Woldia University
School of Computing

Department of Software Engineering
Course Title: **Web Service,**
Course Code: **SEng5127**

Chapter Four: Building RESTful Web Services

By : Demeke G.
AY:2018 E.C

Outline

- ☛Introduction to RESTful webservice
- ☛REST Architectural Style
- ☛Developing RESTful Web Services with JAX–RS

Learning Outcome

- ☛Describe REST principles and differentiate RESTful services from SOAP.
- ☛Develop and annotate RESTful Web Services using JAX-RS.
- ☛Use HTTP methods and headers for content negotiation and resource operations.
- ☛Deploy REST services in a Java application environment.
- ☛Design and map RESTful URLs to classes and methods using JAX-RS.
- ☛Bind URI components (path/query parameters, headers) to method arguments.

Introduction RESTful Web Service

- ☛A RESTful Web API is an architectural style and approach to designing networked applications
- ☛**REST** stands for **representational state transfer** and was created by computer scientist Roy Fielding.
- ☛ leverages the principles of REST to create web services that interact with clients over the HTTP protocol.
- ☛RESTful APIs are used to facilitate communication between different software systems in a stateless, scalable, and flexible manner.
- ☛REST API is a way of accessing web services in a simple and flexible way without having any processing.
- ☛REST technology is generally preferred to the more robust SOAP technology because REST uses less bandwidth, simple and flexible making it more suitable for internet usage.
- ☛It's used to fetch or give some information from a web service.
- ☛All communication done via REST API uses only HTTP/Https request
- ☛The client and server operate independently

How REST works

- A request is sent from client to server in the form of a web URL as HTTP GET or POST or PUT or DELETE request.
- After that, a response comes back from the server in the form of a resource which can be anything like **HTML, XML, Image, or JSON**.
- But now **JSON** is the most popular format being used in Web Services.
- In HTTP there are five methods that are commonly used in a REST-based Architecture
- i.e., POST, GET, PUT, PATCH, and DELETE. These correspond to create, read, update, and delete (or CRUD) operations respectively.

JSON Data Format

- JSON (JavaScript Object Notation) is a lightweight, text-based data format used for data interchange.
- It is easy for humans to read and write and easy for machines to parse and generate.
- JSON is widely used in web applications for data exchange between clients and servers.
- Key-value pairs for data representation.
- It is supported by almost all programming languages.
- Represents objects, arrays, numbers, strings, booleans, and null.
- Objects: Enclosed in curly braces {}, contain key-value pairs.
- Arrays: Enclosed in square brackets [], hold a list of values.
- Key-Value Pairs: Keys are strings, and values can be any valid JSON type.
- Keys must always be strings enclosed in double quotes (").

Cont..

- Strings must be in double quotes ("text"), not single quotes ('text’).
- Numbers can be integers or decimals but cannot have leading zeros.
- JSON does not support comments.
- Represents an empty or missing value.
- Example :

```
{
 "id": 101,
 "name": "Alice Johnson",
 "address": {
  "street": "123 Elm St",
  "city": "Springfield",
  "postalCode": "62704"
 },
 "isActive": true
}
```

Activities

1) **Create a JSON object for an online order with:**
   - orderId
   - customerName
   - items (array of objects: productName, quantity, price)
   - totalAmount
   - orderStatus

2) **Write a JSON array that contains three car objects with:**
   - Brand ,model ,year, color

Rest Architectural Style

☛ Web services that follow the REST architectural style are known as RESTful web services.

☛ It allows requesting systems to access and manipulate web resources by using a uniform and predefined set of rules.

☛ There are six architectural constraints

1) Uniform Interface
2) Stateless
3) Cacheable
4) Client-Server
5) Layered System
6) Code on Demand

Cont..

1)Client-Server:

- Separation of concerns.
- The client is responsible for the user interface and user state.
- The server is responsible for data storage, scalability, and security.
- This allows them to evolve independently.

2)Stateless

- Each client request must contain all the information necessary for the server to understand and process it.
- The server cannot store any session context about the client between requests.
- State is kept entirely on the client (e.g., in session storage or within the request itself as tokens).
- Benefits: Scalability (any server can handle any request), reliability, visibility.

Cont..

3)Cacheable

- Responses from the server must be explicitly labeled as cacheable or non-cacheable.
- This allows clients to cache responses, dramatically improving performance and reducing server load.
- implemented using HTTP caching headers (Cache-Control, ETag, Expires).

4) Layered System

- The architecture can be composed of multiple hierarchical layers (proxies, load balancers, security layers, caches).
- A client cannot tell if it's connected directly to the end server or an intermediary.
- This improves scalability and security.

5) Code on Demand

- The server can temporarily extend client functionality by transferring executable code (e.g., JavaScript in browser, Java applets)
- This is the only optional constraint.

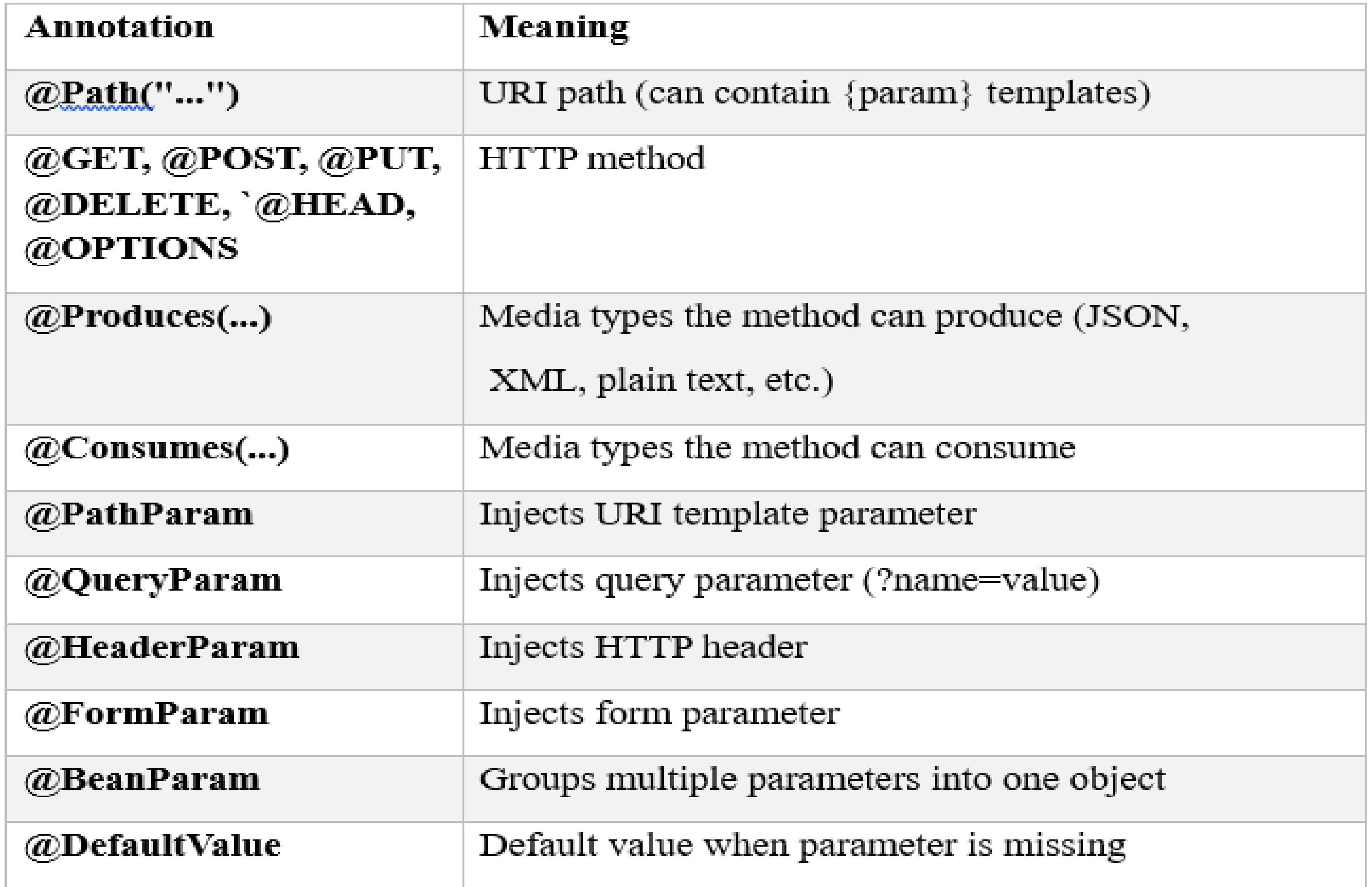
Cont..

6)Uniform Interface

- The most central and defining constraint of REST. It has four sub-principles:
  - **Identification of Resources:** Every piece of data (resource) is uniquely identified by a URI (e.g., https://api.example.com/users/456).
  - **Manipulation of Resources Through Representations:** Clients interact with resources via representations (e.g., a JSON/XML). They never interact with the resource directly.
  - **Self-descriptive Messages:** Each request/response message contains enough information (via HTTP method, status codes, headers, media types) to describe how to process it.
  - **Hypermedia as the Engine of Application State (HATEOAS):** The response from the server should contain hyperlinks to related actions and resources. The client discovers what it can do next dynamically from these links, rather than relying on hardcoded URLs.

Developing RESTful Web Services with JAX–RS

☛Turn a plain Java object into a REST resource by adding a few annotations:



Annotation	Meaning
@Path("...")	URI path (can contain {param} templates)
@GET, @POST, @PUT, @DELETE, `@HEAD, @OPTIONS	HTTP method
@Produces(...)	Media types the method can produce (JSON, XML, plain text, etc.)
@Consumes(...)	Media types the method can consume
@PathParam	Injects URI template parameter
@QueryParam	Injects query parameter (?name=value)
@HeaderParam	Injects HTTP header
@FormParam	Injects form parameter
@BeanParam	Groups multiple parameters into one object
@DefaultValue	Default value when parameter is missing

Example:

```
@Path("/books")                    // root resource path
@Produces(MediaType.APPLICATION_JSON) // default response type for all methods
@Consumes(MediaType.APPLICATION_JSON) // default request type for all methods
public class BookResource {
    private final BookService bookService = new BookService();
    @GET        // HTTP GET
    public List<Book> getAllBooks() {   return bookService.findAll();    }
    @GET
    @Path("/{id}")       // path parameter
    public Book getBook(@PathParam("id") Long id) {
        return bookService.findById(id)
                .orElseThrow(() -> new NotFoundException());
    }
```

Example cont..

```
@POST     // HTTP POST
  public Response createBook(Book book) {
     Book created = bookService.save(book);
     return Response
           .created(URI.create("/books/" + created.getId())) .entity(created).build();
  }
  @PUT
  @Path("/{id}")
public Book updateBook(@PathParam("id") Long id, Book book)
{return bookService.update(id, book);   }
  @DELETE
  @Path("/{id}")
  public Response deleteBook(@PathParam("id") Long id) {
     bookService.delete(id);
     return Response.noContent().build();
  }
}
```

Configuring Result Types via HTTP Headers

☛Configuring the result type of an HTTP response is primarily managed through two HTTP headers:

- The Accept request header (**sent by the client**)
- The Content-Type response header (**sent by the server**).

Client Request Header: Accept

☛The client uses the Accept header to tell the server which media types it can process or prefers.

☛The server then uses this information for content negotiation to select an appropriate representation of the resource.

☛**Syntax:** Accept: media-type/subtype or Accept: */* (accepts all types).

☛Example: To request a JSON response, a client would send:

- Accept: application/json

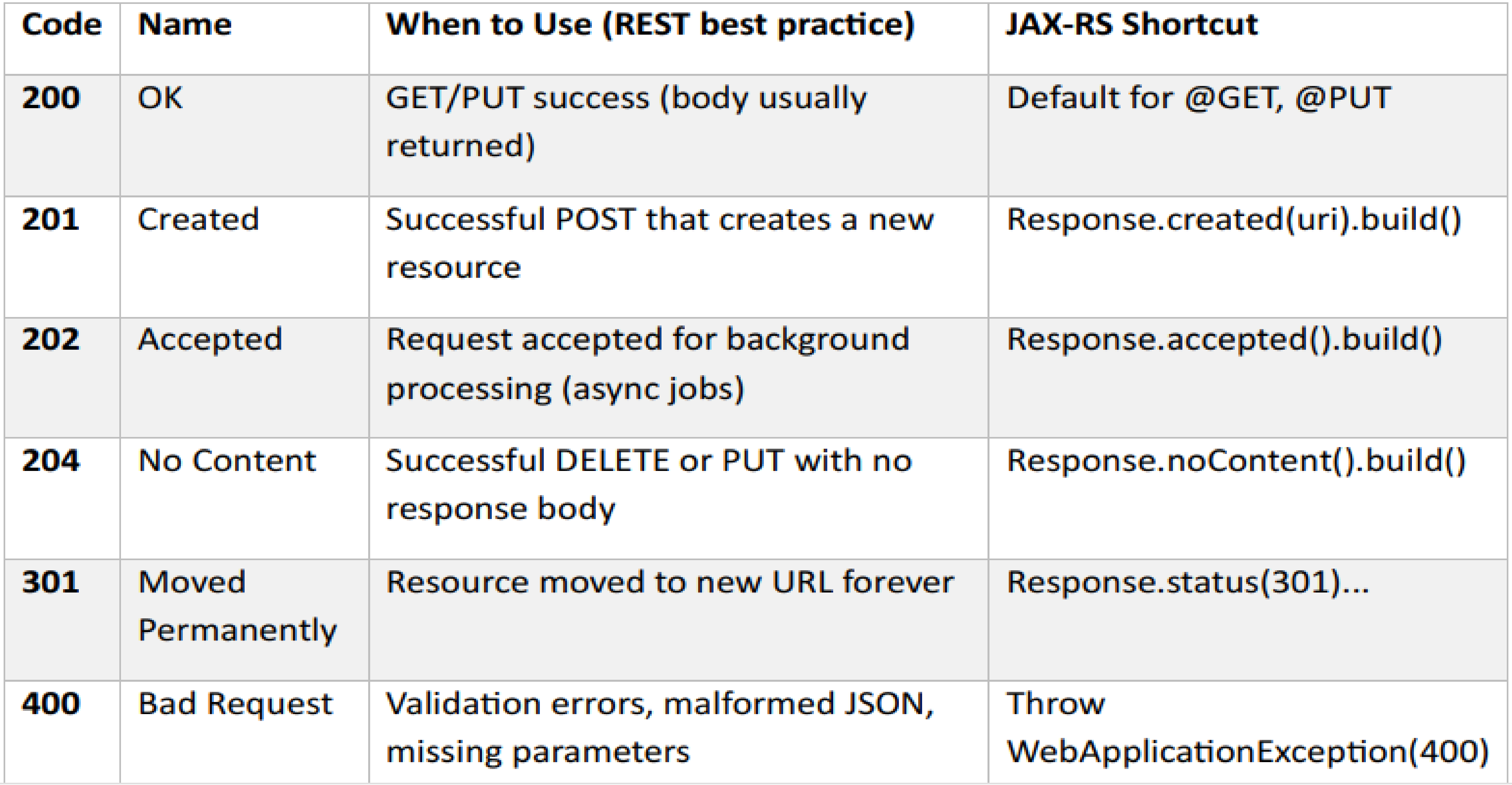
☛To request HTML or plain text with a preference for HTML, a client could send:

- Accept: text/html, text/plain;

Server Response Header: Content-Type

- The server uses the Content-Type header in its response to inform the client about the actual media type of the returned data's body.
- The server uses the Content-Type header in its response to inform the client about the actual media type of the returned data's body.
  - Content-Type: application/json
  - Content-Type: text/html; charset=UTF-8

Status Code (1)



Code	Name	When to Use (REST best practice)	JAX-RS Shortcut
200	OK	GET/PUT success (body usually returned)	Default for @GET, @PUT
201	Created	Successful POST that creates a new resource	Response.created(uri).build()
202	Accepted	Request accepted for background processing (async jobs)	Response.accepted().build()
204	No Content	Successful DELETE or PUT with no response body	Response.noContent().build()
301	Moved Permanently	Resource moved to new URL forever	Response.status(301)...
400	Bad Request	Validation errors, malformed JSON, missing parameters	Throw WebApplicationException(400)

HTTP Status Code (2)

401	Unauthorized	Missing or bad authentication (no credentials)	Automatic with security constraints
403	Forbidden	Authenticated but not authorized (insufficient rights)	Throw WebApplicationException(403)
404	Not Found	Resource does not exist	Throw NotFoundException()
405	Method Not Allowed	Wrong HTTP verb on existing resource (e.g., POST on read-only)	Automatic if method not defined
406	Not Acceptable	Client's Accept header cannot be satisfied	Automatic when no matching @Produces
415	Unsupported Media Type	Client sent body type not listed in @Consumes	Automatic when no matching @Consumes
500	Internal Server Error	Unexpected exception (never expose stack trace!)	Default when exception is uncaught

Example

```
@DELETE
@Path("/{id}")
public Response delete(@PathParam("id") Long id)
{    if (service.delete(id)) {
     return Response.noContent().build();     // 204     }
 else {
return Response.status(404).entity("Customer not found").build(); // 404
  }
}
```

Summary

- ☛REST is simple, scalable, and cache-friendly
- ☛JAX-RS turns POJOs into full-featured HTTP APIs with just annotations
- ☛ Content negotiation via @Produces / Accept header
- ☛No deployment descriptor needed in modern servers
- ☛Use proper HTTP status codes and HATEOAS for mature APIs
- ☛Prefer JSON + OpenAPI documentation