



SPRING

Building a basic REST API

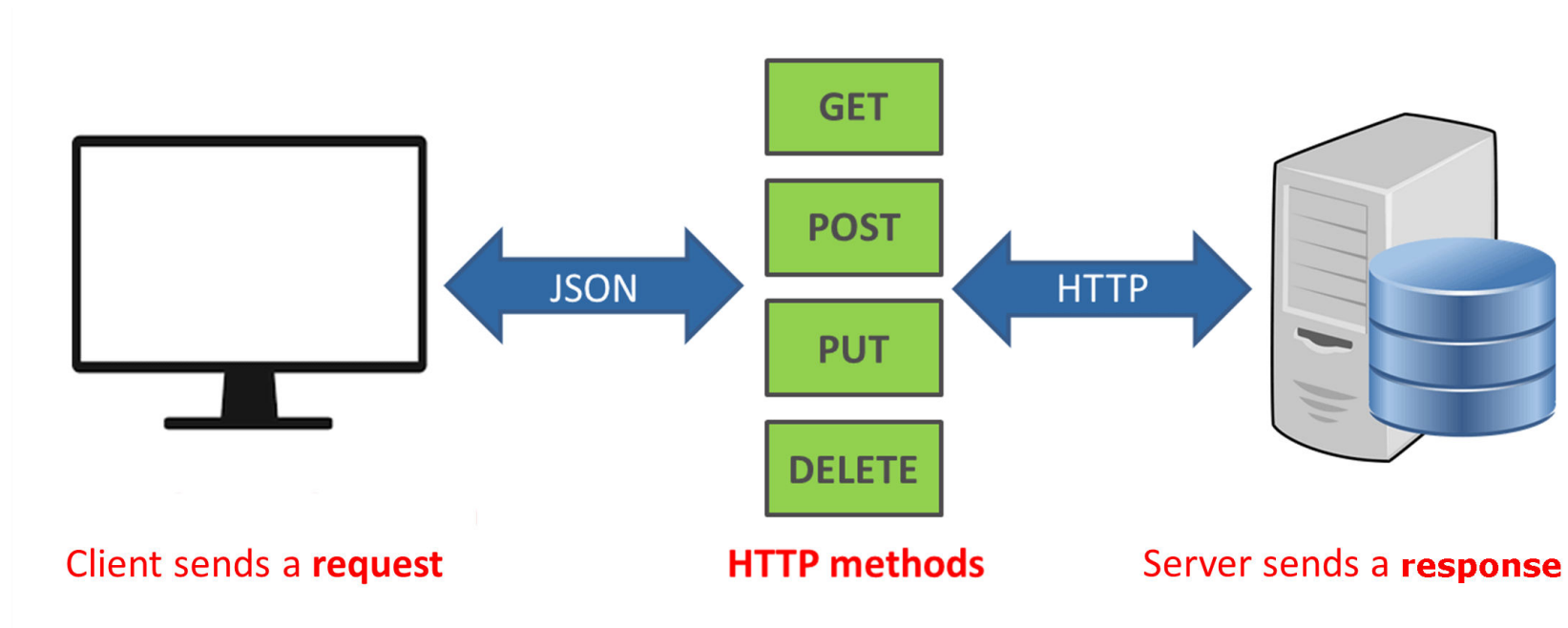
Developing RESTful APIs is a task that most developers have faced at least once in their lives. Nowadays, with the increasing popularity of front-end frameworks like Angular, React, and Vue.js and with the mass adoption of smartphones, RESTful APIs became the most popular approach backend developers leverage to provide an interface for their applications.

- **What is REST?**
- Java Example
 - Working with the Postman tool
 - Use of class "ResponseEntity<E>" and "Optional<E>"
 - GET, POST, PUT, DELETE
- SwaggerUI automated documentation
- Testscript in Postman tool

What is a REST API?



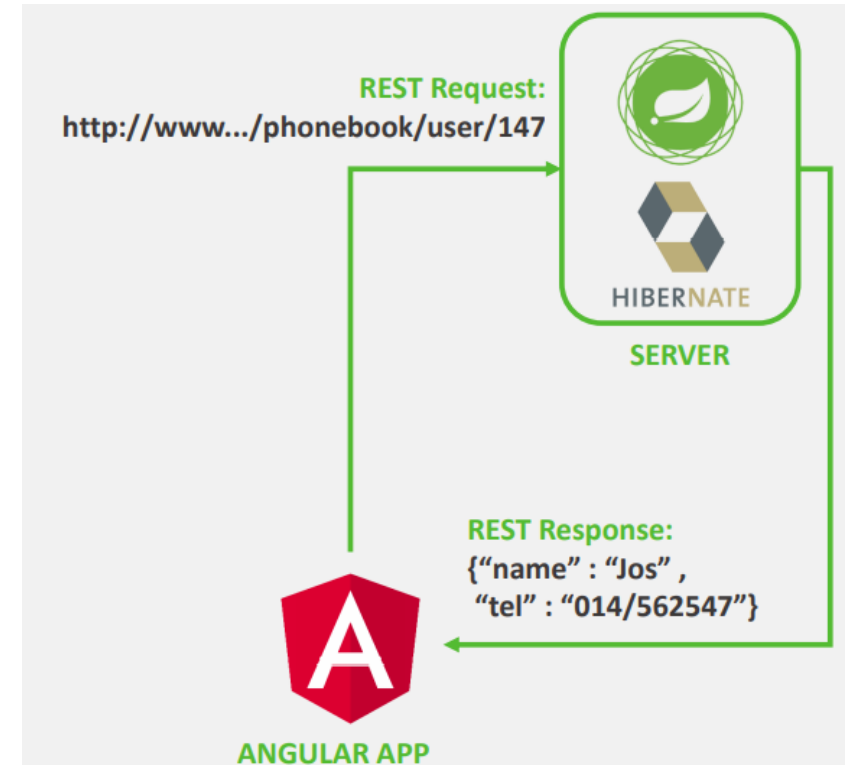
- **REST = Representational State Transfer**
- Something that allows one piece of software to talk to another piece of software over the HTTP protocol



What is a REST API?



- Platform-independent (iOS, Unix, Windows, ...)
- Language-independent (C#, Java, PHP, ...)
 - Very common: REST-service built in Java Spring communicates with apps built in Angular, React, Vue, ...
- Uses HTTP for calls
 - HTTP Methods (GET, POST, PUT, ...)
 - Messages in JSON format:
 - Easy to use by JS, Typescript, Java, C# apps
 - Can in essence also be XML, CSV, ...



REST requires that a client makes a request to the server in order to retrieve or modify data on the server. A request generally consists of:

- an **HTTP verb**, which defines what kind of operation to perform
 - **GET** (retrieve a resource or collection of resources)
 - **POST** (create a new resource)
 - **PUT** (update a resource)
 - **DELETE** (remove a resource)
 - see also: <https://www.codecademy.com/articles/what-is-crud>
- a **header**, which allows the client to pass along information about the request
 - In the header of the request, the client sends the type of content that it is able to receive from the server (= accept field). The options for types of content are MIME Types.
- a **path** to a resource
 - In RESTful APIs, paths should be designed to help the client know what is going on. Conventionally, the first part of the path should be the plural form of the resource. This keeps nested paths simple to read and easy to understand.
 - A path like *fashionboutique.com/customers/223/orders/12* is clear in what it points to, even if you've never seen this specific path before. We can see that we are accessing the order with id 12 for the customer with id 223.

Rest: SEPARATION OF CLIENT AND SERVER



- In the REST architectural style, the implementation of the client and the implementation of the server can be done independently without each knowing about the other. This means that the code on the client side can be changed at any time without affecting the operation of the server, and the code on the server side can be changed without affecting the operation of the client.
- As long as each side knows what format of messages to send to the other, they can be kept modular and separate. Separating the user interface concerns from the data storage concerns, we improve the flexibility of the interface across platforms and improve scalability by simplifying the server components. Additionally, the separation allows each component the ability to evolve independently.
- By using a REST interface, different clients hit the same REST endpoints, perform the same actions, and receive the same responses.

- Systems that follow the REST paradigm are stateless, meaning that the server does not need to know anything about what state the client is in and vice versa. In this way, both the server and the client can understand any message received, even without seeing previous messages. This constraint of statelessness is enforced through the use of *resources*, rather than *commands*. Resources are the nouns of the Web - they describe any object, document, or *thing* that you may need to store or send to other services.
- These constraints help RESTful applications achieve reliability, quick performance, and scalability, as components that can be managed, updated, and reused without affecting the system as a whole, even during operation of the system.

- What is REST?
- **Java Example**
 - Working with the Postman tool
 - Use of class "ResponseEntity<E>" and "Optional<E>"
 - GET, POST, PUT, DELETE
- SwaggerUI automated documentation
- Testscript in Postman tool

Java Example



- [Video](#) : example of GET-requests



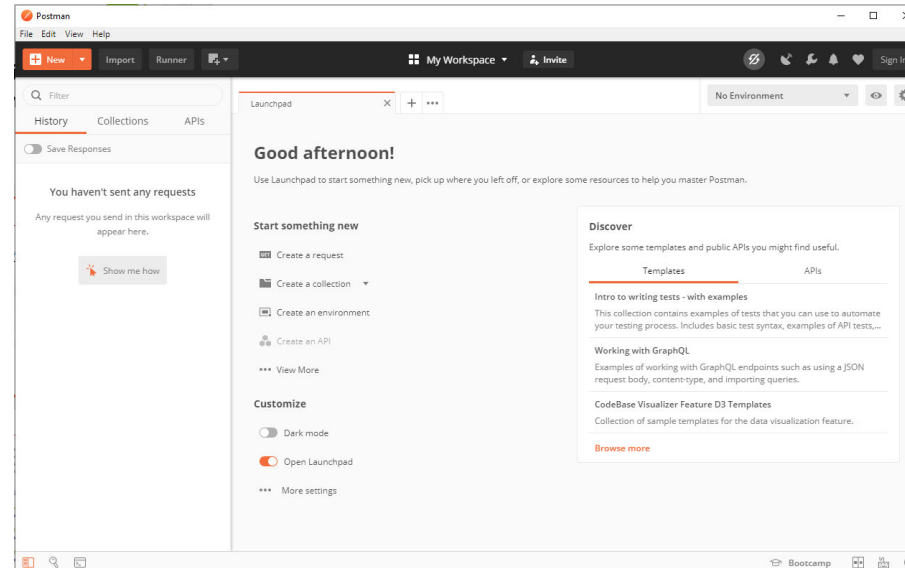
- [Video](#) : example of POST-request



Creating our own API



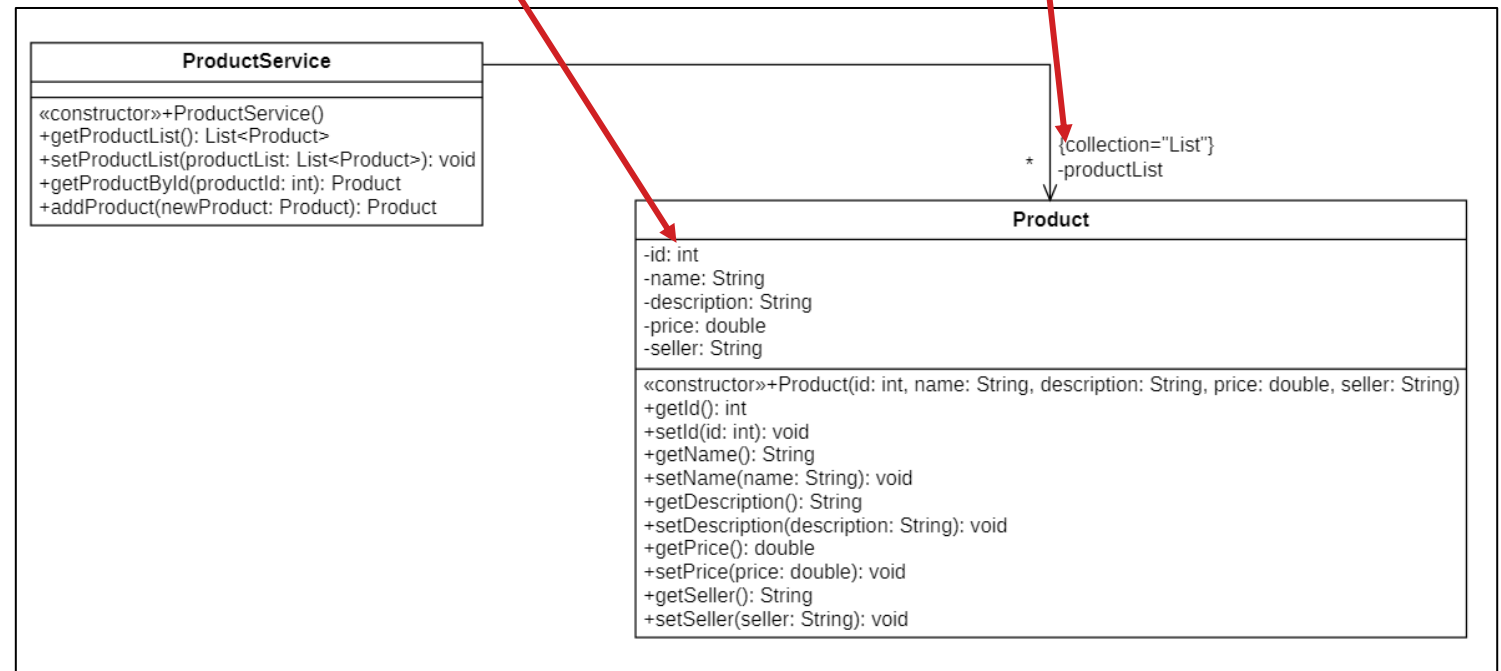
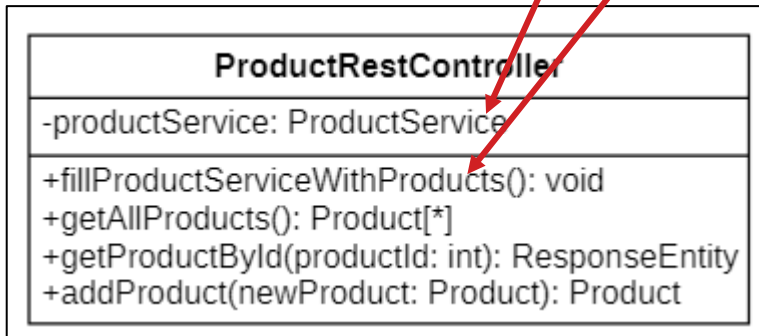
- In the following slides we will build a REST API as shown in the video's with some differences and extensions...
- Preparation:
 - Install Postman Tool that allows you to test REST APIs
 - Download: <https://www.getpostman.com/apps>
 - After install you can continue without signing in



Data storage in the controller



- We will not teach databases until next week; therefore, we simulate this by:
 - adding an "id" to the POJO-class "Product"
 - introducing a ProductService-class which can access every Product-object
 - adding an attribute of the ProductService-class to our RestController which will store the data
 - using @PostConstruct to a method in the RestController to fill the data our attribute with objects



Create the Class Product (in the package model)



```
public class Product {  
    private int id;  
    private String name, description;  
    private double price;  
    private String seller;
```

```
    public Product(int id, String name, String description, double price, String seller) {  
        this.id = id;  
        this.name = name;  
        this.description = description;  
        this.price = price;  
        this.seller = seller;  
    }
```

//getters and setters of all attributes

Create the Class ProductService (in the package model)



```
public class ProductService {  
    private List<Product> productList= new ArrayList<>();  
  
    public ProductService() {  
        productList.add(new Product(1, "IPHONE", "This is an Awesome Iphone", 400.4, "Apple Inc"));  
        productList.add(new Product(2, "Samsung", "This is an Awesome Samsung", 300.4, "Samsung"));  
        productList.add(new Product(3, "LG Z", "This is an Awesome LG", 200.4, "LG Inc"));  
    }  
  
    public List<Product> getProductList() {  
        return productList;  
    }  
  
    public void setProductList(List<Product> productList) {  
        this.productList = productList;  
    }  
}
```

RestController ProductRestController



```
@RestController  
@RequestMapping("/api")  
public class ProductRestController {
```

@RestController is the annotation for creating Restful controllers. It is autodetected through classpath scanning. It converts the response to JSON or XML. It is typically used in combination with annotated handler methods based on the @RequestMapping annotation (@GetMapping, @PostMapping).

@RequestMapping("/api") is the annotation for adding "/api" to the path when sending requests to the RestController

RestController ProductRestController



```
@RestController
@RequestMapping("/api")
public class ProductRestController {
    private ProductService productService;

    @PostConstruct
    public void fillProductServiceWithProducts() {
        productService = new ProductService();
    }
}
```

- We make use of an attribute "productService" in the RestController to access data. Every method in the RestController can use this attribute to access the product-objects.
- Calling the constructor of ProductService() will ensure that the attribute is instantiated and that the productList of the productService is filled with objects. We want this to happen from the very start of the application. We use the "@PostConstruct" annotation for that.

- In the previous lesson every request from an HTML-page was handled by a method with an **@RequestMapping** annotation.
- When building a REST controller we will have 4 different kinds of Request:
 - **GET** = used to retrieve data from the server. Multiple get requests to the same URL should be valid and no data should be changed on the server side.
 - **POST** = used when you need to create data on the server side. The enclosed entity should be considered as a new one that has to be added.
 - **PUT** = used when you want to change data on the server side. The enclosed entity should be considered as a modified version of the one residing on the origin server.
 - **DELETE** = used when you want to delete data on the server.



- The @RequestMapping annotation handles all types of incoming HTTP request including GET, POST, PUT etc. By default, it's assumed all incoming requests to URLs are of the HTTP GET kind, and that was exactly what we needed in the previous lesson. Now we want to differentiate the mapping by HTTP request type. To do that we need to explicitly specify the HTTP request method.
- We have some new annotations to do this:
 - @GetMapping (or alternative: @RequestMapping(method = RequestMethod.GET))
 - @PostMapping (or alternative: @RequestMapping(method = RequestMethod.POST))
 - @PutMapping (or alternative: @RequestMapping(method = RequestMethod.PUT))
 - @DeleteMapping (or alternative: @RequestMapping(method = RequestMethod.DELETE))
- When building a REST service it's very import to use these new annotations or to add a method parameter to the @RequestMapping annotation. If we don't do that and stick to the old @RequestMapping without mentioning the method, we are only offering (the default) GET-functionalities in our REST service

Exposing APIs: GET /products



From here on we start adding code to expose the REST API

```
@GetMapping("/products")
public List<Product> getAllProducts() {
    return
    productService.getProductList();
}
```

- **@GetMapping("/products")**


Maps GET requests for this URL, to the method underneath the annotation.

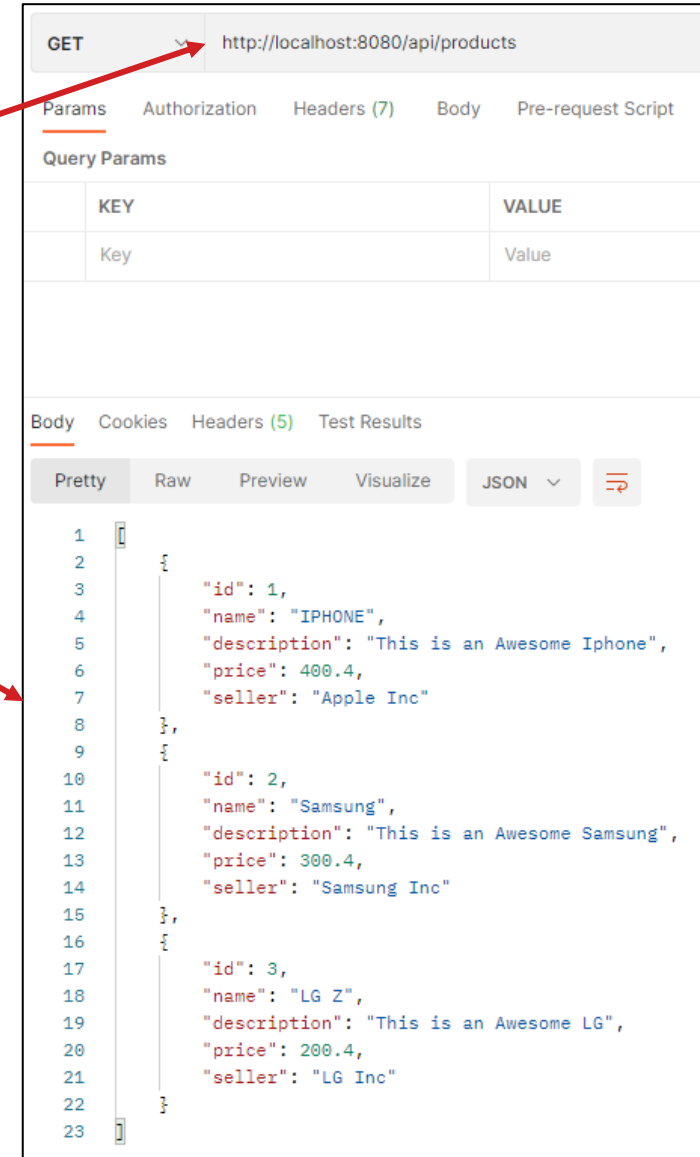
the name of the path should be the plural form of the resource
(resource = product => path = /products)

Exposing APIs: test GET /products



Now it's time for testing our first REST API:

- Run your application in IntelliJ 
- Start Postman and send a GET request to <http://localhost:8080/api/products>
- You get the result back in JSON format



- JSON format?
 - **J**ava**S**cript **O**bject **N**otation
 - a lightweight format for storing and transporting data
 - is often used when data is sent from a server to a web page
 - Standard for NoSQL databases like MongoDB
 - "self-describing" and easy to understand
- JSON Syntax rules
 - Data is in name/value pairs e.g. "name": "IPHONE"
 - Curly braces {} hold objects (= with multiple name/value pairs)
 - Square brackets [] hold arrays
 - Data is separated by commas (between objects and between name/value pairs)

```
{
  "id": 1,
  "name": "IPHONE",
  "description": "This is an Awesome Iphone",
  "price": 400.4,
  "seller": "Apple Inc"
},
{
  "id": 2,
  "name": "Samsung",
  "description": "This is an Awesome Samsung",
  "price": 300.4,
  "seller": "Samsung Inc"
},
{
  "id": 3,
  "name": "LG Z",
  "description": "This is an Awesome LG",
  "price": 200.4,
  "seller": "LG Inc"
}
```

Exposing APIs: GET /product/{id}



```
@GetMapping("/product/{id}")
```

```
public ResponseEntity<Product> getProductById(@PathVariable("id") int productId) {  
    Product product = productService.getOptionalProductById(productId);  
}
```

- A GET request to **/product/{id}** returns the product-object with the given id in the ArrayList
- **@PathVariable** or **@PathVariable(value = "id")**
 - Lets Spring know that the following is a variable from the URL
- **ResponseEntity<Product>**
 - Allows you to modify the response with optional headers and status code.

Exposing APIs: GET /product/{id}



- In ProductRestController:

```
@GetMapping("/product/{id}")  
public ResponseEntity<Product> getProductById(@PathVariable("id") int productId) {  
    Optional<Product> product = productService.getOptionalProductById(productId);  
}
```

- In ProductService:

```
public Optional<Product> getOptionalProductById(int productId){  
    return getProductList().stream().filter(p-> p.getId()==productId).findFirst();  
}
```

- "findFirst()" returns an Optional<T> i.e. Optional<Product>
- Optional is a container object used to contain objects that can be null or not-null. See next slides...

Class Optional <T>



- Every Java Programmer is familiar with [NullPointerException](#). It can crash your code. And it is very hard to avoid it without using too many null checks.
- To overcome this, Java has introduced a class Optional in java.util package.
 - a neat code without using too many null checks.
 - specify alternate values to return or alternate code to run.
 - => makes code more readable

Class Optional <T>



- This class may or may not contain a non-null value and has various methods to facilitate code to handle values as 'available' or 'not available' instead of checking null values:
 - if a value is present, [`isPresent\(\)`](#) will return true
 - [`get\(\)`](#) will return the value
 - [`orElse\(\)`](#) will return the value if present and a default value if value is not present

```
@GetMapping("/product/{id}")
public ResponseEntity<Product> getProductById(@PathVariable("id") int productId)
{
    Optional<Product> product = productService.getOptionalProductById(productId);
    if (product.isPresent()) {
        return new ResponseEntity<>(product.get(), HttpStatus.OK);
    }
    return new ResponseEntity<>(HttpStatus.NOT_FOUND);
}
```

Exposing APIs: test GET /product/{id}



GET

http://localhost:8080/api/product/2

Params

Auth

Headers (7)

Body

Pre-req.

Tests

Settings

Query Params

	KEY	VALUE
	Key	Value

Body

Pretty

Raw

Preview

Visualize

JSON

```
1  {
2    "id": 2,
3    "name": "Samsung",
4    "description": "This is an Awesome Samsung",
5    "price": 300.4,
6    "seller": "Samsung Inc"
7  }
```

A get request to `http://localhost:8080/api/product/2` gives us the second object in the `productList`

Exposing APIs: test GET /product/{id}



GET

Params Authorization Headers (7) Body Pre-request Script Tests Settings

Query Params

KEY	VALUE	DESC
Key	Value	Desc

Body Cookies Headers (4) Test Results

Pretty Raw Preview Visualize Text

Status: 404 Not Found

Get request of an non-existing id

no return and status = Not Found

Exposing APIs: POST /product/add



- A POST request to /product/add adds the new product-object that is delivered with the request, and then returns the saved product-object for the user to view
- **@RequestBody**
- Says that the *body* of the incoming POST *request* is a Product object

```
@PostMapping("product/add")
public Product addProduct(@RequestBody Product newProduct){
    return productService.addProduct(newProduct);
}
```

- With the incoming Product object we get all the information about the new Product. We just need to add an id. In this case we set the id to "size()+1" in the ArrayList. Later on, the database will generate an id for us.

```
public Product addProduct(Product newProduct) {
    newProduct.setId(productList.size()+1);
    productList.add(newProduct);
    return productList.get(productList.size()-1);
}
```

Exposing APIs: test POST /product/add



The screenshot displays a REST client interface with a POST request to `http://localhost:8080/api/product/add`. The request body is a JSON object representing a product. The response is a 201 Created status with a JSON body containing the created product details. Red arrows highlight the POST method, the URL, the Body tab, the JSON format, the request body, and the response body.

Request:

- Method: POST
- URL: `http://localhost:8080/api/product/add`
- Body (JSON):

```
{  "name": "IPHONE Y",  "description": "This is another Awesome Iphone",  "price": 500.4,  "seller": "Apple Inc"}
```

Response:

- Status: 201 Created
- Time: 31 ms
- Size: 277 B
- Body (JSON):

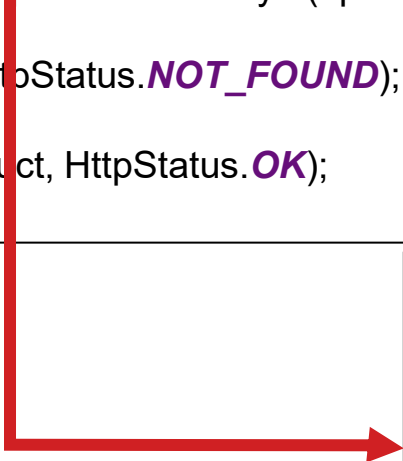
```
{  "id": 4,  "name": "IPHONE Y",  "description": "This is another Awesome Iphone",  "price": 500.4,  "seller": "Apple Inc"}
```

Exposing APIs: PUT /product/update/{id}



- A PUT request to /product/update/{id} updates the product-object with the given id in the ArrayList, and returns the updated product for the user to view.

```
@PutMapping("product/update/{id}")
public ResponseEntity<Product> updateProduct(@RequestBody Product updateProduct, @PathVariable("id") int productid){
    Product product = productService.updateProductById(updateProduct, productid);
    if (product==null) {
        return new ResponseEntity<>(HttpStatus.NOT_FOUND);
    }
    return new ResponseEntity<>(product, HttpStatus.OK);
}
```



```
public Product updateProductById(Product updateProduct, int productid) {
    Optional<Product> productOptional = getOptionalProductById(productid);
    if (productOptional.isPresent()){
        Product product = productOptional.get();
        product.setName(updateProduct.getName());
        product.setDescription(updateProduct.getDescription());
        product.setPrice(updateProduct.getPrice());
        product.setSeller(updateProduct.getSeller());
        return product;
    }
    return null;
}
```

Exposing APIs: PUT /product/update/{id} test



PUT ▼ http://localhost:8080/api/product/update/2

Params Authorization Headers (9) **Body** ● Pre-request Script Tests Settings

☐ none ☐ form-data ☐ x-www-form-urlencoded ☒ raw ☐ binary ☐ GraphQL **JSON** ▼

```
1 {
2   ... "name": "Samsung UPDATED",
3   ... "description": "This is an UPDATED Samsung",
4   ... "price": 200.4,
5   ... "seller": "Samsung Inc"
6 }
```

Body Cookies Headers (5) Test Results 🌐

Pretty Raw Preview Visualize **JSON** ▼ ≡

```
1 {
2   "id": 2,
3   "name": "Samsung UPDATED",
4   "description": "This is an UPDATED Samsung",
5   "price": 200.4,
6   "seller": "Samsung Inc"
7 }
```

Exposing APIs: PUT /product/update/{id} test



PUT ▼ http://localhost:8080/api/product/update/10

Params Authorization Headers (9) **Body** ● Pre-request Script Tests Settings

● none ● form-data ● x-www-form-urlencoded ● raw ● binary ● GraphQL JSON ▼

```
1 {
2   ... "name": "Samsung UPDATED",
3   ... "description": "This is an UPDATED Samsung",
4   ... "price": 200.4,
5   ... "seller": "Samsung Inc"
6 }
```

Body Cookies Headers (4) Test Results 🌐 Status: 404 Not Found

Pretty Raw Preview Visualize Text ▼

1

Put request of an non-existing id

no return and status = Not Found

Exposing APIs: test GET /products



GET http://localhost:8080/api/products

Params Authorization Headers (7) Body Pre-request Script Tests

Query Params

KEY	VALUE
Key	Value

Body Cookies Headers (5) Test Results

Pretty Raw Preview Visualize JSON

```
1 {
2   {
3     "id": 1,
4     "name": "IPHONE",
5     "description": "This is an Awesome Iphone",
6     "price": 400.4,
7     "seller": "Apple Inc"
8   },
9   {
10    "id": 2,
11    "name": "Samsung UPDATED",
12    "description": "This is an UPDATED Samsung",
13    "price": 200.4,
14    "seller": "Samsung Inc"
15  },
16  {
17    "id": 3,
18    "name": "LG Z",
19    "description": "This is an Awesome LG",
20    "price": 200.4,
21    "seller": "LG Inc"
22  },
23  {
24    "id": 4,
25    "name": "IPHONE Y",
26    "description": "This is another Awesome Iphone",
27    "price": 500.4,
28    "seller": "Apple Inc"
29  }
30 }
```

When we do a get request to retrieve all the elements of the list after the post and put request with the changed values we see the results.

Exposing APIs: DELETE /product/delete/{id}



- A DELETE request to /product/delete/{id} deletes the product-object with the given id and returns the size of the productList and sets the http status to 200 (=OK). If the id is not in the productList this method will return a http 404-status (= Not Found)

```
@DeleteMapping("product/delete/{id}")
public ResponseEntity<Integer> deleteProduct(@PathVariable("id") int productid){
    Optional<Product> product = productService.getOptionalProductById(productid);
    if (product.isPresent()){
        productService.getProductList().remove(product.get());
        return new ResponseEntity<>(productService.getProductList().size(), HttpStatus.OK);
    }
    return new ResponseEntity<>(productService.getProductList().size(), HttpStatus.NOT_FOUND);
}
```

Exposing APIs: test DELETE /product/delete/{id}



demoREST / http://localhost:8080/api/product/delete/3

DELETE http://localhost:8080/api/product/delete/3

Params Authorization Headers (7) Body Pre-request Script

Query Params

KEY	VALUE
Key	Value

Body Cookies Headers (5) Test Results

Pretty Raw Preview Visualize JSON

1 2

Delete request of an existing id

size of productList = 2 => return of the method

Get request to check the result

GET http://localhost:8080/api/products

Params Authorization Headers (7) Body Pre-request Script

Query Params

KEY	VALUE
Key	Value

Body Cookies Headers (5) Test Results

Pretty Raw Preview Visualize JSON

```
1 [
2   {
3     "id": 1,
4     "name": "IPHONE",
5     "description": "This is an Awesome Iphone",
6     "price": 400.4,
7     "seller": "Apple Inc"
8   },
9   {
10    "id": 2,
11    "name": "Samsung",
12    "description": "This is an Awesome Samsung",
13    "price": 300.4,
14    "seller": "Samsung Inc"
15  }
16 ]
```

Exposing APIs: test DELETE /product/delete/{id}



DELETE ☐ http://localhost:8080/api/product/delete/10

Params Authorization Headers (7) Body Pre-request Script Tests Settings

Query Params

KEY	VALUE	DESCRIPTION
Key	Value	Description

Body Cookies Headers (4) Test Results

Pretty Raw Preview Visualize Text ☐

1

Delete request of an non-existing id

no return and status = Not Found

- What is REST?
- Java Example
 - Working with the Postman tool
 - Use of class "ResponseEntity<E>" and "Optional<E>"
 - GET, POST, PUT, DELETE
- **SwaggerUI automated documentation**
- Testscript in Postman tool

Generate documentation with Swagger



- Now our REST web service is ready
- We can use Swagger to automatically generate interactive API documentation
- Swagger allows you to describe the structure of your APIs so that machines can read them. The ability of APIs to describe their own structure is the root of all awesomeness in Swagger. Why is it so great? Well, by reading your API's structure, we can automatically build beautiful and interactive API documentation. We can also automatically generate client libraries for your API in many languages and explore other possibilities like automated testing. Swagger does this by asking your API to return a YAML or JSON that contains a detailed description of your entire API.

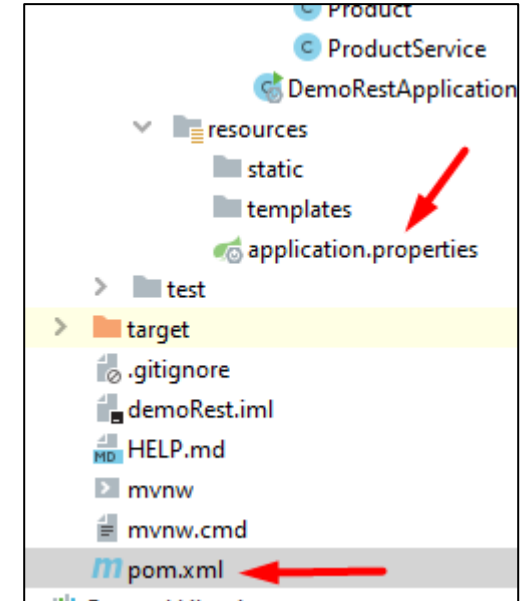
Documentation: SwaggerUI



- To enable SwaggerUI:
 - Add this code in de application.properties-file:
spring.mvc.pathmatch.matching-strategy= *ant-path-matcher*
- import the two dependencies into **pom.xml**:

```
<dependency>
  <groupId>io.springfox</groupId>
  <artifactId>springfox-swagger2</artifactId>
  <version>2.6.1</version>
</dependency>

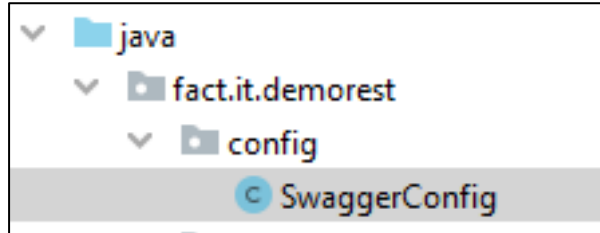
<dependency>
  <groupId>io.springfox</groupId>
  <artifactId>springfox-swagger-ui</artifactId>
  <version>2.6.1</version>
</dependency>
```



Documentation: SwaggerUI



- Make a class SwaggerConfig in a new package named: config



- This class gets the following contents,
there will be a lot of imports to be done as well.
You can also download this file,
with the necessary imports,
from Canvas.

```
import org.springframework.context.annotation.Configuration;

import com.google.common.base.Predicates;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import springfox.documentation.builders.ApiInfoBuilder;
import springfox.documentation.builders.PathSelectors;
import springfox.documentation.builders.RequestHandlerSelectors;
import springfox.documentation.service.ApiInfo;
import springfox.documentation.spi.DocumentationType;
import springfox.documentation.spring.web.plugins.Docket;
import springfox.documentation.swagger2.annotations.EnableSwagger2;

@Configuration
@EnableSwagger2
public class SwaggerConfig {
    @Bean
    public Docket api(){
        return new Docket(DocumentationType.SWAGGER_2)
            .apiInfo(apiInfo())
            .select()
            .apis(RequestHandlerSelectors.any())
            .paths(Predicates.not(PathSelectors.regex("/error.*")))
            .build();
    }

    private ApiInfo apiInfo(){
        return new ApiInfoBuilder()
            .title("Products API")
            .description("This page list all the rest apis for the products API.")
            .version("1.0-SNAPSHOT")
            .build();
    }
}
```


Documentation: SwaggerUI



- **Run and browse to:** <http://localhost:8080/swagger-ui.html>

The screenshot displays the SwaggerUI interface in a web browser. The address bar shows the URL `localhost:8080/swagger-ui.html#/`. The interface has a green header with the Swagger logo, a dropdown menu set to `default (/v2/api-docs)`, and an `Explore` button. Below the header, the title **Products API** is followed by the text `This page list all the rest apis for the products API.`

The main section is titled **product-rest-controller : Product Rest Controller**. To the right of this title are three links: `Show/Hide`, `List Operations` (which is highlighted with a red arrow), and `Expand Operations`.

Below the title, there is a list of API endpoints, each with a colored background indicating the HTTP method:

Method	Endpoint	Operation Name
POST	<code>/api/product/add</code>	<code>addProduct</code>
DELETE	<code>/api/product/delete/{id}</code>	<code>deleteProduct</code>
PUT	<code>/api/product/update/{id}</code>	<code>updateProduct</code>
GET	<code>/api/product/{id}</code>	<code>getProductById</code>
GET	<code>/api/products</code>	<code>getAllProducts</code>

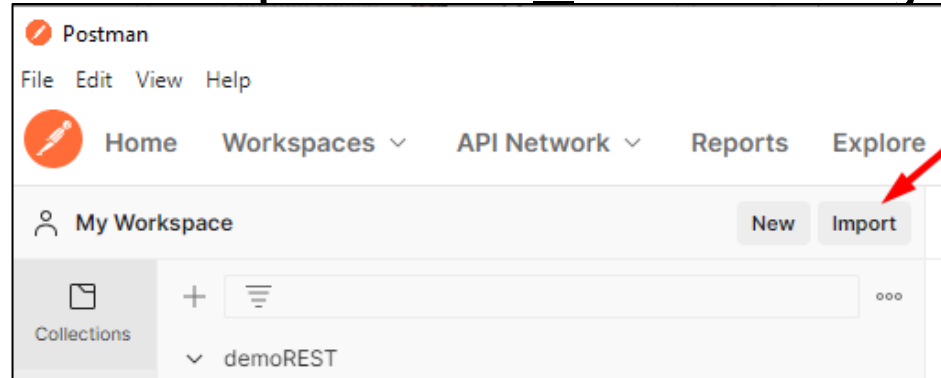
At the bottom of the interface, there is a footer that reads `[BASE URL: / , API VERSION: 1.0-SNAPSHOT]`.

- What is REST?
- Java Example
 - Working with the Postman tool
 - Use of class "ResponseEntity<E>" and "Optional<E>"
 - GET, POST, PUT, DELETE
- SwaggerUI automated documentation
- **Testscript in Postman tool**

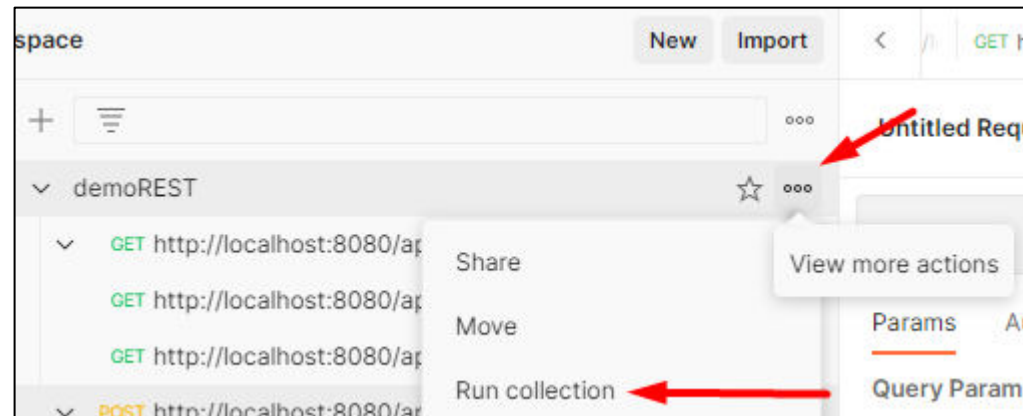
Run Testscript with Postman



- Download “demoREST.postman_collection.json” and import the collection in Postman



- (Re)Start your application and Run the imported collection:



- Click 

Result



- hopefully...

demoREST No Environment, just now

All Tests Passed (15) Failed (0)

Iteration 1

GET <http://localhost:8080/api/products> <http://localhost:8080/api/products>

Pass Status code is 200

Pass Number of records in resultset is 4

Pass Correct resultset

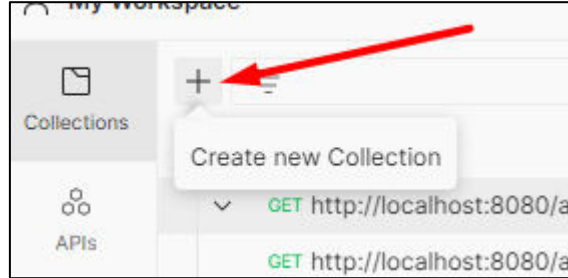
GET <http://localhost:8080/api/product/2> <http://localhost:8080/api/product/2>

Pass Status code is 200

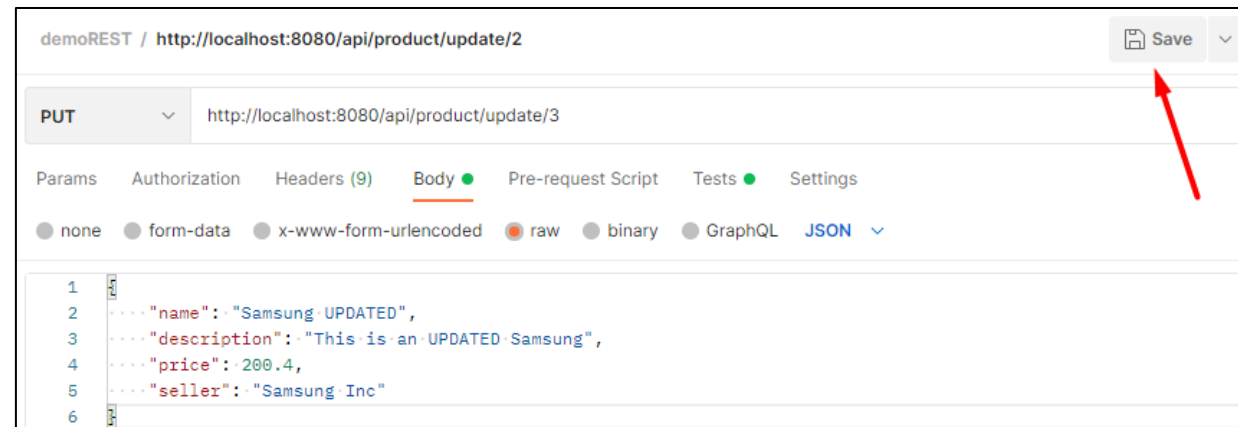
How to create your own testscript



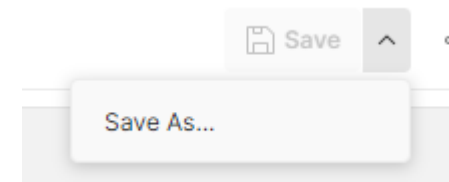
- Create a new collection



- Save every http-request to this collection and add tests to every http-request.



- If you enter a new request use "Save as"



How to create your own testscript



- Add tests to every http-request:

A screenshot of the Postman application interface. At the top, a dropdown menu shows "PUT" and the URL "http://localhost:8080/api/product/update/2". Below this is a tabbed interface with "Params", "Authorization", "Headers (9)", "Body", "Pre-request Script", "Tests", and "Settings". The "Tests" tab is selected and highlighted with a red underline. A red arrow points to the "Tests" tab. The test script content is as follows:

```
1 pm.test("Status code is 200", function(){
2   pm.response.to.have.status(200);
3 });
4
5 var expectedJsonBody =
6 {
7   "id": 2,
8   "name": "Samsung UPDATED",
9   "description": "This is an UPDATED Samsung",
10  "price": 200.4,
```

- Tip: Use your correct(!) output to make these tests and look at the examples in the provided *demoREST.postman_collection.json*

What did we learn?



- how to build a REST API in Spring Boot
- how to avoid Exceptions using *ResponseEntity* and *Optional*:
 - in stead of catching errors, it is better to implement a design that makes it difficult for errors to happen. *ResponseEntity* and *Optional* help you to avoid errors.
- how to generate documentation for this service
- how to create your own testscript in Postman