[References: 1](#_Toc51676929)

[General Questions: 2](#_Toc51676930)

[1) What is REST 2](#_Toc51676931)

[2) What is Resource? 3](#_Toc51676932)

[3) Identify resource 3](#_Toc51676933)

[4) HTTP Methods or VERB 4](#_Toc51676934)

[5) Naming conventions of REST API 4](#_Toc51676935)

[6) HTTP methods usage in CURD 8](#_Toc51676936)

[7) Steps to build a REST API? 8](#_Toc51676937)

[8) Ways to create REST api in java? 9](#_Toc51676938)

[9) Spring MVC architecture 9](#_Toc51676939)

[10) What is @RequestMapping? 10](#_Toc51676940)

[11) @PathVAriable 10](#_Toc51676941)

[12) @REquestParam 11](#_Toc51676942)

[13) Tools to test the rest API 11](#_Toc51676943)

[14) Creating rest api with spring/spring boot 11](#_Toc51676944)

[15) Error handling for REST API 11](#_Toc51676945)

[REST API develop step by step 12](#_Toc51676946)

[16) How to create REST Service in Spring boot? 12](#_Toc51676947)

[17) Response Entity 13](#_Toc51676948)

[18) How to use Model in REST? 14](#_Toc51676949)

[19) How to return XML response from rest api? 14](#_Toc51676950)

[<dependency> 14](#_Toc51676951)

[20) Content negotiation, how to return xml and json both response for a given rest api in 2 different UI applications? 15](#_Toc51676952)

[21) Error handling in REST 15](#_Toc51676953)

[22) Spring boot default Error response? 18](#_Toc51676954)

[23) How to do input validation in REST API 21](#_Toc51676955)

[24) How to add custom validation in spring boot rest API? 24](#_Toc51676956)

[25) Externalization or i18n? 24](#_Toc51676957)

[26) HATEOAS 24](#_Toc51676958)

[27) Swagger or documentation 24](#_Toc51676959)

[28) Versioning 29](#_Toc51676960)

[29) How to implement versioning in swagger? 32](#_Toc51676961)

[30) Customizing swagger 32](#_Toc51676962)

[31) Paging and Sorting 32](#_Toc51676963)

[32) Sorting 37](#_Toc51676964)

[33) REST java client 38](#_Toc51676965)

[34) Testing 38](#_Toc51676966)

[35) Security 38](#_Toc51676967)

# References:

Balaji Varanasi book

<https://restfulapi.net/rest-put-vs-post/>

<https://howtodoinjava.com/spring-restful/spring-rest-crud-jpa-example/>

<https://www.springboottutorial.com/spring-boot-swagger-documentation-for-rest-services>

# General Questions:

## What is REST

REST stands for REpresentational State Transfer and is an architectural style for designing distributed network applications. Roy Fielding coined the term REST in his PhD dissertation1 and proposed the following six constraints or principles as its basis:

• Client-Server—Concerns should be separated between clients and servers. This enables client and server components to evolve independently and in turn allows the system to scale.

• Stateless—The communication between client and server should be stateless. The server need not remember the state of the client. Instead, clients must include all of the necessary information in the request so that server can understand and process it.

• Layered System—Multiple hierarchical layers such as gateways, firewalls, and proxies can exist between client and server. Layers can be added, modified, reordered, or removed transparently to improve scalability.

• Cache—Responses from the server must be declared as cacheable or noncacheable. This would allow the client or its intermediary components to cache responses and reuse them for later requests. This reduces the load on the server and helps improve the performance.

Uniform Interface— All interactions between client, server, and intermediary components are based on the uniformity of their interfaces. This simplifies the overall architecture as components can evolve independently as long as they implement the agreed-on contract. The uniform interface constraint is further broken down into four subconstraints—resource identification, resource representations, self-descriptive messages, and hypermedia as the engine of application state or HATEOAS. We will examine some of these guiding principles in the later sections of this chapter

• Code on demand—Clients can extend their functionality by downloading and executing code on demand. Examples include JavaScript scripts, Java applets, Silverlight, and so on. This is an optional constraint.

## What is Resource?

Uniform Interface— All interactions between client, server, and intermediary components are based on the uniformity of their interfaces. This simplifies the overall architecture as components can evolve independently as long as they implement the agreed-on contract. The uniform interface constraint is further broken down into four subconstraints—resource identification, resource representations, self-descriptive messages, and hypermedia as the engine of application state or HATEOAS. We will examine some of these guiding principles in the later sections of this chapter. • Code on demand—Clients can extend their functionality by downloading and executing code on demand. Examples include JavaScript scripts, Java applets, Silverlight, and so on. This is an optional constraint.

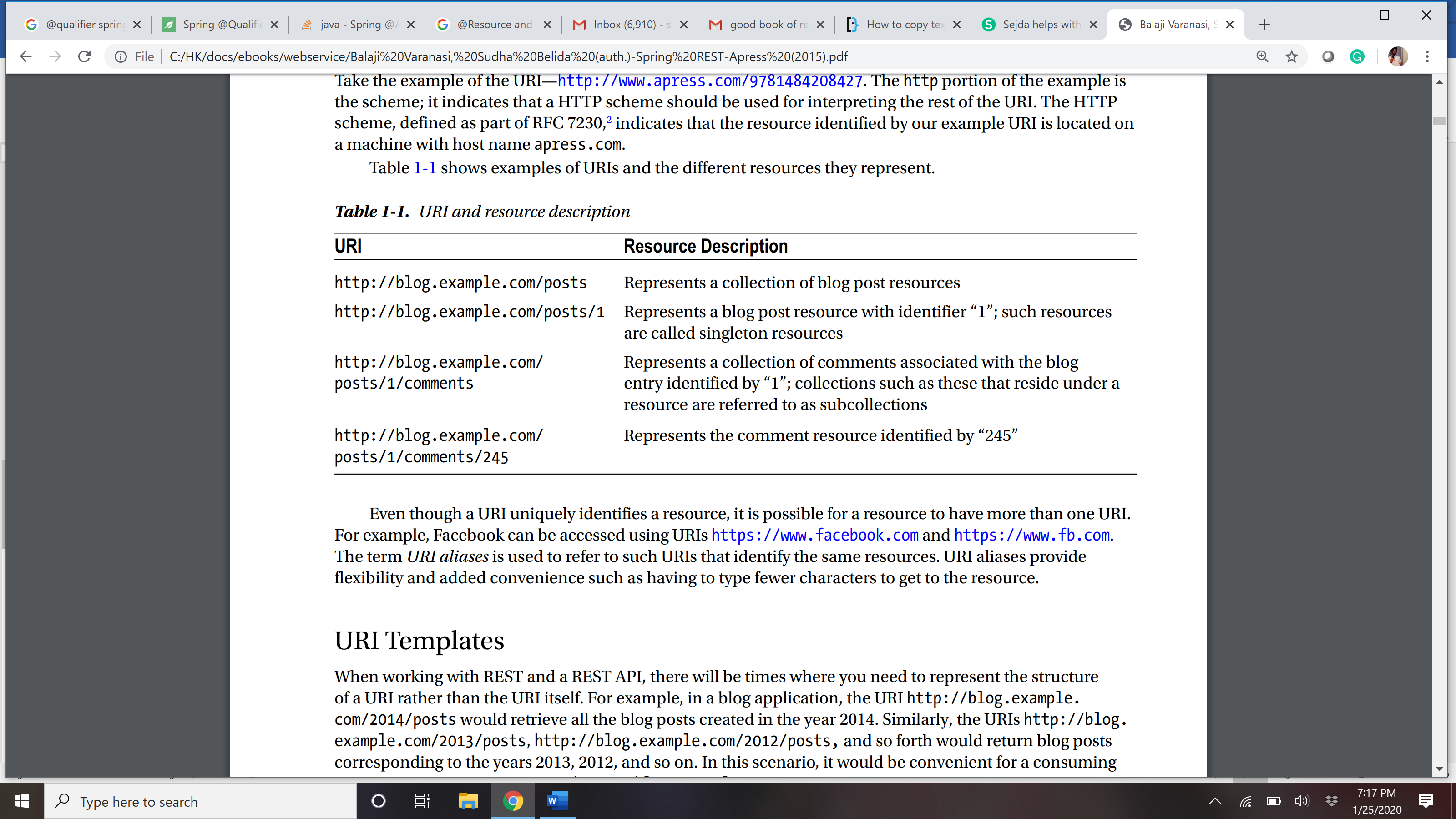
representation are snapshop at a given resource at a point of time. , the same resource can have several representations. These representations can range from text-based HTML, XML, and JSON formats to binary formats such as PDFs, JPEGs, and MP4s. It is possible for the client to request a particular representation and this process is termed as content negotiation. Here are the two possible content negotiation strategies:

• Postfixing the URI with the desired representation—In this strategy, a client requesting product details in JSON format would use the URI http://www.example. com/products/143.json. A different client might use the URI http://www.example. com/products/143.xml to get product details in XML format.

• Using the Accept header—Clients can populate the HTTP Accept header with the desired representation and send it along with the request. The application handling the resource would use the Accept header value to serialize the requested representation. The RFC 26163 provides a detailed set of rules for specifying one or more formats and their priorities.

## Identify resource

Scheme:scheme-specific-part



## HTTP Methods or VERB

## Naming conventions of REST API

### Use nouns to represent resources

RESTful URI should refer to a resource that is a thing (noun) instead of referring to an action (verb) because nouns have properties which verbs do not have – similar to resources have attributes. Some examples of a resource are:

* Users of the system
* User Accounts
* Network Devices etc.

and their resource URIs can be designed as below:

http://api.example.com/device-management/managed-devices

http://api.example.com/device-management/managed-devices/{device-id}

http://api.example.com/user-management/users/

http://api.example.com/user-management/users/{id}

For more clarity, let’s divide the **resource archetypes** into four categories (document, collection, store and controller) and then **you should always target to put a resource into one archetype and then use it’s naming convention consistently**. For uniformity’s sake, resist the temptation to design resources that are hybrids of more than one archetype.

#### document

A document resource is a singular concept that is akin to an object instance or database record. In REST, you can view it as a single resource inside resource collection. A document’s state representation typically includes both fields with values and links to other related resources.

Use “singular” name to denote document resource archetype.

http://api.example.com/device-management/managed-devices/{device-id}

http://api.example.com/user-management/users/{id}

http://api.example.com/user-management/users/admin

#### collection

A collection resource is a server-managed directory of resources. Clients may propose new resources to be added to a collection. However, it is up to the collection to choose to create a new resource or not. A collection resource chooses what it wants to contain and also decides the URIs of each contained resource.

Use “plural” name to denote collection resource archetype.

http://api.example.com/device-management/managed-devices

http://api.example.com/user-management/users

http://api.example.com/user-management/users/{id}/accounts

#### store

A store is a client-managed resource repository. A store resource lets an API client put resources in, get them back out, and decide when to delete them. A store never generates new URIs. Instead, each stored resource has a URI that was chosen by a client when it was initially put into the store.

Use “plural” name to denote store resource archetype.

http://api.example.com/cart-management/users/{id}/carts

http://api.example.com/song-management/users/{id}/playlists

#### controller

A controller resource models a procedural concept. Controller resources are like executable functions, with parameters and return values; inputs and outputs.

Use “verb” to denote controller archetype.

http://api.example.com/cart-management/users/{id}/cart/checkout

http://api.example.com/song-management/users/{id}/playlist/play

### Consistency is the key

Use consistent resource naming conventions and URI formatting for minimum ambiguily and maximum readability and maintainability. You may implement below design hints to achieve consistency:

#### Use forward slash (/) to indicate hierarchical relationships

The forward slash (/) character is used in the path portion of the URI to indicate a hierarchical relationship between resources. e.g.

http://api.example.com/device-management

http://api.example.com/device-management/managed-devices

http://api.example.com/device-management/managed-devices/{id}

http://api.example.com/device-management/managed-devices/{id}/scripts

http://api.example.com/device-management/managed-devices/{id}/scripts/{id}

#### Do not use trailing forward slash (/) in URIs

As the last character within a URI’s path, a forward slash (/) adds no semantic value and may cause confusion. It’s better to drop them completely.

http://api.example.com/device-management/managed-devices/

http://api.example.com/device-management/managed-devices /\*This is much better version\*/

#### Use hyphens (-) to improve the readability of URIs

To make your URIs easy for people to scan and interpret, use the hyphen (-) character to improve the readability of names in long path segments.

http://api.example.com/inventory-management/managed-entities/{id}/install-script-location //More readable

http://api.example.com/inventory-management/managedEntities/{id}/installScriptLocation //Less readable

#### Do not use underscores ( \_ )

It’s possible to use an underscore in place of a hyphen to be used as separator – But depending on the application’s font, it’s possible that the underscore (\_) character can either get partially obscured or completely hidden in some browsers or screens.

To avoid this confusion, use hyphens (-) instead of underscores ( \_ ).

http://api.example.com/inventory-management/managed-entities/{id}/install-script-location //More readable

http://api.example.com/inventory\_management/managed\_entities/{id}/install\_script\_location //More error prone

#### Use lowercase letters in URIs

When convenient, lowercase letters should be consistently preferred in URI paths.

[RFC 3986](https://www.ietf.org/rfc/rfc3986.txt) defines URIs as case-sensitive except for the scheme and host components. e.g.

http://api.example.org/my-folder/my-doc //1

HTTP://API.EXAMPLE.ORG/my-folder/my-doc //2

http://api.example.org/My-Folder/my-doc //3

In above examples, 1 and 2 are same but 3 is not as it uses **My-Folder** in capital letters.

#### Do not use file extenstions

File extensions look bad and do not add any advantage. Removing them decreases the length of URIs as well. No reason to keep them.

Apart from above reason, if you want to highlight the media type of API using file extenstion then you should rely on the media type, as communicated through the Content-Type header, to determine how to process the body’s content.

http://api.example.com/device-management/managed-devices.xml /\*Do not use it\*/

http://api.example.com/device-management/managed-devices /\*This is correct URI\*/

### Never use CRUD function names in URIs

URIs should not be used to indicate that a CRUD function is performed. URIs should be used to uniquely identify resources and not any action upon them. HTTP request methods should be used to indicate which CRUD function is performed.

HTTP GET http://api.example.com/device-management/managed-devices //Get all devices

HTTP POST http://api.example.com/device-management/managed-devices //Create new Device

HTTP GET http://api.example.com/device-management/managed-devices/{id} //Get device for given Id

HTTP PUT http://api.example.com/device-management/managed-devices/{id} //Update device for given Id

HTTP DELETE http://api.example.com/device-management/managed-devices/{id} //Delete device for given Id

### Use query component to filter URI collection

Many times, you will come across requirements where you will need a collection of resources sorted, filtered or limited based on some certain resource attribute. For this, do not create new APIs – rather enable sorting, filtering and pagination capabilities in resource collection API and pass the input parameters as query parameters. e.g.

http://api.example.com/device-management/managed-devices

http://api.example.com/device-management/managed-devices?region=USA

http://api.example.com/device-management/managed-devices?region=USA&brand=XYZ

http://api.example.com/device-management/managed-devices?region=USA&brand=XYZ&sort=installation-date

## HTTP methods usage in CURD

Data-driven applications typically use the term CRUD to indicate four basic persistence functions— Create, Read, Update, and Delete. Some developers building REST applications have mistakenly associated the four popular HTTP verbs GET, POST, PUT, and DELETE with CRUD semantics. The typical association often seen is:

Create -> POST

Update -> PUT

Read -> GET

Delete -> DELETE

## Steps to build a REST API?

1. Identify Resources—Central to REST are resources. We start modeling different resources that are of interest to our consumers. Often, these resources can be the application’s domain or entities. However, a one-to-one mapping is not always required.

2. Identify Endpoints—The next step is to design URIs that map resources to endpoints. In Chapter 4, we will look at best practices for designing and naming endpoints.

3. Identify Actions—Identify the HTTP methods that can be used to perform operations on the resources.

1. Identify Responses—Identify the supported resource representation for the request and response along with the right status codes to be returned.

## Ways to create REST api in java?

RESTeasy

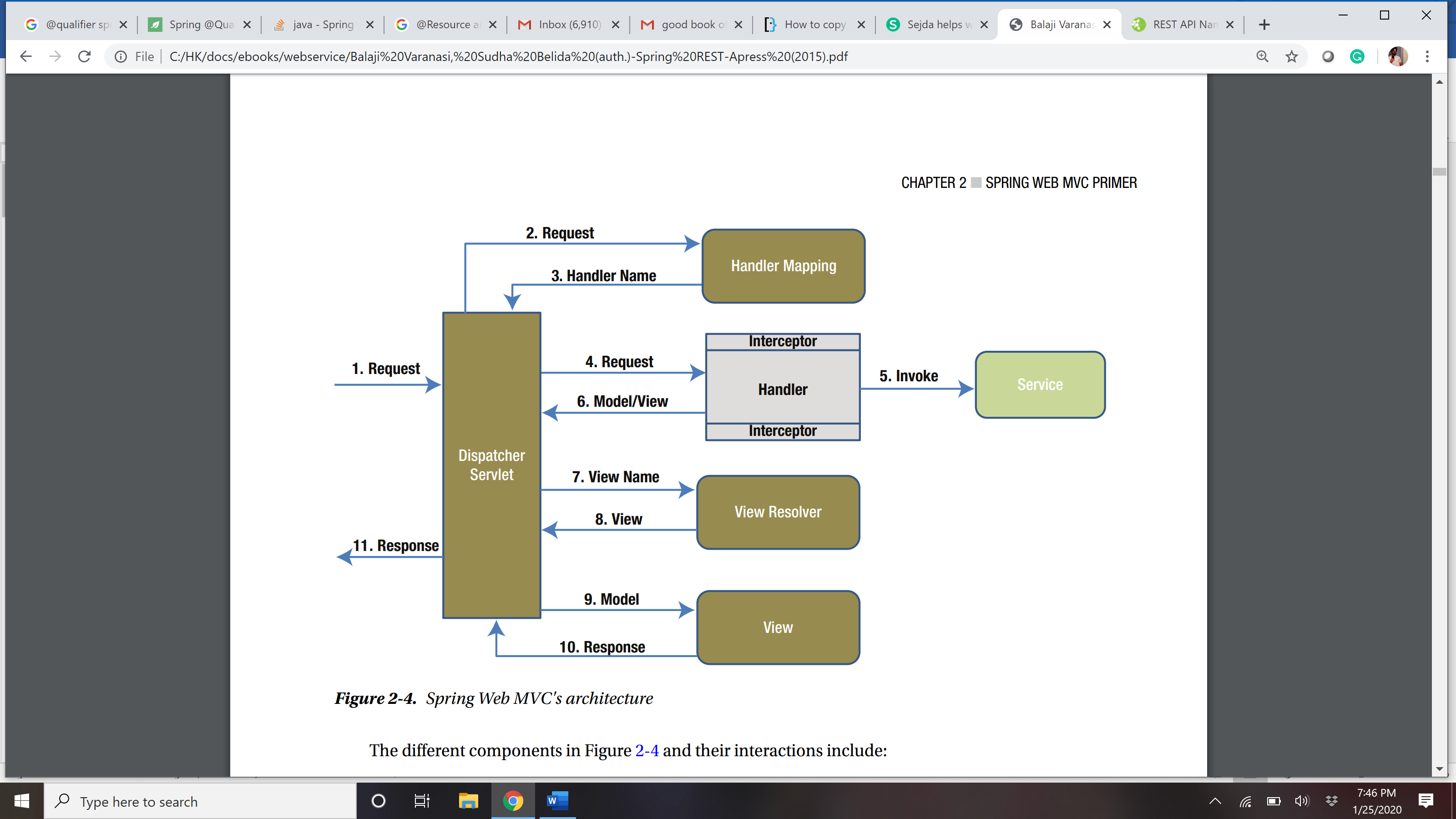
Axis

Jersy

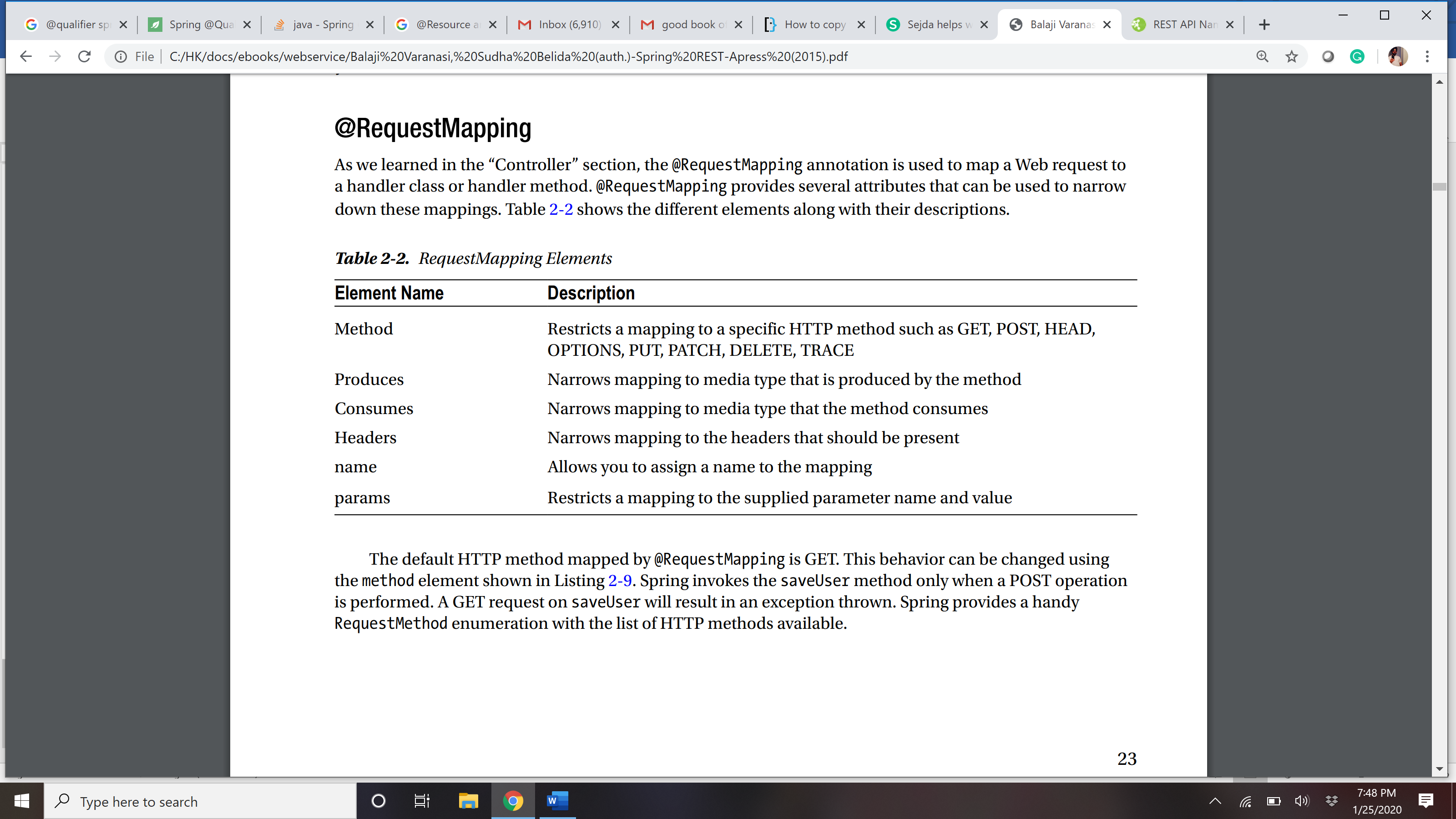
Spring

Etc

## Spring MVC architecture



## What is @RequestMapping?



@RequestMapping(value="/search.html", method=RequestMethod.GET, produces="MediaType. TEXT\_HTML")

public String search(@RequestParam String query, @RequestParam(value="page", required=false) int pageNumber) { model.put("currentDate", new Date());

return "home";

}

## @PathVAriable

The @RequestMapping annotation supports dynamic URIs via URI templates. As discussed in Chapter 1, URI templates are URIs with placeholders or variables. The @PathVariable annotation allows you to access and use these placeholders via method parameters. Listing 2-11 gives an example of @PathVariable. In this scenario, the getUser method is designed to serve user information associated with the path variable {username}. The client would perform a GET on the URL /users/jdoe to retrieve user information associated with username jdoe.

@RequestMapping("/users/{username}") public User getUser(@PathVariable("username") String username) { User user = null; // Code to construct user object using username return user; }

## @REquestParam

To get the parameter from request

## Tools to test the rest API

POSTMAN

RESTCLIENT – a firefox extension

## Creating rest api with spring/spring boot

## Error handling for REST API

* 1. Lets say when data does not exist for a given resource id, showing blank with 200 wont make any sense then we should throw the custom Exception as resourceNotFoundException with 404 status code.

import org.springframework.http.HttpStatus;

import org.springframework.web.bind.annotation.ResponseStatus;

@ResponseStatus(HttpStatus.NOT\_FOUND)

public class ResourceNotFoundException extends RuntimeException {

private static final long serialVersionUID = 1L;

public ResourceNotFoundException() {}

public ResourceNotFoundException(String message) {

super(message);

}

public ResourceNotFoundException(String message, Throwable cause) {

super(message, cause);

}

}

* 1. HTTP status codes play an important role in REST APIs. API developers should strive to return the right codes indicating the request status. Additionally, it is good practice to provide helpful, fine-grained details regarding the error in the response body. These details will enable API consumers to troubleshoot issues easily and help them to recover. As you can see in Figure 5-3, Spring Boot follows this practice and includes the following details in error response bodies: • timestamp—The time in milliseconds when the error happened • status—HTTP status code associated with the error; this is partly redundant as it is same as the response status code • error—The description associated with the status code Chapter 5 ■ Error Handling 77 • exception—The fully qualified path to the exception class resulting in this error • message—The message providing more details about the error • path—The URI that resulted in the exception



* 1. Create a ErrorDetail class that is useful to your business .

# REST API develop step by step

## How to create REST Service in Spring boot?

* + 1. Select web dependencies from spring stater
    2. Create a controller with Annotation @RestController (@Controller + @Response)
    3. Create methods with either @RequestMapping or @GetMapping(and other method annotations) example :

@RestController

public class HelloWorldRestApplication {

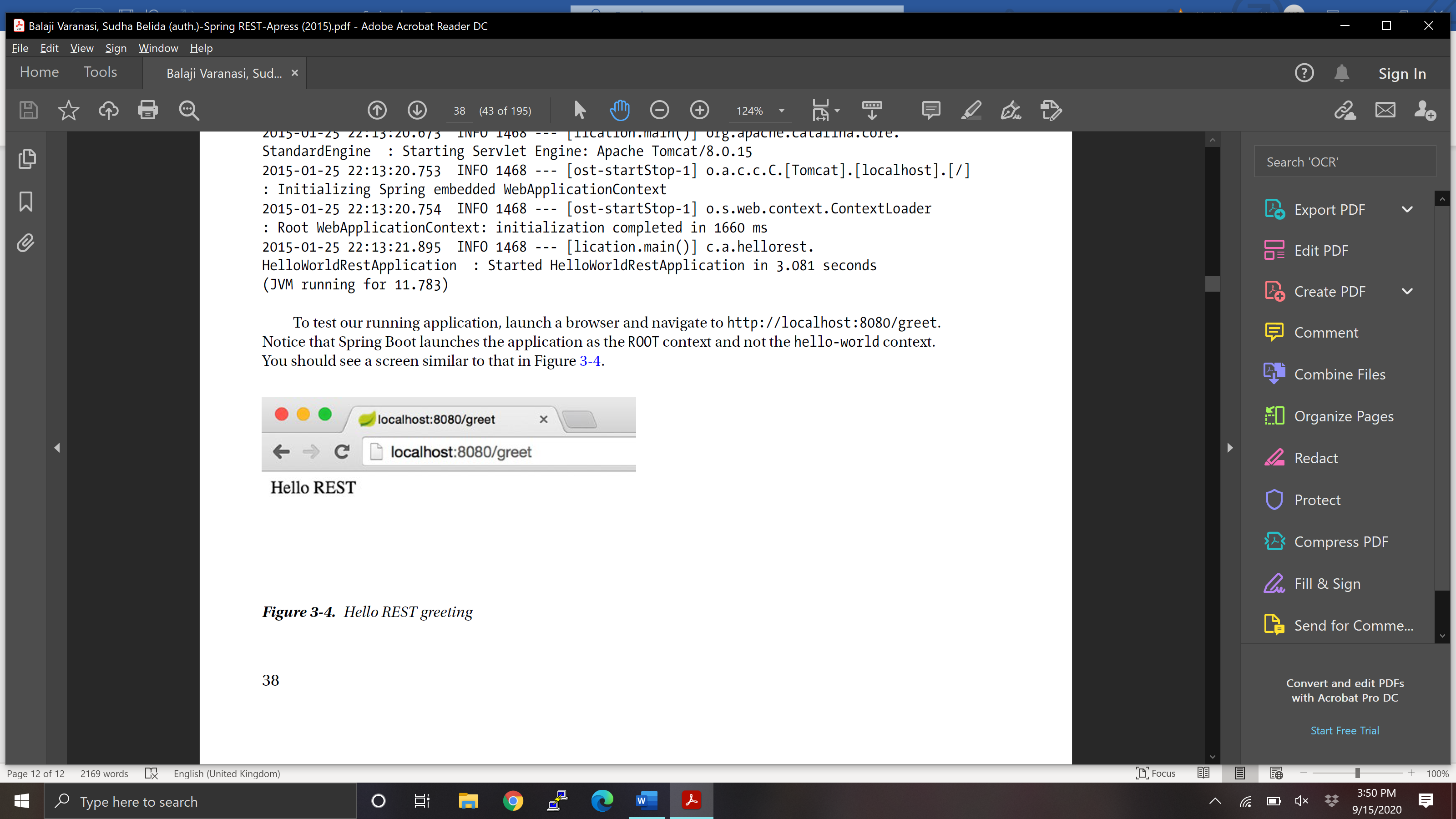
@RequestMapping("/greet")

public String helloGreeting() {

return "Hello REST";

}

}



* 1. To specify method in request mapping use –

@RequestMapping(value=”/path” , method=”METHODNAME”)

* 1. We can return Model as return value from the controller method that will produce corresponding JSON, but we should use ResponseEntity<DomainModel> to return other things as well.

Example:

## Response Entity

### *ResponseEntity* **represents the whole HTTP response: status code, headers, and body**. As a result, we can use it to fully configure the HTTP response.

### If we want to use it, we have to return it from the endpoint; Spring takes care of the rest.

### *ResponseEntity* is a generic type. Consequently, we can use any type as the response body:

|  |  |
| --- | --- |
|  | @GetMapping("/hello") |
|  | ResponseEntity<String> hello() { |
|  | **return** **new** ResponseEntity<>(**"Hello World!"**, HttpStatus.OK); |
|  | } |

@GetMapping("/customHeader")

ResponseEntity<String> customHeader() {

HttpHeaders headers = new HttpHeaders();

headers.add("Custom-Header", "foo");

return new ResponseEntity<>(

"Custom header set", headers, HttpStatus.OK);

}

Another example with @Controllere only-

@Controller

public class MyController {

@RequestMapping(value = "/getCountry")

public ResponseEntity<Country> getCountry() {

var c = new Country();

c.setName("France");

c.setPopulation(66984000);

var headers = new HttpHeaders();

headers.add("Responded", "MyController");

return ResponseEntity.accepted().headers(headers).body(c);

}

@RequestMapping(value = "/getCountry2")

@ResponseBody

public Country getCountry2() {

var c = new Country();

c.setName("France");

c.setPopulation(66984000);

return c;

}

}

## How to use Model in REST?

### Just return any model from your method directly or use responseEntity to send your model and status. This will produce JSON output by default.

## How to return XML response from rest api?

## <dependency>

<groupId>com.fasterxml.jackson.dataformat</groupId>

<artifactId>jackson-dataformat-xml</artifactId>

</dependency

\*\* Since the JAXB API was removed from the Java SE in Java 11, we need to add the jaxb-api dependency as well.

Use produces = MediaType.APPLICATION\_XML\_VALUE in @Request Mapping

@RequestMapping(value = "/remote/search", method = RequestMethod.GET, produces = MediaType.APPLICATION\_XML\_VALUE)

public List<Sdn> search(@ModelAttribute SdnSearch sdnSearch) {

List<Sdn> foundSdns = sdnSearchService.find( sdnSearch );

return foundSdns;

}

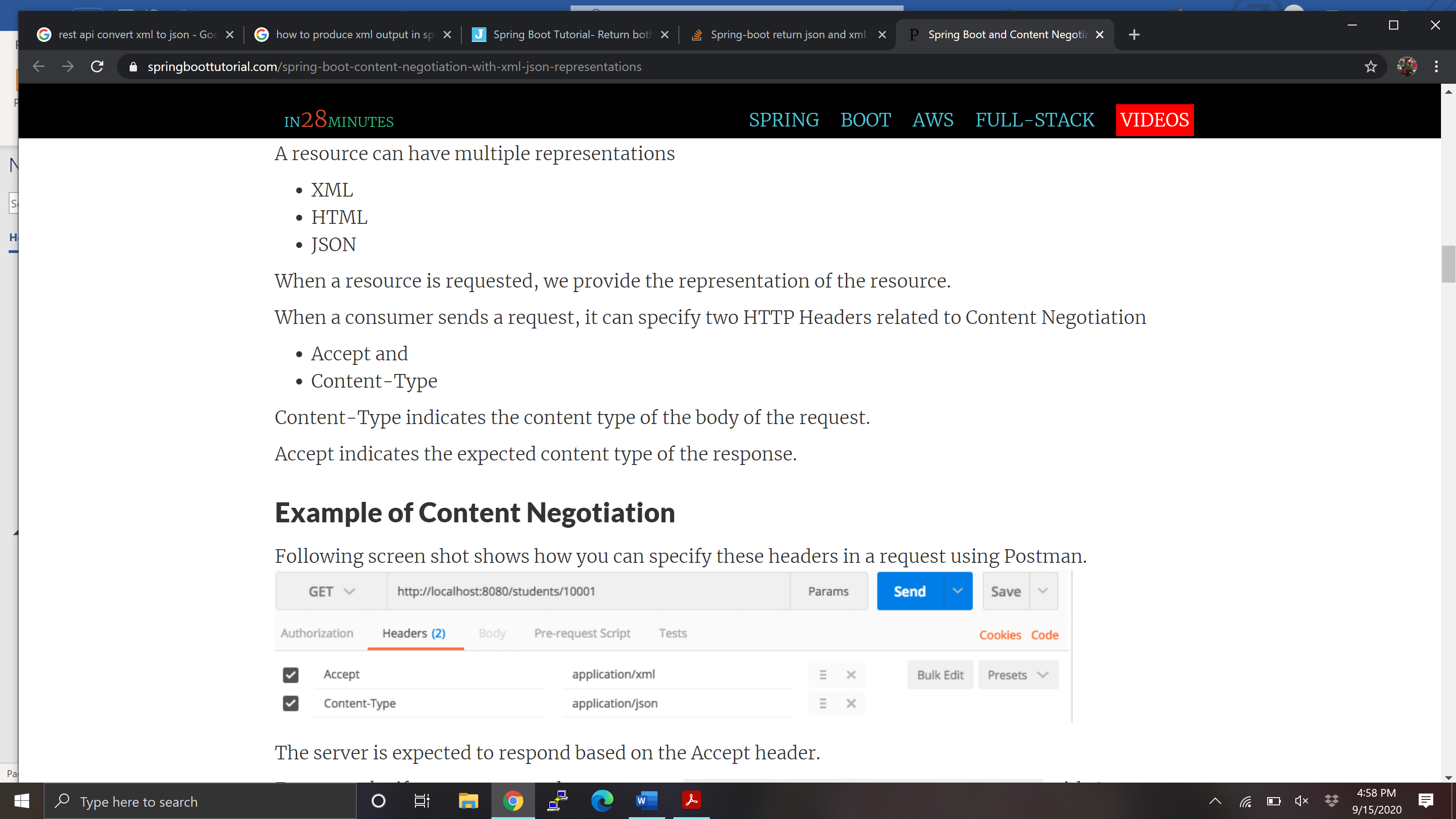
## Content negotiation, how to return xml and json both response for a given rest api in 2 different UI applications?

<https://www.javainuse.com/spring/spring-boot-content-negotiation>

<https://www.springboottutorial.com/spring-boot-content-negotiation-with-xml-json-representations>

@RequestMapping(value = "/employee", method = RequestMethod.GET,

**produces = { "application/json", "application/xml" })**



## Error handling in REST

<https://www.baeldung.com/spring-response-status-exception>

<https://spring.io/blog/2013/11/01/exception-handling-in-spring-mvc>

### **Before Spring 3.2, the two main approaches to handling exceptions in a Spring MVC application were**HandlerExceptionResolver**or the**@ExceptionHandler**annotation.** Both have some clear downsides.

### Since 3.2, we've had the @ControllerAdvice annotation to address the limitations of the previous two solutions and to promote a unified exception handling throughout a whole application.

### Now Spring 5 introduces the ResponseStatusException class — a fast way for basic error handling in our REST APIs.

### There are multiple ways to handle the exception-

### **Solution1:**

### Create Custom Exception class and annotate this @ResponseStaus(HttpStatus.NOT\_FOUND) or whatever

@ResponseStatus(HttpStatus.NOT\_FOUND)

public class ResourceNotFoundException extends RuntimeException {

private static final long serialVersionUID = 1L;

public ResourceNotFoundException() {}

public ResourceNotFoundException(String message) {

super(message);

}

public ResourceNotFoundException(String message, Throwable cause) {

super(message, cause);

}

}

### Throw exception from controller

@RequestMapping(value="/polls/{pollId}", method=RequestMethod.GET)

public ResponseEntity<?> getPoll(@PathVariable Long pollId) {

Poll p = pollRepository.findOne(pollId);

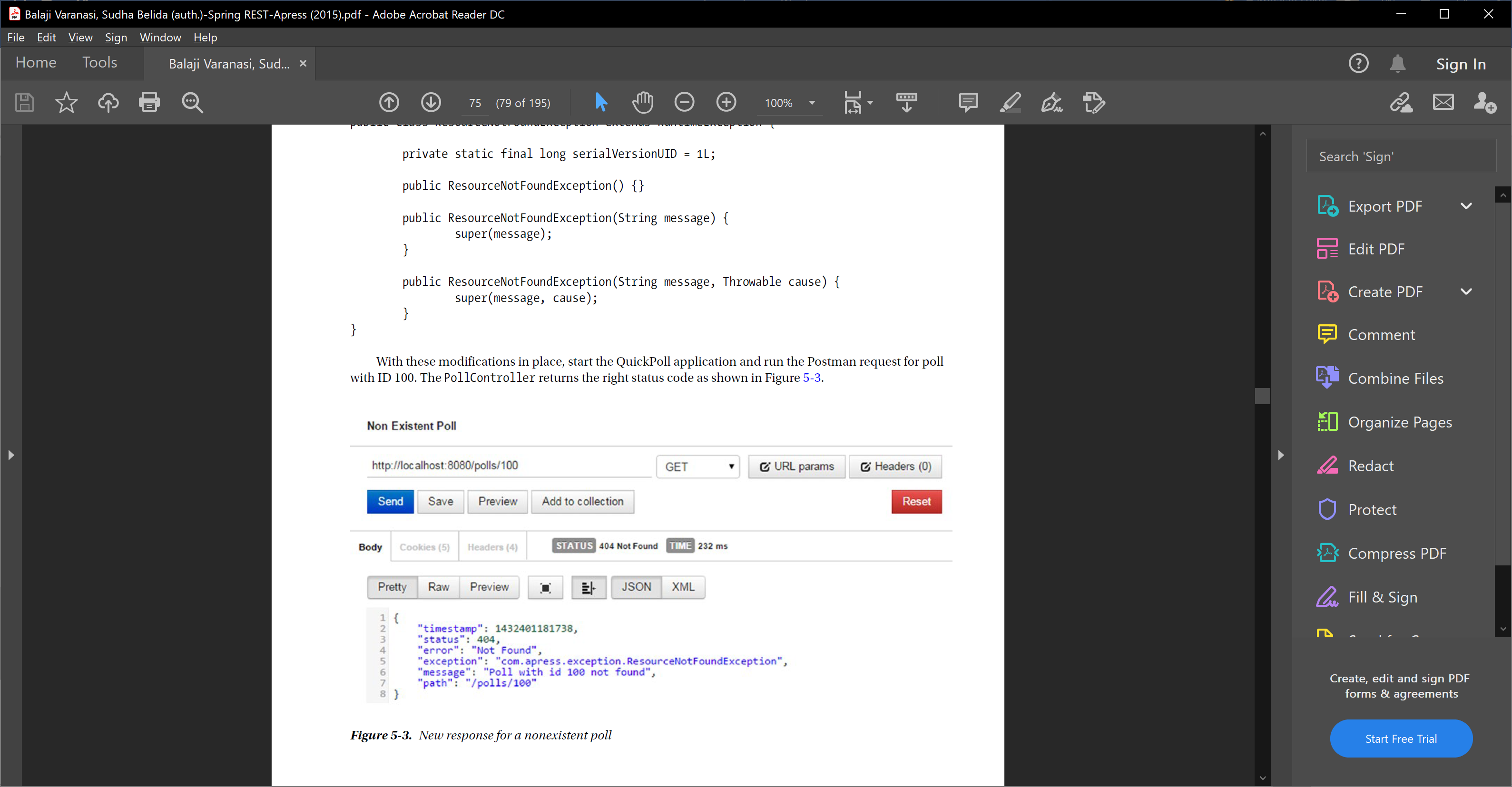
if(p == null) {

throw new ResourceNotFoundException("Poll with id " + pollId + " not found");

}

return new ResponseEntity<> (p, HttpStatus.OK);

}



**Solution 2: Global Exception handler**

**@ControllerAdvice – introduced inspring 3.2**

### Create ErrorDetail class, this is optional but good practice to create own Error response class.

public class ErrorDetail {

private String title;

private int status;

private String detail;

private long timeStamp;

private String developerMessage;

// Getters and Setters ommited for brevity

}

### Create a class , say RestExceptionHandler, extend this with REsponseEntityExceptionHandler and annotate this @ControllerAdvice, Eithere override existing method or create new wmethod with annotation @ExceptionHandler –

@ControllerAdvice

public class RestExceptionHandler {

@ExceptionHandler(ResourceNotFoundException.class)

public ResponseEntity<?> handleResourceNotFoundException(ResourceNotFoundException

rnfe, HttpServletRequest request) {

ErrorDetail errorDetail = new ErrorDetail();

errorDetail.setTimeStamp(new Date().getTime());

errorDetail.setStatus(HttpStatus.NOT\_FOUND.value());

errorDetail.setTitle("Resource Not Found");

errorDetail.setDetail(rnfe.getMessage());

errorDetail.setDeveloperMessage(rnfe.getClass().getName());

return new ResponseEntity<>(errorDetail, null, HttpStatus.NOT\_FOUND);

}

}

## Spring boot default Error response?

### HTTP status codes play an important role in REST APIs. API developers should strive to return the right

### codes indicating the request status. Additionally, it is good practice to provide helpful, fine-grained details

### regarding the error in the response body. These details will enable API consumers to troubleshoot issues

### easily and help them to recover. As you can see in Figure 5-3, Spring Boot follows this practice and includes

### the following details in error response bodies:

### • timestamp—The time in milliseconds when the error happened

### • status—HTTP status code associated with the error; this is partly redundant as it is

### same as the response status code

### • error—The description associated with the status code

### • exception—The fully qualified path to the exception class resulting in this error

### • message—The message providing more details about the error

### • path—The URI that resulted in the exception

### These details are generated by the Spring Boot framework. This feature is not available out of the box in non–Boot Spring MVC applications

### Spring Boot provides good default implementation for exception handling for RESTful Services. Let’s quickly look at the default Exception Handling features provided by Spring Boot.

### Resource Not Present – when no custom exception is thrown expictly.

### Heres what happens when you fire a request to a non existent resource http://localhost:8080/some-dummy-url

### {

### "timestamp": 1512713804164,

### "status": 404,

### "error": "Not Found",

### "message": "No message available",

### "path": "/some-dummy-url"

### }

**When custom exception is thrown:**

### What happens when you throw an Exception?

### Let’s see what Spring Boot does when an exception is thrown from a Resource.

### Lets create a StudentNotFoundException.

### /src/main/java/com/in28minutes/springboot/rest/example/student/StudentNotFoundException.java

### public class StudentNotFoundException extends RuntimeException {

### public StudentNotFoundException(String exception) {

### super(exception);

### }

### }

### Let’s enhance the GET method to throw this exception when a student is not found.

### @GetMapping("/students/{id}")

### public Resource<Student> retrieveStudent(@PathVariable long id) {

### Optional<Student> student = studentRepository.findById(id);

### if (!student.isPresent())

### throw new StudentNotFoundException("id-" + id);

### Resource<Student> resource = new Resource<Student>(student.get());

### ControllerLinkBuilder linkTo = linkTo(methodOn(this.getClass()).retrieveAllStudents());

### resource.add(linkTo.withRel("all-students"));

### return resource;

### }

### This is the response when you try getting details of a non existing student http://localhost:8080/students/9.

### {

### "timestamp": 1512714275589,

### "status": 500,

### "error": "Internal Server Error",

### "message": "id-9",

### "path": "/students/9"

### }

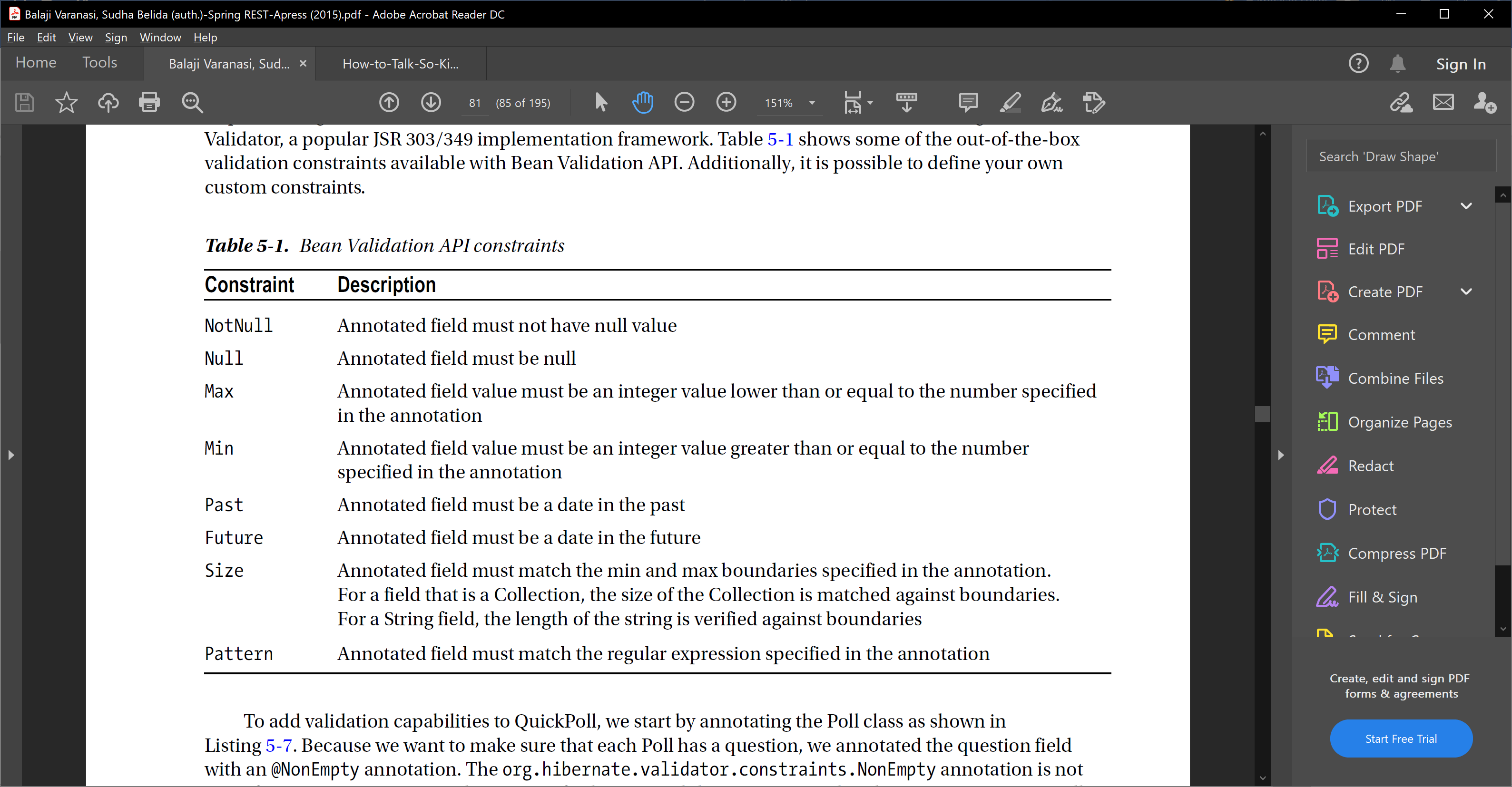
## How to do input validation in REST API

### This can be done using java validator framework or more popular hibernate validator framework-

### Spring MVC provides two options for validating user input. In the first option, we create a validator that implements the org.springframework.validation.Validator interface. Then we inject this validator into a controller and invoke validator’s validate method manually to perform validation.

### The second option is to use the JSR 303 validation, an API intended to simplify field validation in any layer of the application.

### Considering the simplicity and the declarative nature of the framework, we will be using JSR 303 validation framework in this book.



@Column(name="QUESTION")

@NotEmpty

private String question;

@OneToMany(cascade=CascadeType.ALL)

@JoinColumn(name="POLL\_ID")

@OrderBy

@Size(min=2, max = 6)

private Set<Option> options;

### We now move our attention to the com.apress.controller.PollController and add an @Valid

### annotation to the createPoll method’s Poll parameter, as shown in Listing 5-8. The @Valid annotation

### instructs Spring to perform data validation after binding the user-submitted data. Spring delegates the actual

### validation to a registered Validator. With Spring Boot adding JSR 303/JSR 349 and Hibernate validator jars to

### the class path, the JSR 303/JSR 349 is enabled automatically and will be used to perform the validation.

***Listing 5-8.*** PollController annotated with @Valid annotations

@RequestMapping(value="/polls", method=RequestMethod.POST)

public ResponseEntity<?> createPoll(@Valid @RequestBody Poll poll) {

poll = pollRepository.save(poll);

// Set the location header for the newly created resource

HttpHeaders responseHeaders = new HttpHeaders();

URI newPollUri = ServletUriComponentsBuilder

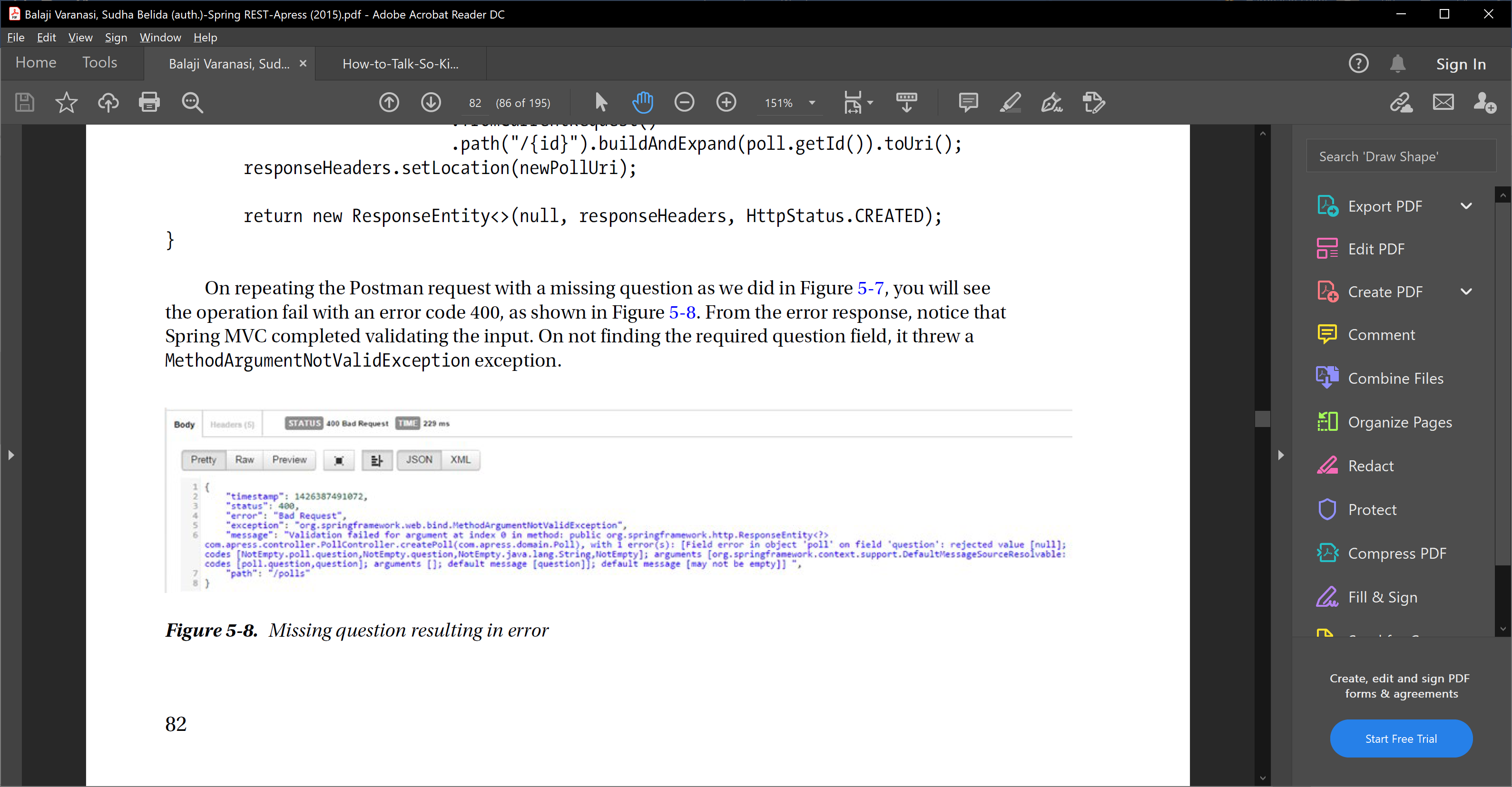
.fromCurrentRequest()

.path("/{id}").buildAndExpand(poll.getId()).toUri();

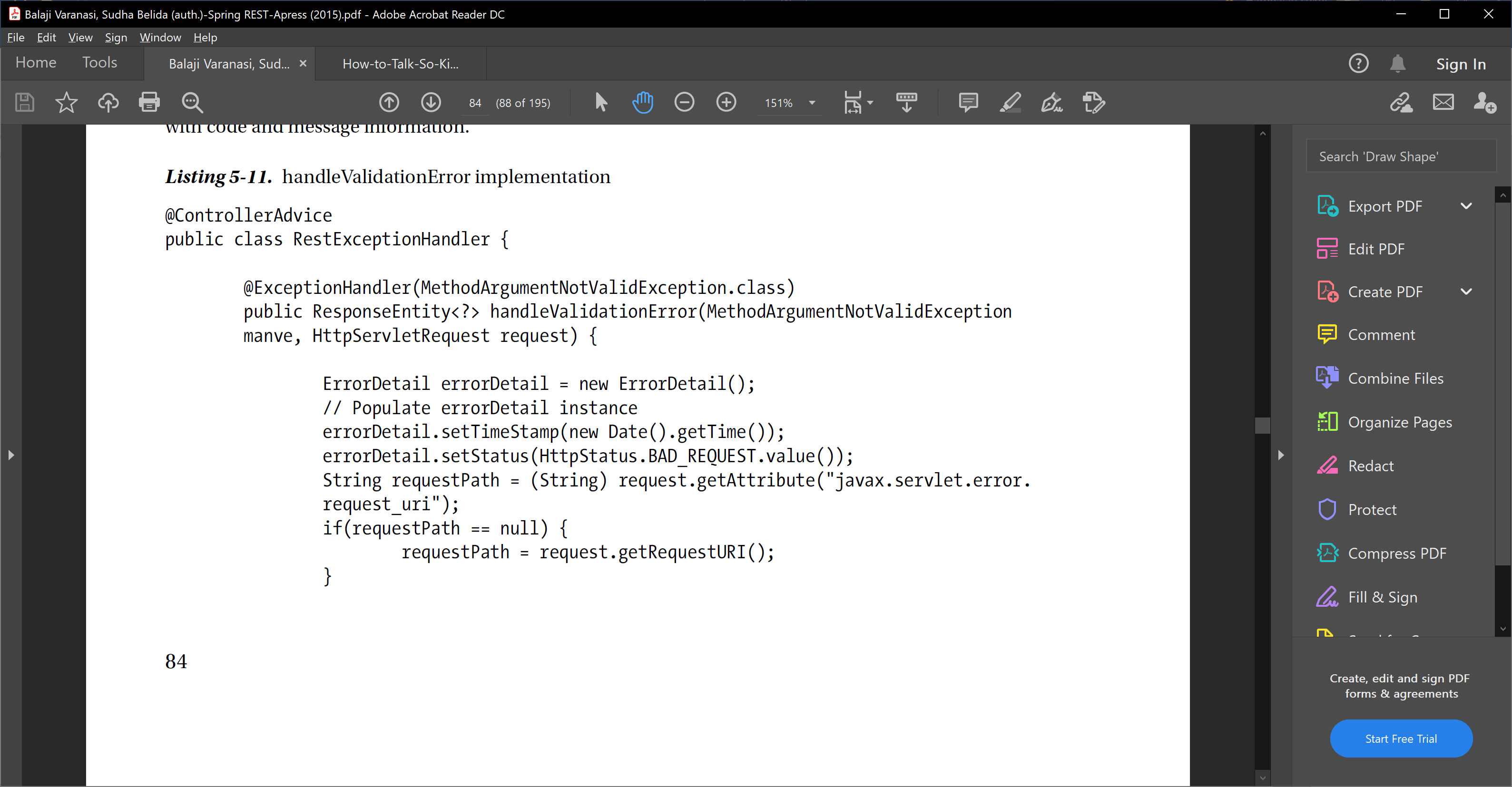
responseHeaders.setLocation(newPollUri);

return new ResponseEntity<>(null, responseHeaders, HttpStatus.CREATED);

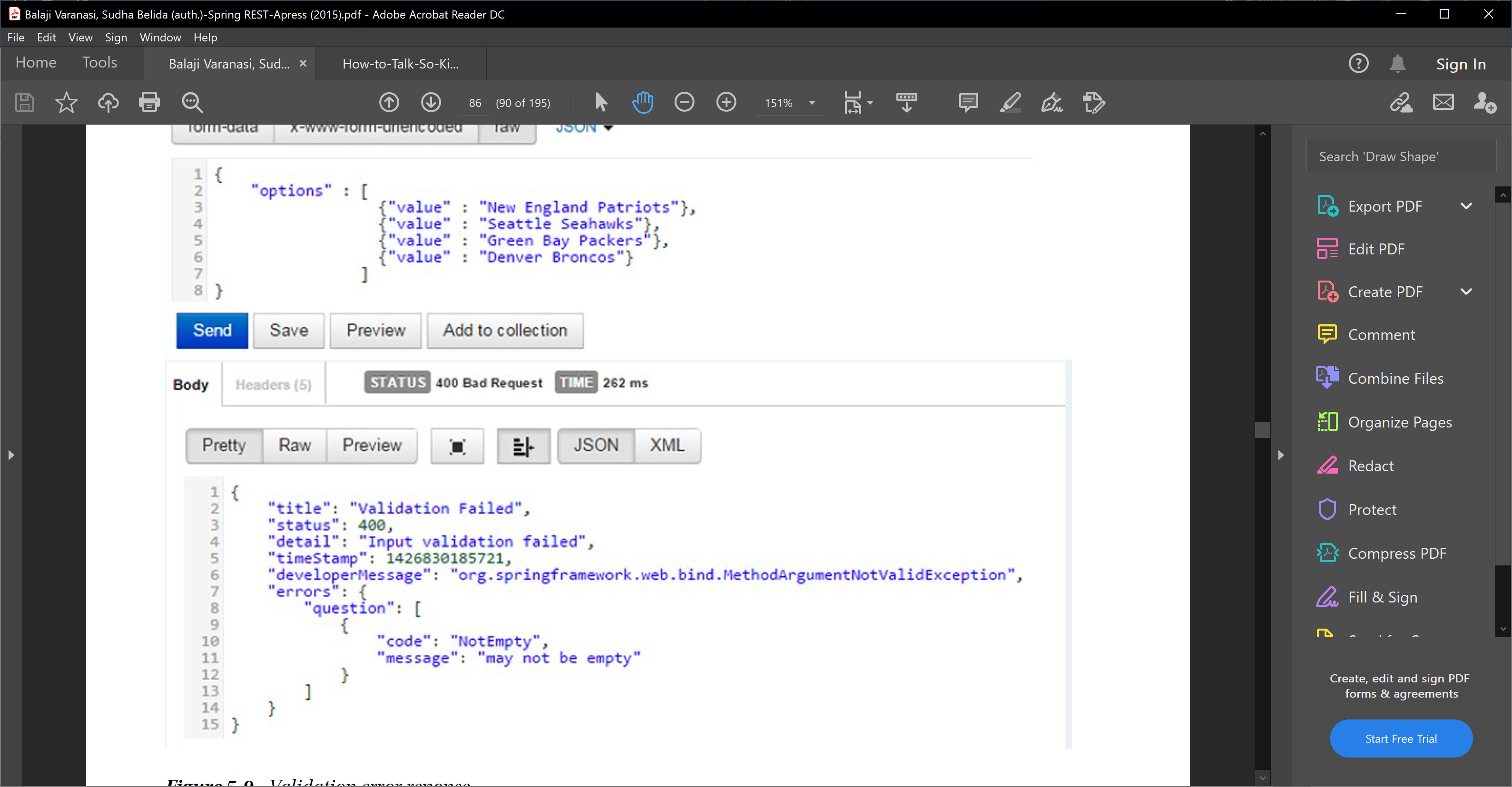
### }



To show appropriate Error message format –







<https://www.baeldung.com/spring-boot-bean-validation>

## How to add custom validation in spring boot rest API?

<https://dimitr.im/validating-the-input-of-your-rest-api-with-spring>

## Externalization or i18n?

## HATEOAS

## Swagger or documentation

### One option is to maintain documentation manually. But that gets outdated quickly.

### Other option is to generate documentation from code. And that’s the approach we would discuss in this guide.

### There are multiple approaches to documenting your RESTful API

### WADL

### RESTDocs

### Swagger or OpenDocs

### Swagger has picked up momentum in the last couple of years and is now the most popular REST API documentation standard. We will use Swagger in this guide.

There are two versions of swagger – swagger 1.0 and swagger 2.0

We are discussing here swagger 2.0 ->

### We would need to add a couple of dependencies related to Swagger and configure a Docket to generate Swagger Documentation. We will also use Swagger UI to have a visual representation of the Documentation and execute Test Requests.

### Adding Swagger Dependencies

### Let’s add a couple of dependencies to our Swagger Project pom.xml.

### <dependency>

### <groupId>io.springfox</groupId>

### <artifactId>springfox-boot-starter</artifactId>

### <version>3.0.0-SNAPSHOT</version>

### </dependency>

### Let’s now add the Spring configuration needed to generate Swagger Documentation.

### /src/main/java/com/in28minutes/springboot/rest/example/swagger/SwaggerConfig.java

### @Configuration

### @EnableSwagger2WebMvc

### public class SwaggerConfig {

### public static final Contact DEFAULT\_CONTACT = new Contact(

### "Ranga Karanam", "http://www.in28minutes.com", "in28minutes@gmail.com");

### 

### public static final ApiInfo DEFAULT\_API\_INFO = new ApiInfo(

### "Awesome API Title", "Awesome API Description", "1.0",

### "urn:tos", DEFAULT\_CONTACT,

### "Apache 2.0", "http://www.apache.org/licenses/LICENSE-2.0",Arrays.asList());

### private static final Set<String> DEFAULT\_PRODUCES\_AND\_CONSUMES =

### new HashSet<String>(Arrays.asList("application/json",

### "application/xml"));

### @Bean

### public Docket api() {

### return new Docket(DocumentationType.SWAGGER\_2)

### .apiInfo(DEFAULT\_API\_INFO)

### .produces(DEFAULT\_PRODUCES\_AND\_CONSUMES)

### .consumes(DEFAULT\_PRODUCES\_AND\_CONSUMES);

### }

### }

### public Docket api() { - Docket to decide what kind of APIs you would want to document. In this example, we are documenting all APIs. You can filter out APIs you do not want to document with Swagger.

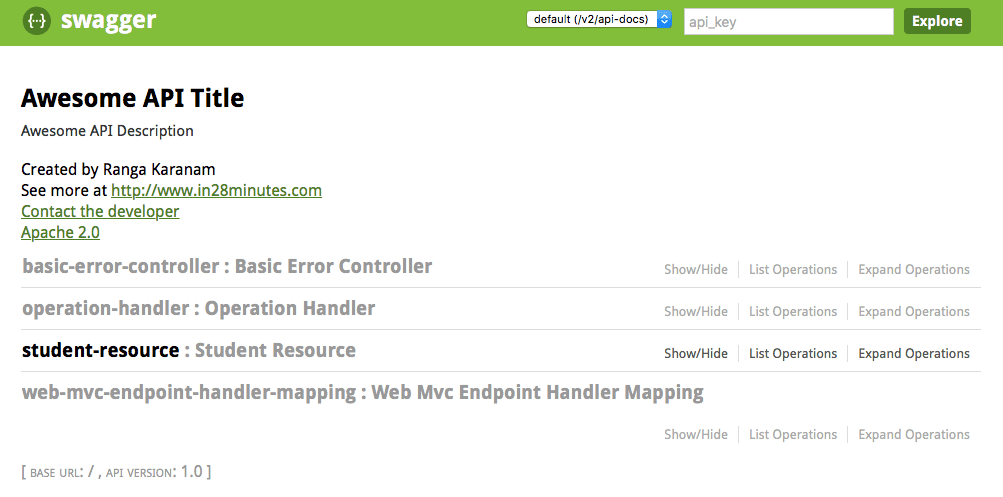
### When you restart the application, you are all set to view the documentation that is generated.

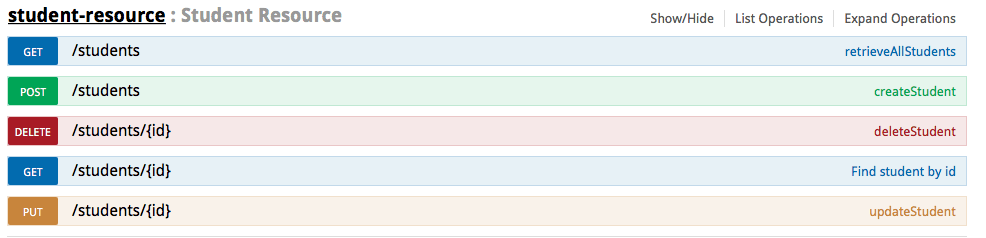
### Go to URL http://localhost:8080/v2/api-docs

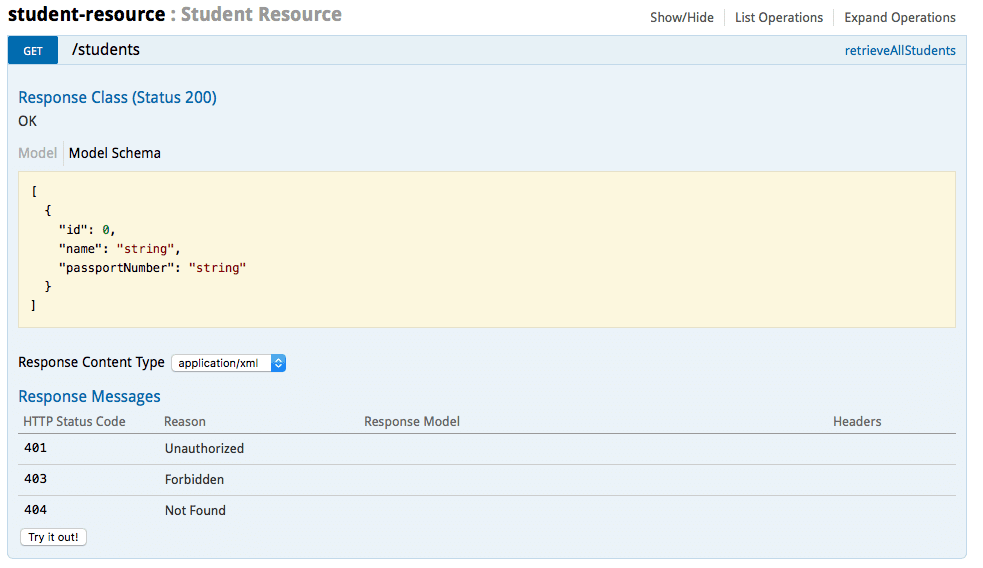
You can also use the Swagger UI available at http://localhost:8080/swagger-ui/index.html.

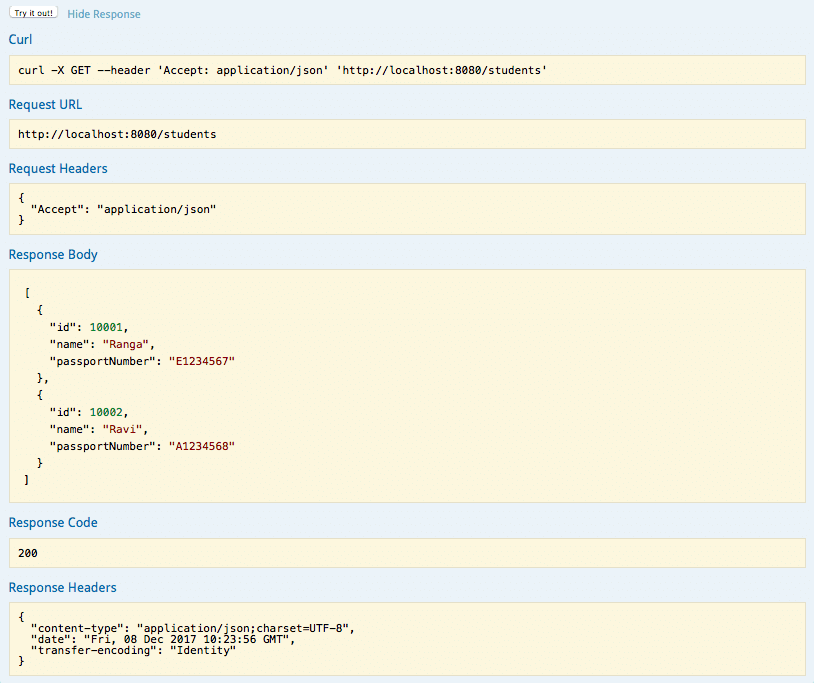
*Older url was http://localhost:8080/swagger-ui.html*

Below screenshot shows the Home Page of Swagger UI. It shows a list of all the resources that are exposed.



Choosing the Student resource takes you to details of the resource. It shows all the request methods that can be used with a Resource. 

You can also see the details for a Specific Request Method. 

You can use the ‘Try it out’ button to execute a request and see the response. 

### Customizing Swagger Documentation with Annotations

You can add notes on the resource method to add more documentation

@GetMapping("/students/{id}")

@ApiOperation(value = "Find student by id",

notes = "Also returns a link to retrieve all students with rel - all-students")

public Resource<Student> retrieveStudent(@PathVariable long id) {

Also supported is enhancing the documentation on the Request and Response Beans.

@Entity

@ApiModel(description="All details about the student. ")

public class Student {

@ApiModelProperty(notes="Name should have atleast 2 characters")

@Size(min=2, message="Name should have atleast 2 characters")

private String name;

## Versioning

### There are four popular approaches to versioning a REST API:

### • URI versioning

### • URI parameter versioning

### • Accept header versioning

### • Custom header versioning

### **URI Versioning**

### In this approach, version information becomes part of the URI. For example, http://api.example.org/v1/users

### and http://api.example.org/v2/users represent two different versions of an application API. Here we

### use v notation to denote versioning and the numbers 1 and 2 following the v indicate the first and second

### API versions.

### URI versioning has been one of the most commonly used approaches and is used by major public APIs

### such as Twitter, LinkedIn, Yahoo, and SalesForce. Here are some examples:

### • LinkedIn: https://api.linkedin.com/v1/people/~

### • Yahoo: https://social.yahooapis.com/v1/user/12345/profile

### • SalesForce: http://na1.salesforce.com/services/data/v26.0

### • Twitter: https://api.twitter.com/1.1/statuses/user\_timeline.json

### • Twilio: https://api.twilio.com/2010-04-01/Accounts/{AccountSid}/Calls

### As you can see, LinkedIn, Yahoo, and SalesForce use the v notation. In addition to a major version,

### SalesForce uses a minor version as part of its URI version. Twilio, by contrast, takes a unique approach and

### uses a timestamp in the URI to differentiate its versions.

### Making a version part of the URI is very appealing as the version information is right in the URI. It also simplifies API development and testing. Folks can easily browse and use different versions of REST services via a Web browser. On the contrary, this might make client’s life difficult. For example, consider a client

### storing references to user resources in its database. On switching to a new version, these references get

### outdated and the client has to do a mass database update to upgrade references to new version.

### **URI Parameter Versioning**

### This is similar to the URI versioning that we just looked at except that the version information is specified

### as a URI request parameter. For example, the URI http://api.example.org/users?v=2 uses the version

### parameter v to represent the second version of the API. The version parameter is typically optional and a

### default version of the API will continue working for requests without version parameter. Most often, the

### default version is the latest version of the API.

### Although as not popular as other versioning strategies, a few major public APIs such as Netf lix have

### used this strategy. The URI parameter versioning shares the same disadvantages of URI versioning. Another

### disadvantage is that some proxies don’t cache resources with a URI parameter, resulting in additional

### network traffic.

### **Accept Header Versioning**

### This versioning approach uses the Accept header to communicate version information. Because the header

### contains version information, there will be only one URI for multiple versions of API.

### Up to this point, we have used standard media types such as "application/json" as part of the Accept

### header to indicate the type of content the client expects. To pass additional version information, we need a

### custom media type. The following convention is popular when creating a custom media type:

### vnd.product\_name.version+ suffix

### The vnd is the starting point of the custom media type and indicates vendor. The product or producer

### name is the name of the product and distinguishes this media type from other custom product media types.

### The version part is represented using strings such as v1 or v2 or v3. Finally, the suffix is used to specify the structure of the media type. For example, the +json suffix indicates a structure that follows the guidelines

### established for media type "application/json". RFC 6389 (https://tools.ietf.org/html/rfc6839) gives

### a full list of standardized prefixes such as +xml, +json, and +zip. Using this approach, a client, for example,

### can send an application/vnd.quickpoll.v2+json accept header to request the second version of the API.

### The Accept header versioning approach is becoming more and more popular as it allows fine-grained

### versioning of individual resources without impacting the entire API. This approach can make browser

### testing harder as we have to carefully craft the Accept header. GitHub is a popular public API that uses this

### Accept header strategy. For requests that don’t contain any Accept header, GitHub uses the latest version

### of the API to fulfill the request.

### **Custom Header Versioning**

### The custom header versioning approach is similar to the Accept header versioning approach except that,

### instead of the Accept header, a custom header is used. Microsoft Azure takes this approach and uses the

### custom header x-ms-version. For example, to get the latest version of Azure at the time of writing this book,

### your request needs to include a custom header:

### x-ms-version: 2014-02-14

### This approach shares the same pros and cons as that of the Accept header approach. Because the

### HTTP specification provides a standard way of accomplishing this via the Accept header, the custom header

### approach hasn’t been widely adopted.

How to implement versioning:

1. Either separate the complete code base
2. Have multiple methods corresponding to version in same class
3. Separate the code, which is different and keep the other code same, this can be very tricky.

### **Factors affecting Versioning Choice**

### Following factors affect the choice of versioning:

### URI Pollution - URL versions and Request Param versioning pollute the URI space.

### Misuse of HTTP Headers - Accept Header is not designed to be used for versioning.

### Caching - If you use Header based versioning, we cannot cache just based on the URL. You would need take the specific header into consideration.

### Can we execute the request on the browser? - If you have non technical consumers, then the URL based version would be easier to use as they can be executed directly on the browser.

### API Documentation - How do you get your documentation generation to understand that two different urls are versions of the same service?

## How to implement versioning in swagger?

## Customizing swagger

## Paging and Sorting

### 1. Resource vs Representation

### Before we start designing our pagination API, we need to have a clear understanding of page as a resource or a representation of the resource. There are numbers of fundamentals we need to keep in mind

### 

### The page is not a resource in REST but its a property of the request.

### Let’s take an example of building pagination for a resource name Product, on a high level we do have following three options to build the pagination.

### Take product as a resource and use query strings to handle pagination along with other parameters like sorting etc. (e.g. http://domainname/products?page=1).

### The second option is to use the page as a resource and query string for sorting. (e.g. http://domainname/products/page/1?sort\_by=date).

### use page as resource and URL part for sorting. (e.g. http://domainname/products/date/page/1)

### With above question in mind, let’s try to answer few of the question which will be helpful while designing pagination for the REST API.

### Do you see the page as a resource of the products within the page as the resource?

### Keep in mind that REST API is not built around any predefined rules or specifications, all the three above options are valid and based on the answer to the above question. If we think the page as a resource, option three is a valid choice (query to page 1 will give page 1 and 2 give page 2), but if we say that products on the page are the resource than option 3 is no longer valid (product might change in the future on page 1,2 etc.).Personally, I will choose option 1 since for me page is not a resource, it’s a property of the request.

### Treating the page itself as a resource introduces a host of problems such as no longer being able to uniquely identify resources between calls. This, coupled with the fact that, in the persistence layer, the page is not a proper entity but a holder that is constructed when necessary, makes the choice straightforward: **the page is part of the representation**.

### The next question in the pagination design in the context of REST is **where to include the paging information**:

### in the URI path: */foo/page/1*

### the URI query: */foo?page=1*

### Keeping in mind that **a page is not a Resource**, encoding the page information in the URI is no longer an option.

### We're going to use the standard way of solving this problem by **encoding the paging information in a URI query.**

### 

### 

### 2. Discoverability

### Discoverability helps to make RESTful API more useful and elegant. Making the REST API discoverable is often overlooked. Here is a high-level summary of the REST API discoverability.

### With this feature, REST API providing full URI’s in the responses to the client means that no client will ever need to “compose” a URI.

### Client API become independent of the URI structure.

### With above 2 points, API is more flexible and allow the developer to change URI schema without breaking API. (Remember, API provides all URI, they are not created dynamically by the client API).

### Discoverability is closely related to HATEOAS in the REST API. REST API pagination discoverability will pass on "next","previous","first" and "last" link as part of the response data. We are coving how to add this feature to your API during pagination.

### 

### **3. Pagination Design Considerations**

### Let’s quickly cover some of the main points while building your REST API pagination interface.

### 3.1. Limit

### Limit allow API and client to control the number of results requested in the resultset. By passing the limit parameter, you can specify how many items you want each page to return.API can configure default limit but should allow the client to specify a limit.

### http://hostname/products?page=1&limit=50

### In the above request, the client is setting the limit as 50. Be careful while allowing the customer to set limit parameter as a very high number of limit can degrade API performance. REcommendation to have a maximum allowed limit during the API design.

### 3.2 Page Number Pagination

### In this pagination style, the clients specify a page number containing the data they need. For example, a client

### wanting all the blog posts in page 3 of our hypothetical blog service, can use the following GET method:

### http://blog.example.com/posts?page=3

### The REST service in this scenario would respond with a set of posts. The number of posts returned

### depends on the default page size set in the service. It is possible for the client to override the default page

### size by passing in a page-size parameter:

### <http://blog.example.com/posts?page=3&size=20>

### Limit Offset Pagination

### In this pagination style, the clients uses two parameters: a limit and an offset to retrieve the data that they

### need. The limit parameter indicates the maximum number of elements to return and the offset parameter

### indicates the starting point for the return data. For example, to retrieve 10 blog posts starting from the item

### number 31, a client can use the following request:

### <http://blog.example.com/posts?limit=10&offset=30>

### Cursor-Based Pagination

### In this pagination style, the clients make use of a *pointer or a cursor* to navigate through the data set.

### A cursor is a service-generated random character string that acts as a marker for an item in the data set.

### To understand this style, consider a client making the following request to get blog posts:

### http://blog.example.com/posts

### On receiving the request, the service would send data similar to this:

### {

### "data" : [

### ... Blog data

### ],

### "cursors" : {

### "prev" : null,

### "next" : "123asdf456iamcur"

### }

### }

### This response contains a set of blogs representing a subset of the total dataset. The cursors that are part

### of the response contains a prev field that can be used to retrieve the previous subset of the data. However,

### because this is the initial subset, the prev field value is empty. The client can use the cursor value in the next

### field to get the next subset of the data using the following request:

### http://api.example.com/posts?cursor=123asdf456iamcur

### On receiving this request, the service would send the data along with the prev and next cursor fields.

### This pagination style is used by applications such as Twitter and Facebook that deal with real-time datasets

### (tweets and posts) where data changes frequently. The generated cursors typically don’t live forever and

### should be used for short-term pagination purposes only

### Time-Based Pagination

### In this style of pagination, the client specifies a timeframe to retrieve the data in which they are interested.

### Facebook supports this pagination style and requires time specified as a Unix timestamp. These are two

### Facebook example requests:

### https://graph.facebook.com/me/feed?limit=25&until=1364587774

### https://graph.facebook.com/me/feed?limit=25&since=1364849754

### Both examples use the limit parameter to indicate the maximum number of items to be returned.

### The until parameter specifies the end of the time range, whereas the since parameter specifies the beginning

### of the time range.

## Sorting

### Sorting allows REST clients to determine the order in which items in a dataset are arranged. REST services

### supporting sorting allow clients to submit parameters with properties to be used for sorting. For example, a

### client can submit the following request to sort blog posts based on their created date and title:

### http://blog.example.com/posts?sort=createdDate,title

### Sort Ascending or Sort Descending

### The REST services can also allow the clients specify one of the two sort directions: ascending or descending.

### Because there is no set standard around this, the following examples showcase popular ways for specifying

### sort direction:

### http://blog.example.com/posts?sortByDesc=createdDate&sortByAsc=title

### http://blog.example.com/posts?sort=createdDate,desc&sort=title,asc

### http://blog.example.com/posts?sort=-createdDate,title

### In all of these examples, we are retrieving blog posts in the descending order of their created date.

### Posts with the same created date are then sorted based on their titles.

### • In the first approach the sort parameter clearly specifies if the direction should be

### ascending or descending.

### • In the second approach, we have used the same parameter name for both directions.

### However, the parameter value spells out the sort direction.

### • The last approach uses the “-” notation to indicate that any property prefixed with a

### “-” should be sorted on a descending direction. Properties that are not prefixed with

### a “-” will be sorted in the ascending direction.

## REST java client

## Testing

## Security