#### GITHUB Repo:

#### <https://github.com/ai1045proscensus/RESTWithSpringBoot>

Learn to design Rest APIs and develop RESTFul Webservices using Spring Boot

A RESTful web service is the implementation of a REST API

#### 

#### 

#### What you’ll learn

* Learn to design Rest APIs and develop RESTFul Webservices using Spring Boot
* Swagger
* HATEOAS
* We will be using:
  + Spring (Dependency Management),
  + Spring MVC (or Spring REST),
  + Spring Boot,
  + Spring Security (Authentication and Authorization),
  + Spring Boot Actuator (Monitoring),
  + Swagger (Documentation),
  + Maven (dependencies management),
  + Eclipse (IDE),
  + Postman (REST Services Client) and
  + Tomcat Embedded Web Server.

#### Who this course is for:

* Java Developers who want to learn how to design Rest APIs and develop RESTFul Webservices using Spring Boot

**contents**

***RESTful Web Services with spring Boot***

1. **What** is a ws?
2. **How** do ws work?
3. Soap ws
4. Rest ws
5. Comparison soap - rest

**What** is a ws?

* **interapp** communication (**app2app / machine2machine** interaction over a **network => http**)
* **interoperable** (**platform independent** /not language dependent (java, .net, …) alle sollen mit meinem ws kommunizieren können)

**How** do ws work?

* ***communication*** via **request-response** model

(web service ***like webserver*** (“service **provider: hosts** the ws”)**:** takes request, gives response back. **Request** is the **input** of a ws and **response** is the **output** of a ws)

* platform independency achieved by **independent** format of request/response, i.e. **data (or message) exchange format**: xml **/ json**

(to make the ws platform-independent we make the request and response platform independent)

(JSON : javascript object notation.

JSON is how js represent its objects.)

* **service definition:** (thats how apps (clients/”service **consumers”**) know how to “talk” to the ws. every ws has one)
  + **format** (xml / json) for request and response
  + **structure** (**how to define a request** / consume a response)
  + **endpoint: how to call the service (url)**
  + **“contract”** between ws and clients **/ “API”**
  + **WSDL** (web service definition language)
    - defines the endpoint (url)
    - all operations that are exposed (like all methods of a class (API): get all user details, delete user, add new user… )
    - request structure
      * **how do i send request for** deleting user / adding user, …?
    - response structure
      * what kind of response to expect?
* **transport: how** a ws is **called**
  + called over internet? => url =>
    - **http => REST, SOAP**
  + ws exposed over a **QUEUE** =>
    - **MQ => SOAP**
      * client places a message in the queue (request)
      * ws listens to / polls the queue
      * when request comes in => ws takes the request, handles it
      * ws puts the response back to the queue
      * client gets response from queue

soap ws

(simple object access protocol)

SOAP defines a specific XML request and response structure:

* Soap **envelope**, contains (wie html und head body tags)
  + soap **header (optional)**
  + soap **body**
* **header**: meta information
  + authentication
  + authorization
  + signatures …
* **body**: real content (“meat”) of request/response

fazit:

SOAP is all about adhering to the soap XML structure (envelop header body)

rest

**RE**presentational **S**tate **T**ransfer

rest was invented by same person who invented **http** protocol.

no need to reinvent the wheel => use http to develop webservices as well

=> **same principles** like in webapps:

* client sends **httprequest** to server
* server responds back with **httpresponse** in html format
* client displays response
* http**request** **methods**: GET , POST (neu erzeugen), PUT (update)
* http**response status codes**: 200, 404, …

**Key concept: RESOURCE**

resource: anything that i want to **expose** to the outside world through my app.

=> anything that my ws can **return**

zb: task management app:

* users
* tasks
* list of tasks

are all resources.

=> **like ENTITIES or OBJECTS** (in ER models or class diagrams)

Zusammenhang http: URI

**every resource has a URI (Uniform Resource Identifier)**

* user/ai1045
  + resource (“blatt”) is ai1045
  + with the uri user/ai1045
* user/ai1045/tasks
  + resource is tasks (ALL tasks of ai1045)
  + its uri is user/ai1045/tasks
* user/ai1045/tasks/313
  + resource is task number 313
  + its uri is user/ai1045/tasks/313

**=> assign a uri to each resource**

format of resource:

* html
* xml
* json

=> ist REST egal. most **important**:

1. define your **resource**
2. perform **actions** on resource using facilities provided by HTTP.

zb:

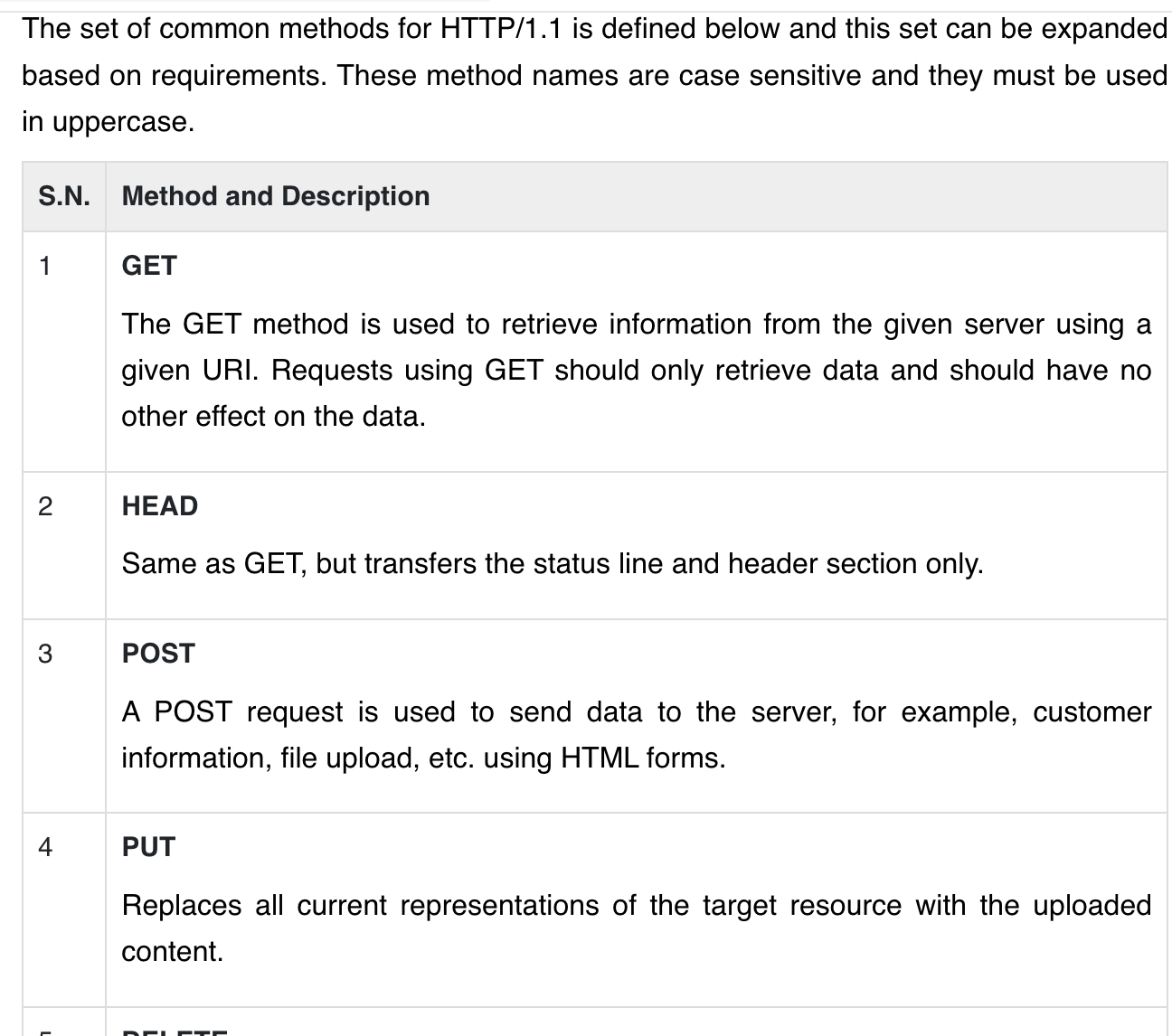
* user updaten: PUT /users/ai1045
* delete all users: DELETE /users
* Get all users: GET /users
* Delete one user: DELETE /users/ai1045

=> you have to **think in terms of the resources:**

* **what** are the **resources** (that I’d like to **expose** to other apps)
* make use of **HTTP** for **actions / operations** on resources
  + GET
  + POST
  + PUT
  + FETCH
  + DELETE
  + …

How many HTTP request methods are there?

API developers typically only use **GET, PUT, or POST,** but the official HTTP Request Method registry lists **39 total HTTP verbs**, each providing a method for powerful interactions.09.01.2020



from <https://www.tutorialspoint.com/http/http_methods.htm>

Fazit:

* data exchange format egal (**json**, xml, … )
* transport: MUST be HTML (REST is built completely on top of HTTP)
* service definition:
  + not standarized (drawback..man weiss nicht wie man senden muss den request).
  + (WADL: web application definition language (eher unbekannt))
  + **Swagger**: specify / define your RESTful Web services.

rest vs soap

REST: defines architectural approach.

SOAP: poses restrictions on the format of XML for data exchange (between client and ws)

SOAP:

* data exchange structure must be predefined XML structure (envelop header body)
* transport egal (mq, http)

REST:

* data structure egal(json, xml,...)
* transport must be http
* service definition:
  + not standarized(drawback)