# References:

Balaji Varanasi book

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

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

# 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.

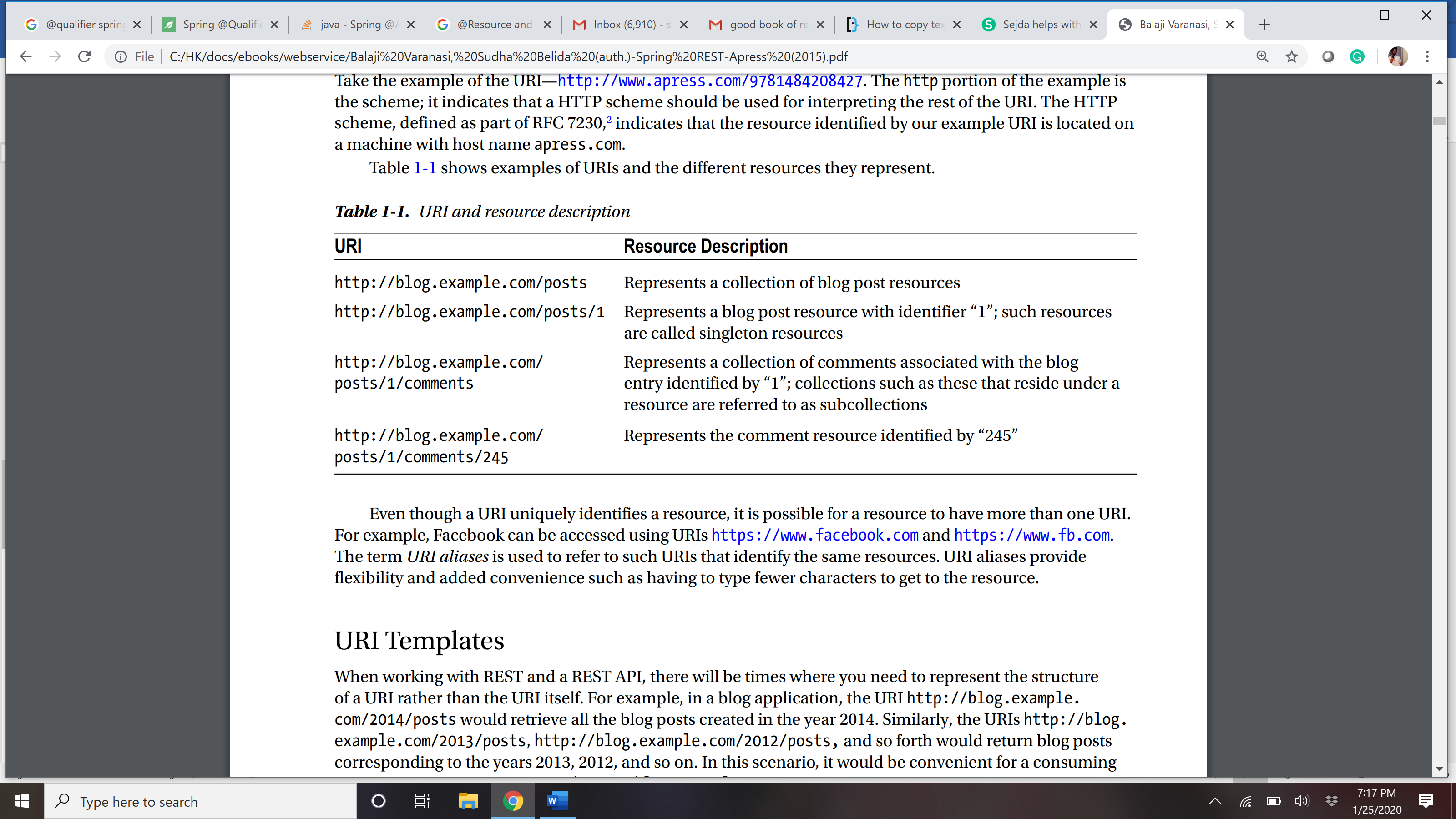
representation are snapshop at a given resource at a point of time. , the same resource can have several representations. These representations can range from text-based HTML, XML, and JSON formats to binary formats such as PDFs, JPEGs, and MP4s. It is possible for the client to request a particular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

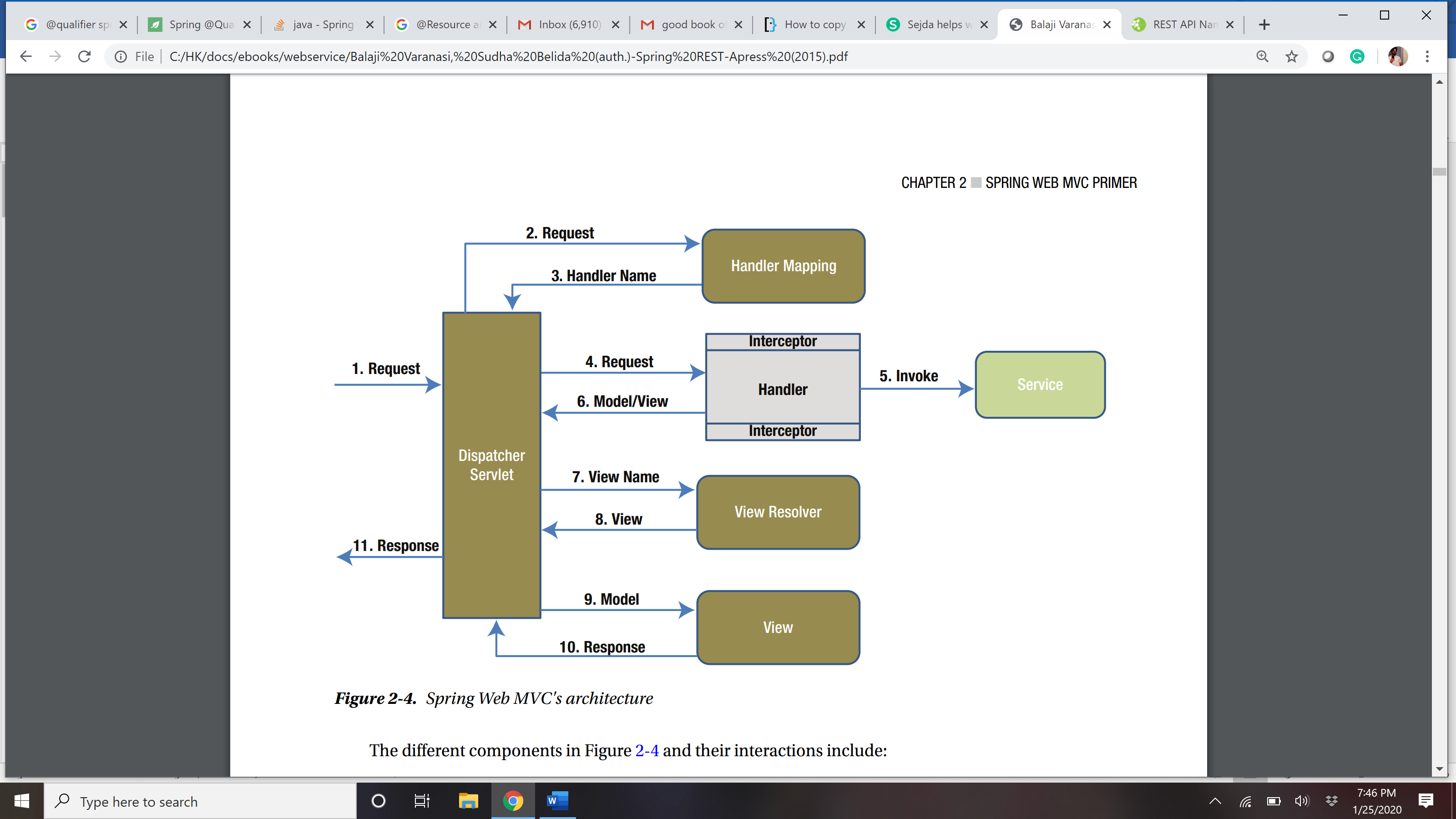
Axis

Jersy

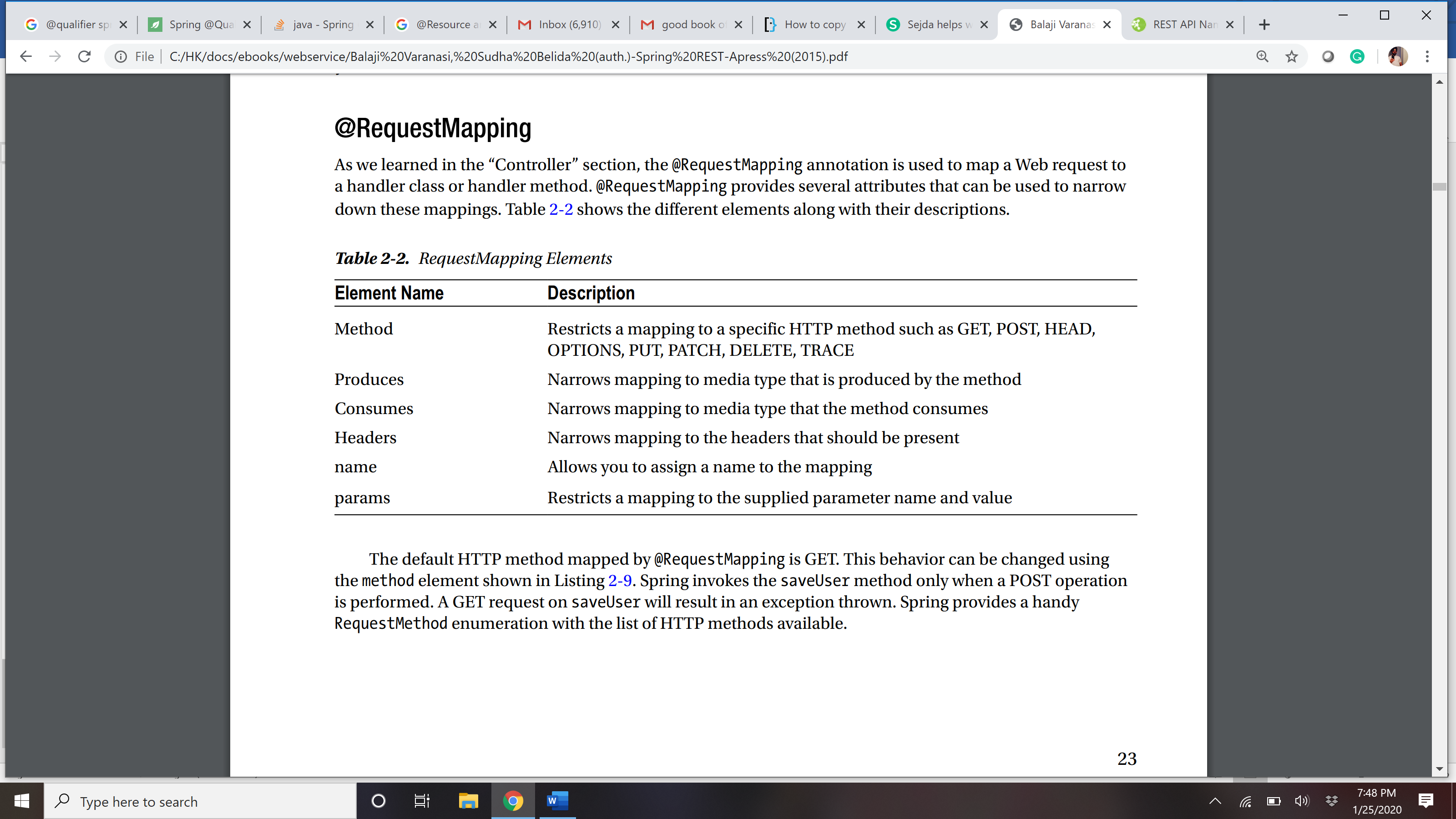
Spring

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 .