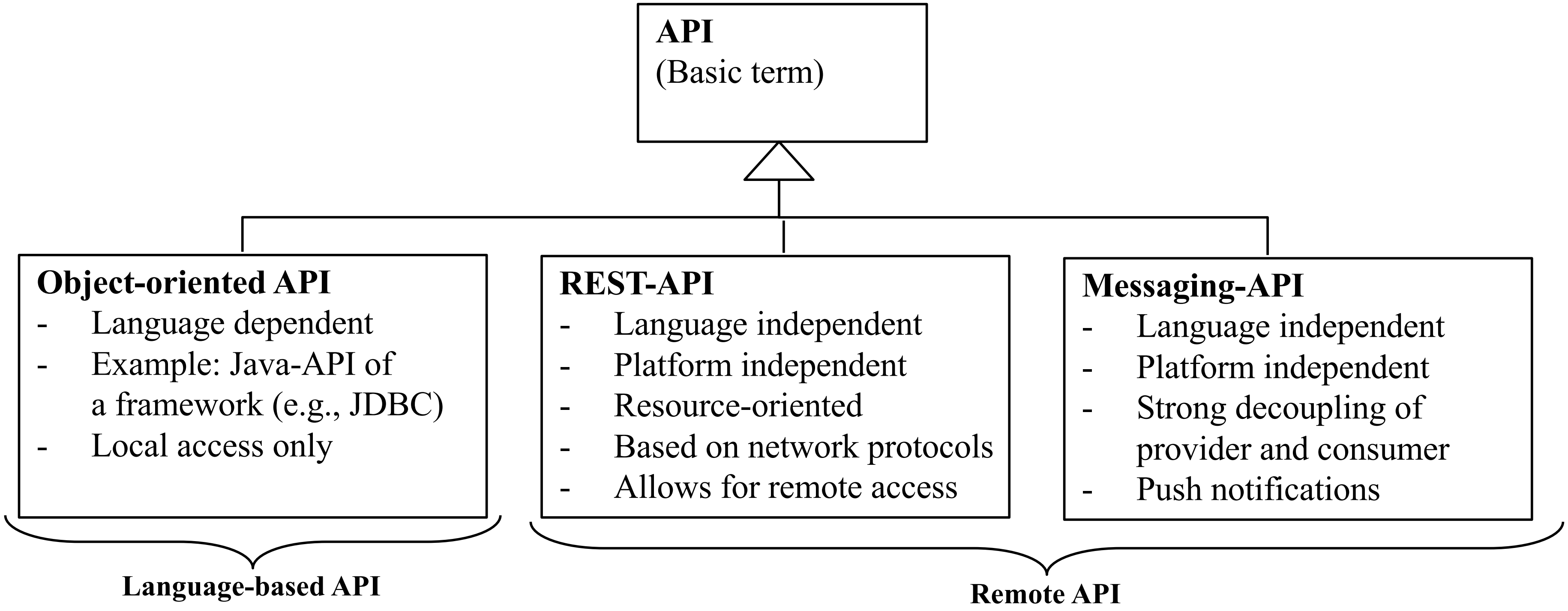
Structure of this Module

Chapter 2: Introduction to Software Architectures and Architectural Integration		
1	Motivation – Software Development Lifecycle	✓
2	Definition and Properties of a Software Architecture	✓
3	Architectural Integration: Basic concepts and assumptions	✓
4	Modelling of Software Architectures with UML (4 Views Model)	✓
5	Revisiting a fundamental Architecture Pattern: Layer	✓
6	**API Design: Basic concepts**	
7	The REST architectural style: Basic concepts	
8	Modularization in JavaScript: Introduction to Node.js	

Definition API

- An API specifies the operations as well as the input and output data of a software component. The core idea is to have a set of functions independent of their implementation. If the implementation changes, then this will have no effect on the consumers of the software components. (Spichale, 2019)
- A good API accomplishes both simple reuse and easy integration!
- Distinction Language-based API vs. Remote API (on the basis of (Spichale, 2019))



API
(Basic term)

Object-oriented API
- Language dependent
- Example: Java-API of a framework (e.g., JDBC)
- Local access only

REST-API
- Language independent
- Platform independent
- Resource-oriented
- Based on network protocols
- Allows for remote access

Messaging-API
- Language independent
- Platform independent
- Strong decoupling of provider and consumer
- Push notifications

Language-based API

Remote API

Public APIs

- Today, many vendors and communities published their APIs that allows for consuming their services by 3rd parties. Prominent examples:
  - Facebook API (https://developers.facebook.com/docs/apis-and-sdks)
  - Twitter REST API (https://developer.twitter.com/en/docs)
  - Google Maps API (https://developers.google.com/maps/documentation/?hl=de)
  - Spring Initializer API (https://github.com/spring-io/initializr)
  - OpenFIGI API (https://www.openfigi.com/api)

- Based on the idea of publishing APIs, complete platform were created (PaaS (Platform as a Service) for hosting and accessing services
  - Amazon Web Service (AWS, cf.: https://aws.amazon.com/de/)
  - Salesforce.com (https://developer.salesforce.com/)

- Mass integration of small hardware devices (Internet Of Thing (IoT))

Last check of links: Oct 21st, 2019

A few heuristics on developing interfaces

- Apply the „need-to-know“ principle (information hiding)
  - Only when an information / a functionality is required to the consuming client class, it will become part of the interface
  - Avoid sending critical data to clients
  - Develop classes for merge coherent data (Data transfer object (DTO))

- Access subsystems based on a minimal set of well-defined methods (channels) that can be verified and controlled (no back doors!)
  - Inconsistencies may occur when using back doors

- Think about CRUD mechanisms for fully accessing and controlling your entity objects:
  - **C**(reate) an object (or: add)
  - **R**(ead) an object
  - **U**(pdate) an object
  - **D**(elete) an object
- Mostly used in persistence mechanisms, e.g., for storing business objects

A few heuristics on developing interfaces

- The semantic (intention) of a method should be made clear by the signature:
  - Important: Use business terms, no technical vocabulary
  - Good example: `bookHotel ( hotelname: String, dur: Duration);`
  - Bad example: `startBatchJob ( d : Data, s : int, e : int );`

- Avoid „Chatty“ Interfaces (Anti Pattern)
  - Chatty interfaces offer to exchange a lot of small data of primitive types
  - The chatty interface is also characterized by a high number of method invocations.
  - Result: God father object (too many dependencies)

- A interface should also be (Spichale, 2019):
  - Consistent (always apply same patterns for describing similar tasks)
  - Intuitive (`delete()` vs. `erase()` → which is more intuitive…?)
  - Documented (e.g., with tool Swagger or JavaDoc)
  - Easy to learn (with a low entry threshold, easy coding examples)
  - Arranged, so that it hard to make mistakes (cf. Class `Date` in Java)
  - Extendable

Best Practices (Starke, 2015)

- Always keep track of possible exceptions that may arise
- Often, only authenticated and authorized consumers are allowed to access an API
  - Security check must be utilized before and should be inserted in the API
  - Authentication is the process of ascertaining that somebody really is who they claim to be. (Implementation in practice: login mechanisms, SSO, Web token, HTTPs)
  - Authorization refers to rules that determine who is allowed to do what. (Implementation in practice: tables (realms) for operations and users)
- In case of sensitive information, data must be encrypted before sending
- Provider must often give guaranties (Quality of Service (QoS)) regarding throughput, performance, or availability. A continuous inspection and improvement is necessary!
- In commercial environments, each access to an API results in costs (per access or per volume). Keep track of consumers and their costs in a reliable manner!
  - Allow for inspection to your consumers to check current balance (e.g. Dashboard)

Structure of this Module

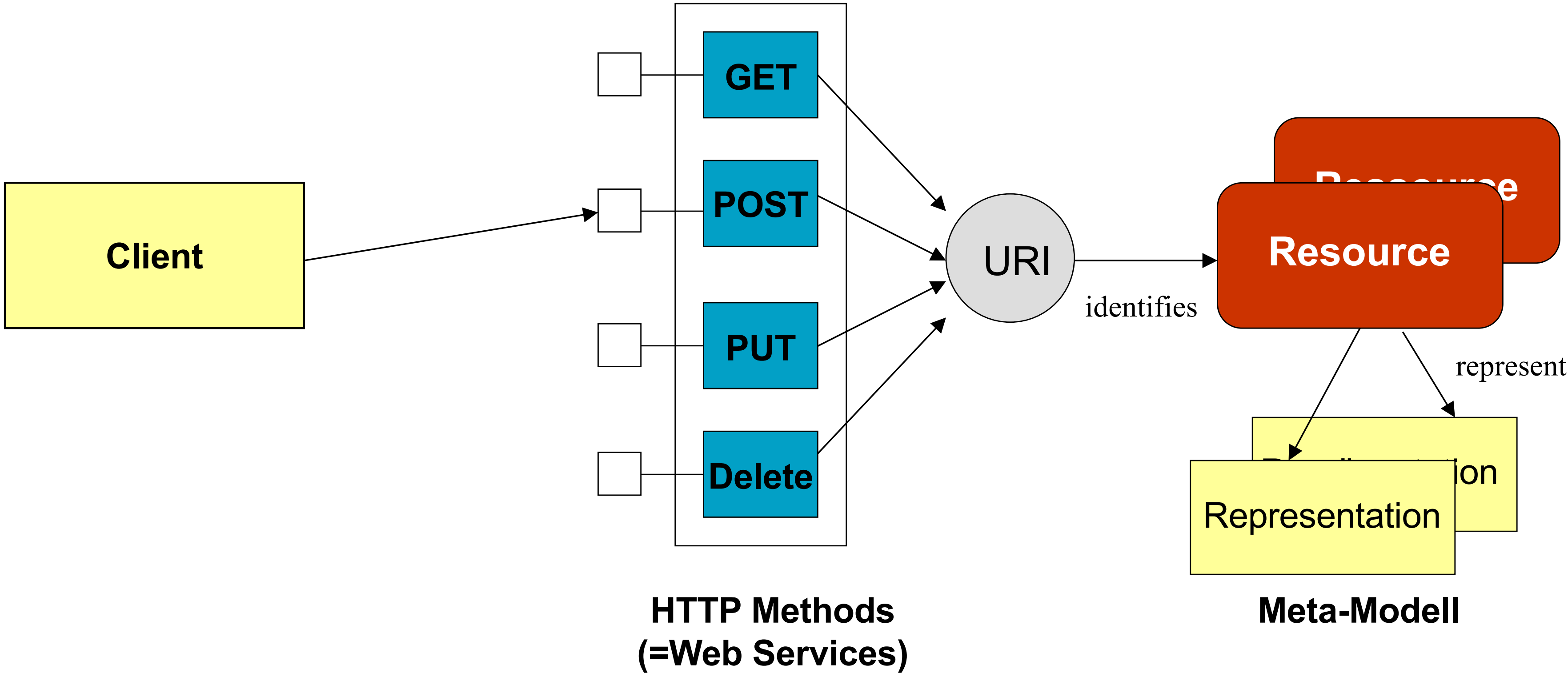
Chapter 2: Introduction to Software Architectures and Architectural Integration		
1	Motivation – Software Development Lifecycle	✓
2	Definition and Properties of a Software Architecture	✓
3	Architectural Integration: Basic concepts and assumptions	✓
4	Modelling of Software Architectures with UML (4 Views Model)	✓
5	Revisiting a fundamental Architecture Pattern: Layer	✓
6	API Design: Basic concepts	✓
7	**The REST architectural style: Basic concepts**	
8	Modularization in JavaScript: Introduction to Node.js	

REST (REpresentational State Transfer)

- REST represents an architectural style for implementing Web Services (or: a Remote API that can be accessed remotely)
- Based on the doctoral thesis of Roy Thomas Fielding (Fielding, 2000)
  - http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm

- Approach:
  - Accessing a Web Service by the explicit usage of the HTTP-protocol (the standard today, however, not explicitly stipulated by (Fielding, 2000), other protocols are conceivable)
  - The adoption of HTTP as the underlying protocol is also referred as RESTful HTTP (Tilkov, 2015)
  - However, not every HTTP-based API is conform to REST (Spichale, 2019)
  - Resource-based view. Resources may use hypermedia (Spichale, 2019)
  - REST also makes assumptions on further architectural elements (e.g., cache, stateless server) → not handled in this lecture (cf. Master CS)
  - Focus in this lecture: Adoption of REST as an API-style

Structure of a REST-based Architecture



- Examples of a resource: Web page, functions, pictures, documents
- A resource can have various representations (e.g., JSON (= standard), XML)
- A resource together with a corresponding representation can be uniquely *identified* by a URI (Uniform Resource Identifier)
- Primarily accessing resources by four HTTP methods (other methods are applicable). HTTP methods fulfill the CRUD pattern

JSON - JavaScript Object Notation

- JSON is is a lightweight data-interchange format that can be parsed and consumed easily by modern programming languages
  - Source: http://www.json.org/

- JSON is built on two structures:
  - A collection of name/value pairs. In various languages, this is realized as an *object*, record, struct, dictionary, hash table, keyed list, or associative array.
  - An ordered list of values. In most languages, this is realized as an *array*, or sequence.

- Thus, two major elements are given: Objects and Arrays. Objects can be *nested*:

```
{ “employee” : { “Name“ : “Kaiser“ }}
```

= an Object with one attribute

```
{ “employee” : { “First Names“ : [ “Axel“, “Torben“ ]  } }
```

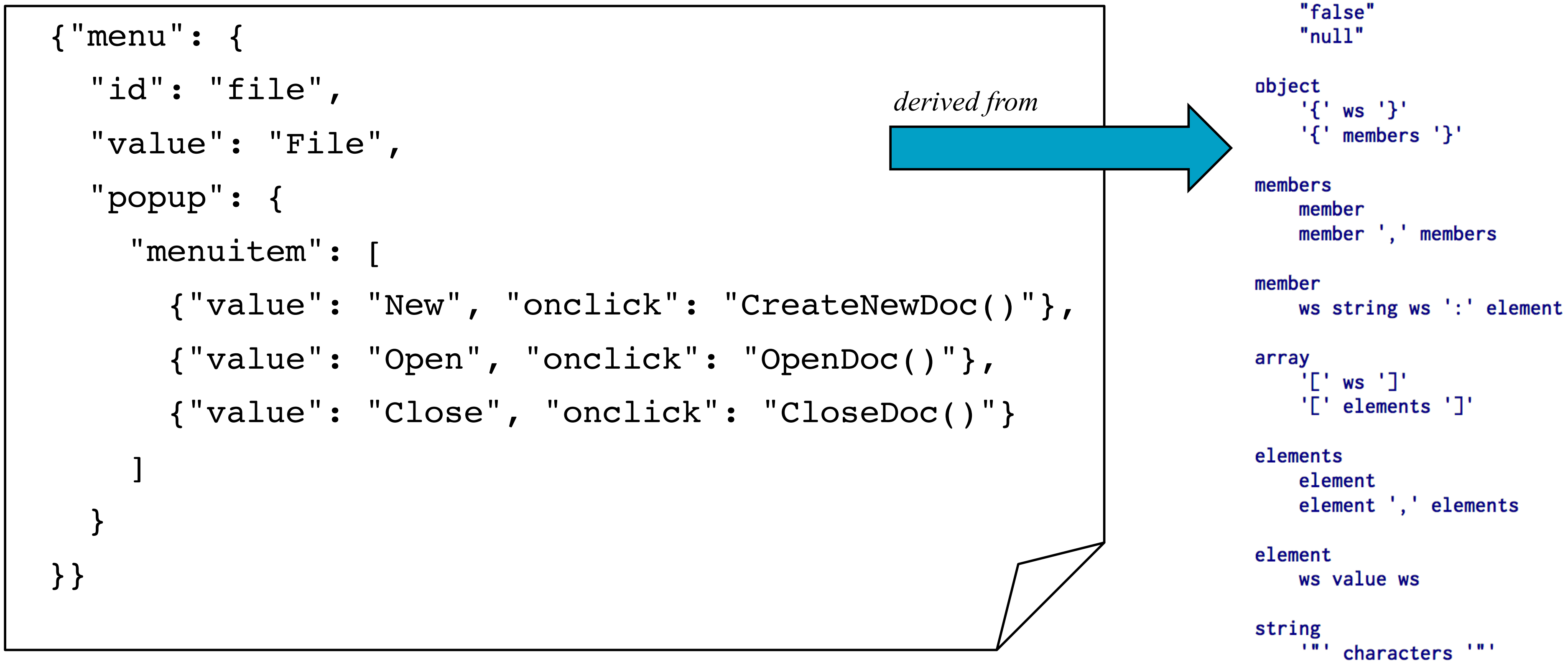
= Array with two elements

```
{ “employee” : { “ID“ : 123, job : { “Titel“ : “Manager“, “id“ : 12 } } }
```

= a nested sub object

A bit more complex example, Relation to Grammar

- For a JSON document, a clear grammar is given.
- Further examples can be studied here:
  - http://json.org/example.html



```
{"menu": {
  "id": "file",
  "value": "File",
  "popup": {
    "menuitem": [
      {"value": "New", "onclick": "CreateNewDoc()"},
      {"value": "Open", "onclick": "OpenDoc()"},
      {"value": "Close", "onclick": "CloseDoc()"}
    ]
  }
}}
```

derived from

```
json
    element

value
    object
    array
    string
    number
    "true"
    "false"
    "null"

object
    '{' ws '}'
    '{' members '}'

members
    member
    member ',' members

member
    ws string ws ':' element

array
    '[' ws ']'
    '[' elements ']'

elements
    element
    element ',' elements

element
    ws value ws

string
    '"' characters '"'
```

Example based on JSON

Underlying grammer (excerpt)

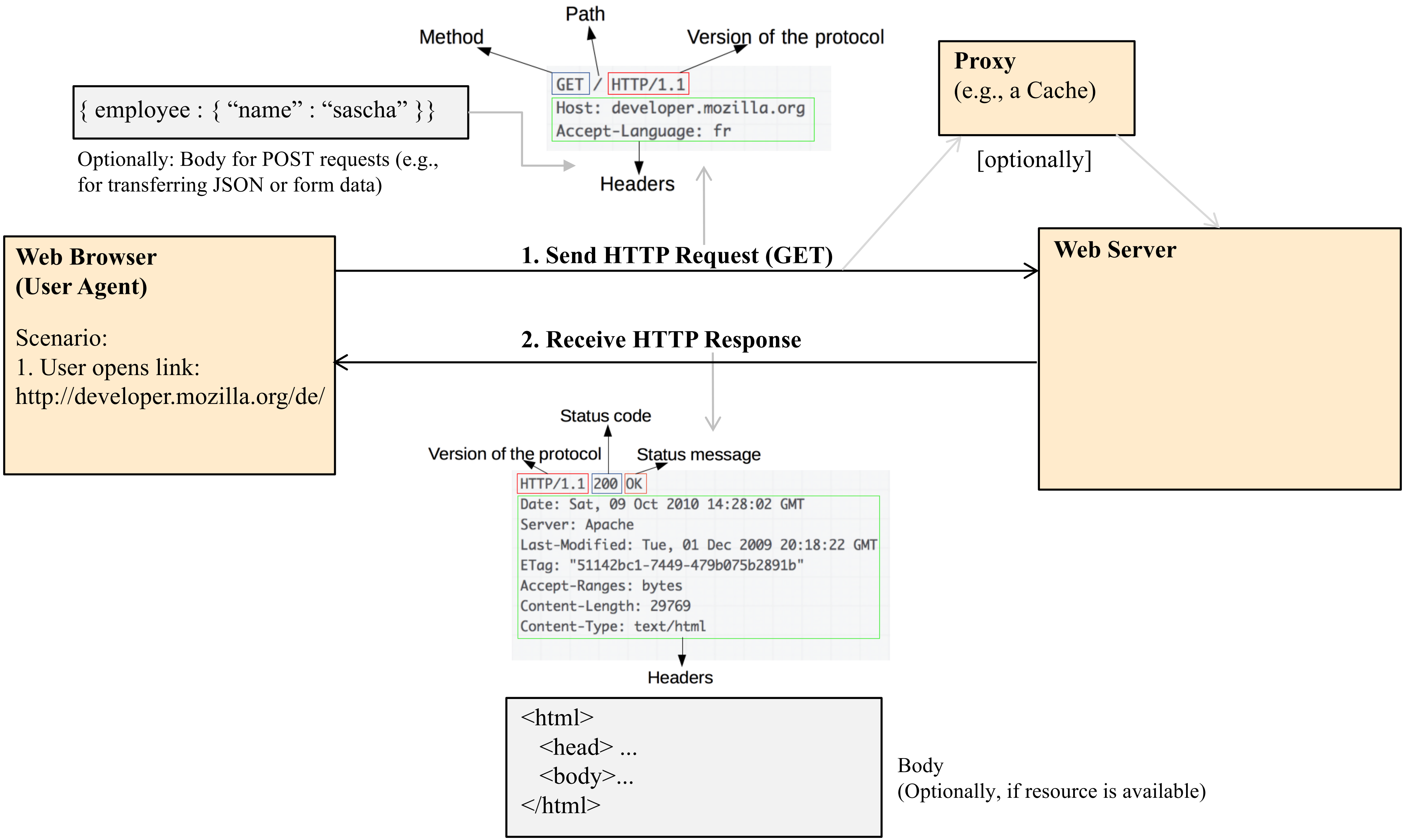
A short introduction to HTTP

- HTTP is a protocol which allows the fetching of resources (e.g., HTML documents). It is the foundation of any data exchange on the Web and it is a client-server protocol, which means requests are initiated by the recipient, usually the Web browser
- Data is conveyed in terms of HTTP messages, requests and responses.
- HTTP makes use of various (request) methods for initiating requests

- GET
  - Used for accessing (reading) data of content from a Web server (e.g., a static web page)
  - Data can be passed at the URL. Example:
    `action_page.php?firstname=Mickey&lastname=Mouse`
  - Passing of non-sensitive data only! Mostly, the length of a URL is limited (ca. 2000 symbols)
- POST
  - Used for passing data (for either creating or updating)
  - Data is encapsulated in the HTTP request body with a specific content–type (e.g., JSON)
  - Passing of sensitive data, cannot be read from the URL!
  - No limitation on the size. Often used for passing form-data

Source: https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview

The underlying software architecture of the HTTP protocol



```
GET / HTTP/1.1
Host: developer.mozilla.org
Accept-Language: fr
```

```
{ employee : { “name” : “sascha” }}
```

Optionally: Body for POST requests (e.g., for transferring JSON or form data)

Scenario:
1. User opens link: http://developer.mozilla.org/de/

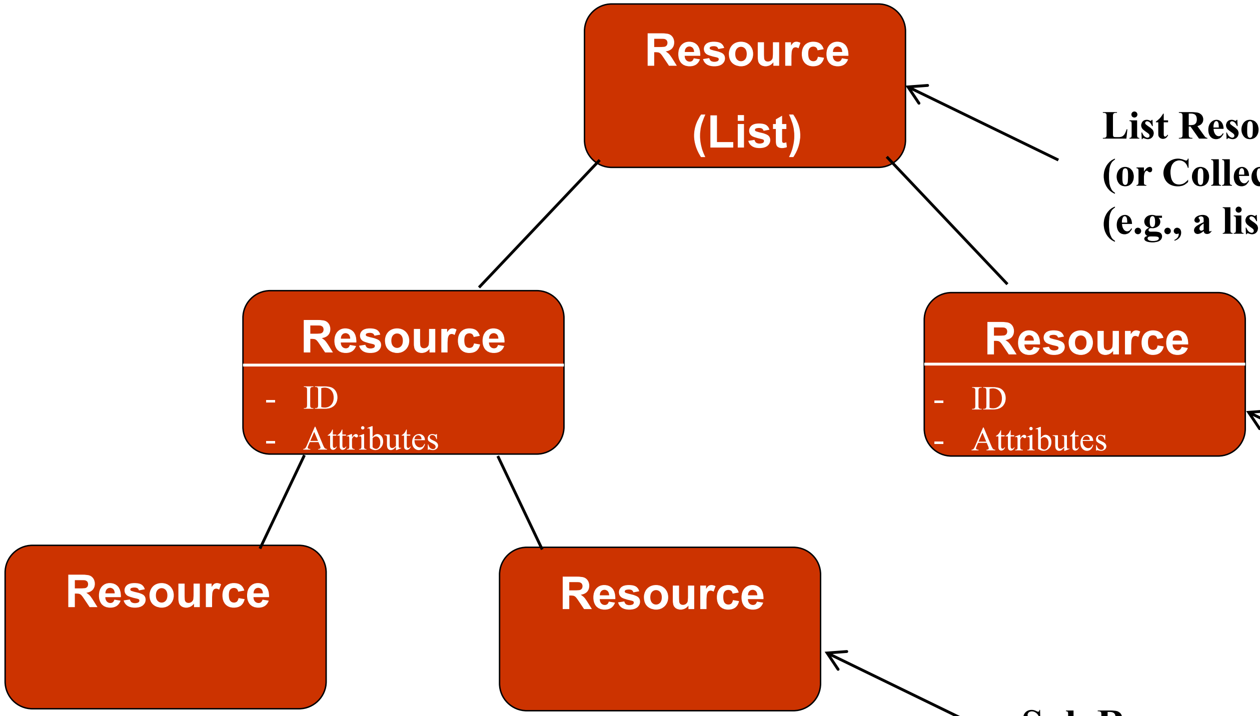
```
HTTP/1.1 200 OK
Date: Sat, 09 Oct 2010 14:28:02 GMT
Server: Apache
Last-Modified: Tue, 01 Dec 2009 20:18:22 GMT
ETag: "51142bc1-7449-479b075b2891b"
Accept-Ranges: bytes
Content-Length: 29769
Content-Type: text/html
```

```
<html>
  <head> ...
  <body>...
</html>
```

Body
(Optionally, if resource is available)

Hierarchical model of REST-based resources

- In REST, resources can be arranged in a hierarchical order
- A REST tree basically consists of List-based resources and primary resources (Tilkov et al., 2015). Further resource types are possible (Sub Resource (Spichale, 2019))
- Both the structure and the naming of the resources must be analyzed and designed adequately (Spichale, 2019, Chapter 9)



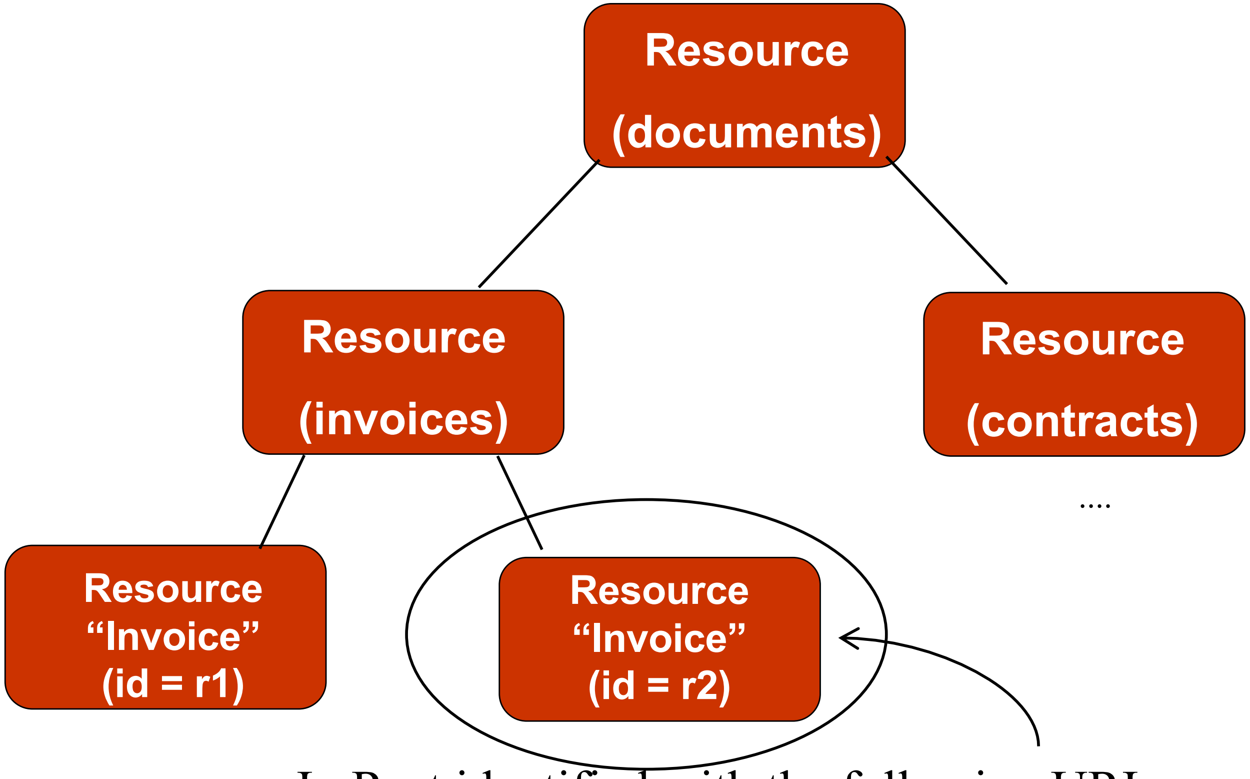
List Resource
(or Collection Resource)
(e.g., a list of clients)

Primary Resource
(e.g., a Client)

Sub Resource (e.g., an address of a client)

Addressing Resourcing in REST

- In REST, resources can be arranged in a hierarchical order. Given the hierarchical order, they can be identified and, thus, accessed by an URI.
- Identification is mastered by appending a unique ID (identifier) to the URI
  - Best Practice: Name of the Resource within the URI in plural

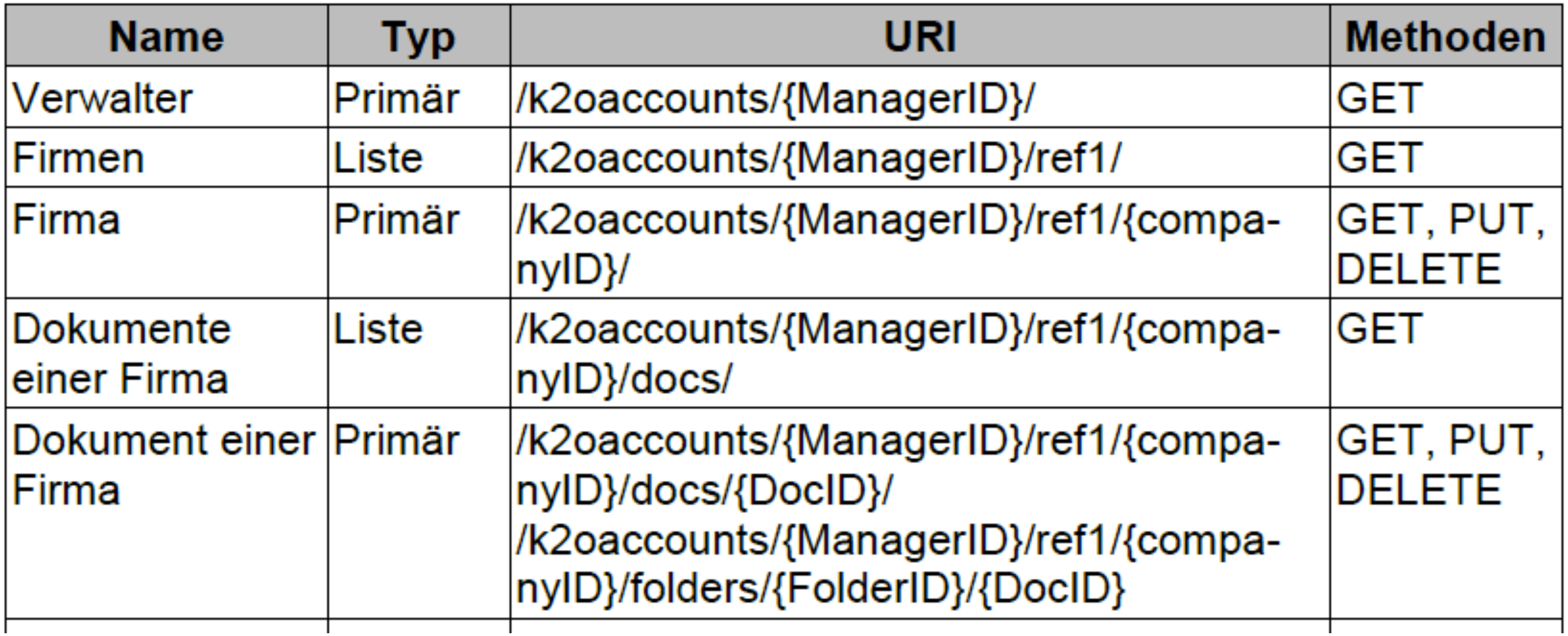


In Rest identified with the following URI:

$\underbrace{\text{http://myserver/documents/invoices/r2}}_{\text{URI}}$

URIs in REST – a common template for structuring URIs

- URIs are often structured in a table-based view
- Definition of routes to the resources
- Terms that are enclosed in {curly brackets} represent the key to the primary resource (in practice: further attributes are possible (e.g., a search key))



Name	Typ	URI	Methoden
Verwalter	Primär	/k2oaccounts/{ManagerID}/	GET
Firmen	Liste	/k2oaccounts/{ManagerID}/ref1/	GET
Firma	Primär	/k2oaccounts/{ManagerID}/ref1/{companyID}/	GET, PUT, DELETE
Dokumente einer Firma	Liste	/k2oaccounts/{ManagerID}/ref1/{companyID}/docs/	GET
Dokument einer Firma	Primär	/k2oaccounts/{ManagerID}/ref1/{companyID}/docs/{DocID}/ /k2oaccounts/{ManagerID}/ref1/{companyID}/folders/{FolderID}/{DocID}	GET, PUT, DELETE

Source: Projektarbeit Glowinski, 2011

Basic HTTP Methods (“Verbs”) in REST (Spichale, 2019)

- **GET**
  - Reading (or: querying) the representation of a resource
  - Secure query: GET does not change the internal status of a resource

- **POST**[1]
  - Creating (or adding) a new resource
  - Apply when the route (the URI) is unknown from the client perspective – server decides, where the resource is stored. Client receives the new URI as a response.

- **DELETE**
  - Deleting of a resource

- **PUT**[1]
  - Primary Usage: Updating of resource with a given know URI
  - Also: Adding a resource, where the route (URI) is *predetermined* by the client.

- Remark: PUT und POST are controversially discussed and interpreted, oftentimes.

Quelle [1]: http://restcookbook.com/HTTP%20Methods/put-vs-post/

Basic HTTP methods – Correspondence to an Object-Oriented API

Object-oriented API	RESTful HTTP
`getUsers()`	`GET /users`
`updateUser()`	`PUT /users/{id}`
`addUser()`	`POST / users`
`deleteUser()`	`DELETE / users / {id}`
`getUserRoles()`	`GET /users/ {id} / roles`

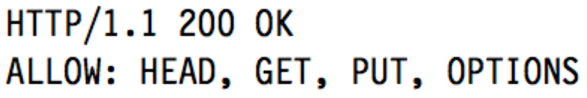
{ employee : { "name" : "sascha" }}

The new resource "user" is placed in a
given representation (here: JSON)
in the Body of the HTTP Request!

Source: (Spichale, 2019), Chapter 8

Further HTTP Methods in REST

- **OPTIONS**
  - Provided the possible communication options of a resource
  - Example:



```
HTTP/1.1 200 OK
ALLOW: HEAD, GET, PUT, OPTIONS
```

- **PATCH**
  - Changing a part of a resource (e.g., selected attributes, only)
  - Reduces the communication overhead

- **HEAD**
  - Similar to GET, however, the response has got no body (in REST: the actual representation will not be sent)
  - Client might check, if a resource is available at all. Also, client might check the size of a resource (e.g., of a Video)

REST for Java – JAX-RS

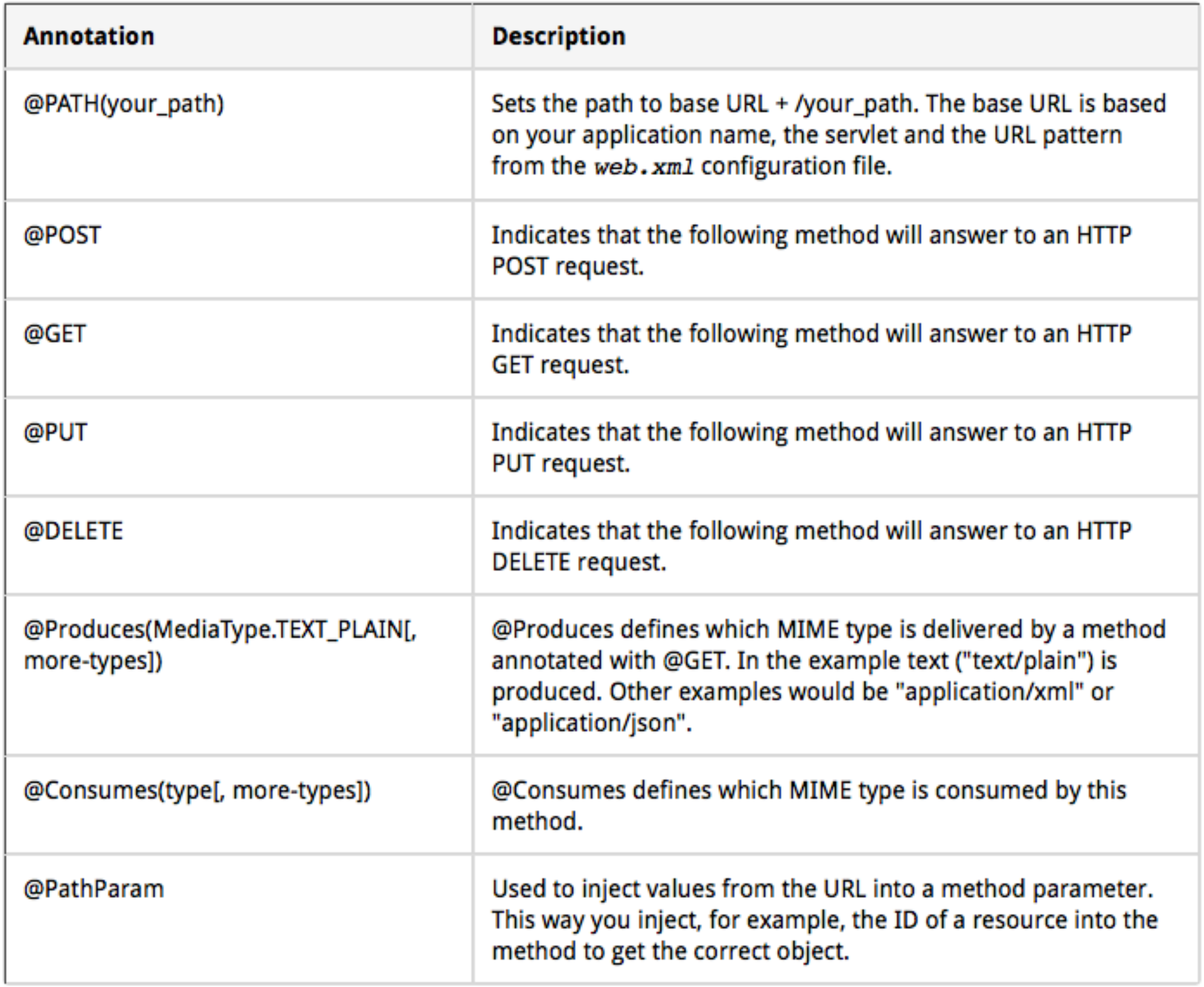
- REST support for Java has been defined by the JSR 370, which nowadays known as JAX-RS (Java API for RESTful Web Services).
  - Recent Version: 2.1 (July 2017, as part of Java EE 8)
  - Source: https://jcp.org/en/jsr/detail?id=370

- Based on annotations (see next slides): annotations are used to declare resources, methods for providing the HTTP methods, etc.

- Reference implementation: Jersey (2.28)
  - Source: https://jersey.java.net/

- Complex Development model:
  - Return values must be composed and read in a complex way.
  - Example: https://crunchify.com/how-to-build-restful-service-with-java-using-jax-rs-and-jersey/

REST for Java – JAX-RS Annotations



Annotation	Description
@PATH(your_path)	Sets the path to base URL + /your_path. The base URL is based on your application name, the servlet and the URL pattern from the `web.xml` configuration file.
@POST	Indicates that the following method will answer to an HTTP POST request.
@GET	Indicates that the following method will answer to an HTTP GET request.
@PUT	Indicates that the following method will answer to an HTTP PUT request.
@DELETE	Indicates that the following method will answer to an HTTP DELETE request.
@Produces(MediaType.TEXT_PLAIN[, more-types])	@Produces defines which MIME type is delivered by a method annotated with @GET. In the example text ("text/plain") is produced. Other examples would be "application/xml" or "application/json".
@Consumes(type[, more-types])	@Consumes defines which MIME type is consumed by this method.
@PathParam	Used to inject values from the URL into a method parameter. This way you inject, for example, the ID of a resource into the method to get the correct object.

Standard for developing REST Applications
Spring Boot

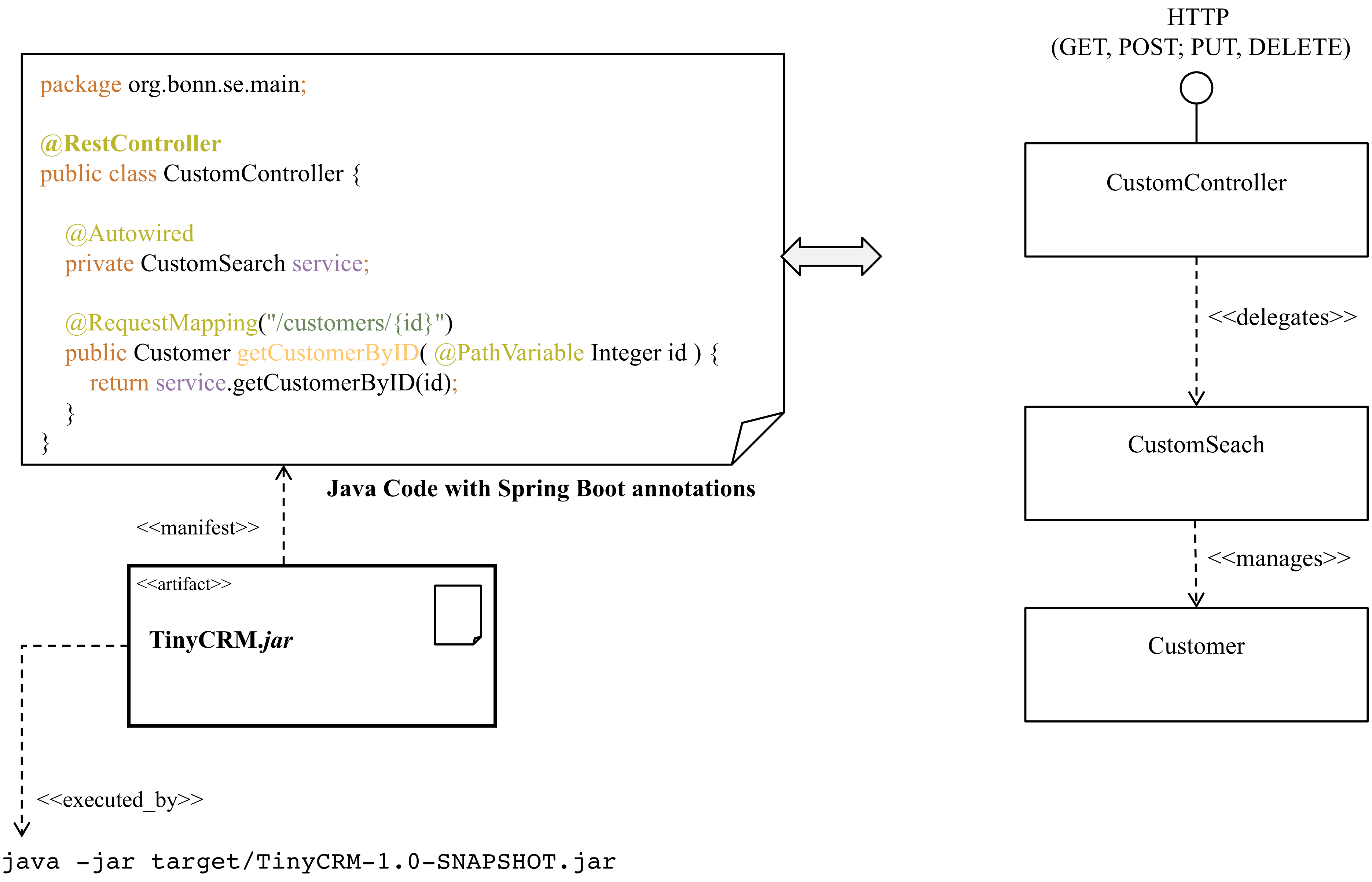
- The Spring Boot framework offers both a mature development model and a execution environment for REST-based applications

- Generation of Stand-alone REST-based applications (modules) by means of using Spring-based annotations
  - URL: http://projects.spring.io/spring-boot/

- The modularization principle allows for the implementation of Microservices

- Usage of Spring MVC for declaring REST-based applications with annotations (Pay attention: no JAX-RS! Can optionally be involved)

Implemementation of a REST Service (Sprint Boot)



```
package org.bonn.se.main;

@RestController
public class CustomController {

    @Autowired
    private CustomSearch service;

    @RequestMapping("/customers/{id}")
    public Customer getCustomerByID( @PathVariable Integer id ) {
        return service.getCustomerByID(id);
    }
}
```

Java Code with Spring Boot annotations

```
java -jar target/TinyCRM-1.0-SNAPSHOT.jar
```

Synchronous Interaction between REST applications

- From the Spring framework, class RestTemplate can be used for calling an external REST service from within a given (local) REST service,

- A direct mapping of the JSON-based representation and the internally POJO objects is possible by using the Framework Jackson
  - https://github.com/FasterXML/jackson

- Example with GET (shorted representation, including Mapping via Jackson):

```
RestTemplate template = new RestTemplate();
HttpEntity<String> head = new HttpEntity<>(headers);
String url = "http://sepp-crm.inf.h-brs.de/opencrx-rest-CRX/…;
ResponseEntity<String> res = template.exchange(url, HttpMethod.GET, head, String.class);

//Mapping JSON zu DTO via Jackson
ObjectMapper mapper = new ObjectMapper();
AccountDTO acc = mapper.readValue(res.getBody(), AccountDTO.class);
System.out.println("Account von: " + acc.getFirstName() + " " + acc.getLastName());
```

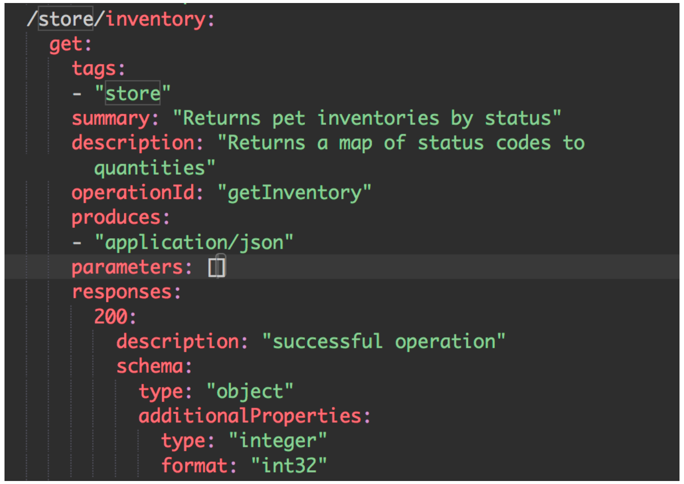
Good source on how to use RestTemplate: http://www.baeldung.com/rest-template

Documentation of REST-APIs with Swagger

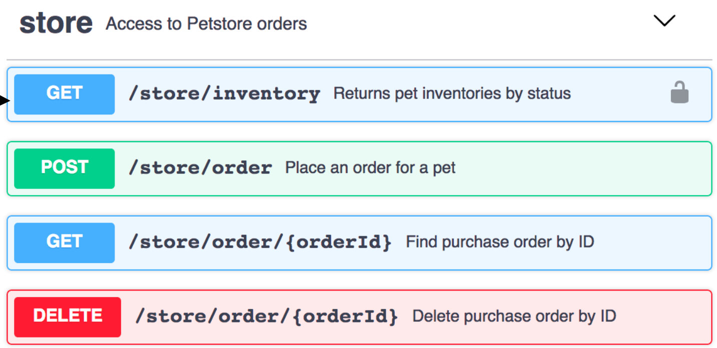
- Swagger is a collection of tools for both the documentation and the development of REST-APIs, pertaining server skeletons as well as client proxies.
  - Source: https://swagger.io/tools/open-source/getting-started/

- The interface can be defined with the YAML-based language OAS (OpenAPI Specification). The corresponding tool is called Swagger Editor.
  - Source: https://swagger.io/specification/v3/

- The interface can be rendered intuitively with the Swagger UI :



```
/store/inventory:
  get:
    tags:
    - "store"
    summary: "Returns pet inventories by status"
    description: "Returns a map of status codes to
      quantities"
    operationId: "getInventory"
    produces:
    - "application/json"
    parameters: []
    responses:
      200:
        description: "successful operation"
        schema:
          type: "object"
          additionalProperties:
            type: "integer"
            format: "int32"
```



store Access to Petstore orders

GET /store/inventory Returns pet inventories by status

POST /store/order Place an order for a pet

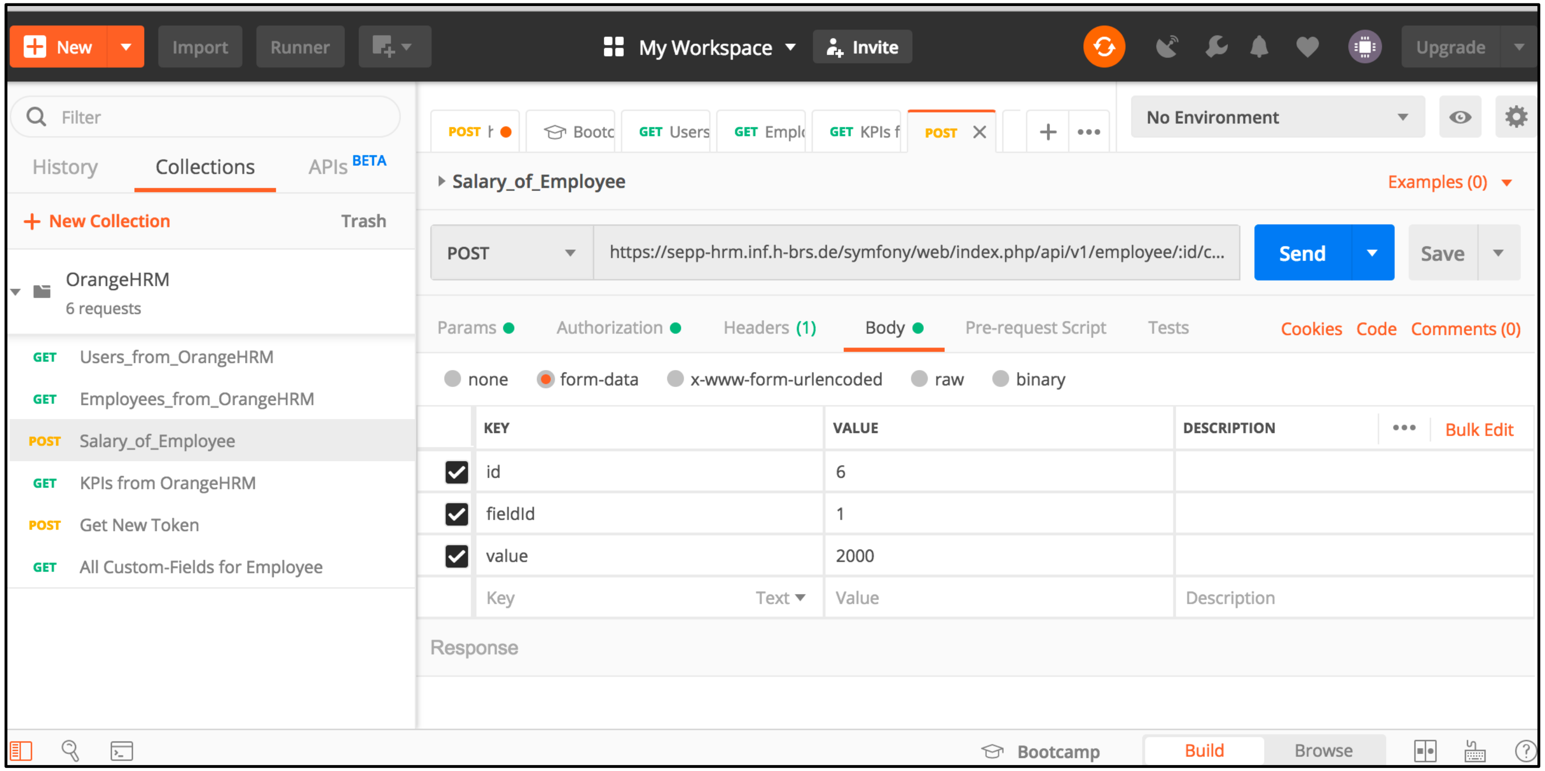
GET /store/order/{orderId} Find purchase order by ID

DELETE /store/order/{orderId} Delete purchase order by ID

The testing of REST-based APIs

- The testing especially of GET requests can be done with a conventional Web browser. However, most Browsers do not support methods like DELETE

- For a thorough and methodical test of the whole REST-based API, tools like Postman can be used.
  - Source: https://www.getpostman.com/



Salary_of_Employee

POST https://sepp-hrm.inf.h-brs.de/symfony/web/index.php/api/v1/employee/:id/c...

○ none ● form-data ○ x-www-form-urlencoded ○ raw ○ binary

	KEY	VALUE	DESCRIPTION
☑	id	6	
☑	fieldId	1	
☑	value	2000	
	Key	Value	Description

Video Tutorial for developing a REST-based Microservice

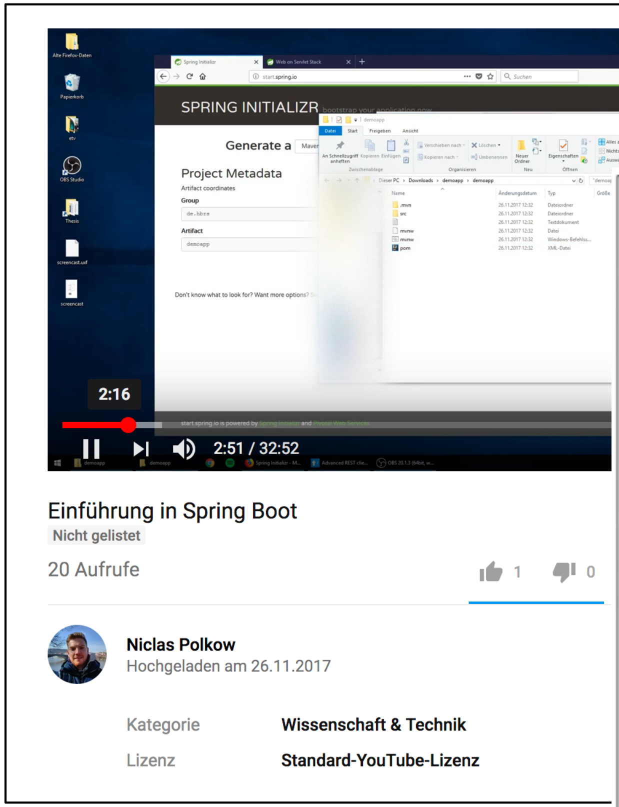
- A step-by-step tutorial (screencast) for developing a REST-based service:

  https://www.youtube.com/watch?v=t64LxbkHVjw

- You will learn basic concepts with this tutorial (German language):
  - Spring Boot
  - Spring Initializer
  - IntelliJ IDEA
  - Maven
  - REST
  - Test tool Postman

- c/o Niclas Polkow 2017, HBRS

- Note: the visual appearances of the current version of tools might differ slightly



Einführung in Spring Boot

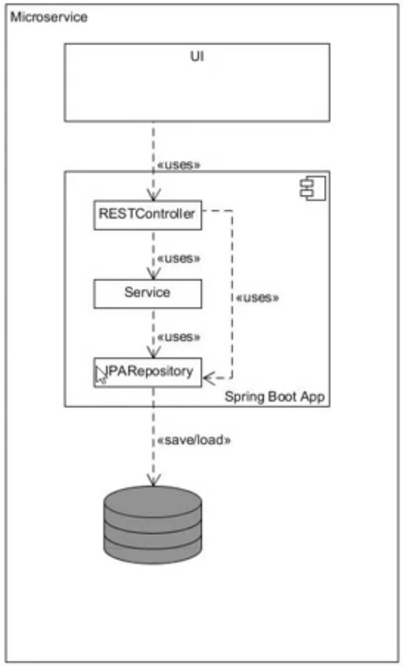
Nicht gelistet

20 Aufrufe

Niclas Polkow
Hochgeladen am 26.11.2017

Kategorie Wissenschaft & Technik

Lizenz Standard-YouTube-Lizenz



Structure of this Module

Chapter 2: Introduction to Software Architectures and Architectural Integration		
1	Motivation – Software Development Lifecycle	✓
2	Definition and Properties of a Software Architecture	✓
3	Architectural Integration: Basic concepts and assumptions	✓
4	Modelling of Software Architectures with UML (4 Views Model)	✓
5	Revisiting a fundamental Architecture Pattern: Layer	✓
6	API Design: Basic concepts	✓
7	The REST architectural style: Basic concepts	✓
8	**Modularization in JavaScript: Introduction to Node.js**	