- Computer Networks, Andrew Tanenbaum, 2010
- Inter-Process Communication, Hephaestus 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 with and Spring Boot

Project Metadata

Artifact coordinates

Group

Artifact

Dependencies

Add Spring Boot Starters and dependencies to your application

Search for dependencies

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.

The screenshot shows the Swagger UI interface for a REST API. At the top, there is a green header bar with the 'swagger' logo, the URL 'http://localhost:8080/api/swagger.json', and a 'Explore' button. Below the header, a small '1.0.1' badge is visible. The main content area displays the 'books' endpoint documentation. It includes a dropdown menu for 'Schemes' set to 'HTTP' and a '▼' button. The 'books' endpoint is shown with five operations: 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), and POST /books (Create book). Each operation is represented by a colored box (blue for GET, orange for PUT, red for DELETE, blue for GET, green for POST) containing the method, verb, and path.

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