

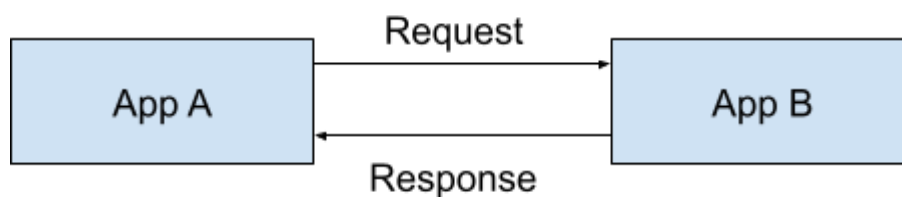
## Rest API and Spring Boot

---

### Introduction to Web Services

Web services are a way to interact with different software systems over the Internet. They allow us to load and send data between systems, enabling integration and communication between different software applications. Web services use standardized protocols and interfaces like HTTP and XML to exchange information and perform tasks. This makes it possible for different software systems to communicate and exchange data, regardless of the specific technologies or platforms they are built on. Web services are an important part of modern software development and are widely used in various applications, including e-commerce, data integration, and business automation.

### Data exchange between applications



Assume there is an application A, which needs some data from another application B. Application A will send a request to application B with the appropriate requirement using some standardized protocol. Application B would process the request and send the response back to application A. The application can then use this data according to its needs.

There are 2 most popular ways to exchange data between applications - XML and JSON.

Let's assume that we want to share the following details of a Student -

Student [ ID - 123, Name - Tester ]

### XML

XML is the abbreviation for Extensible Markup Language. In XML, we would format the data as follows -

```
<Student>
  <ID> 123 </ID>
  <Name> Tester </Name>
</Student>
```

## JSON

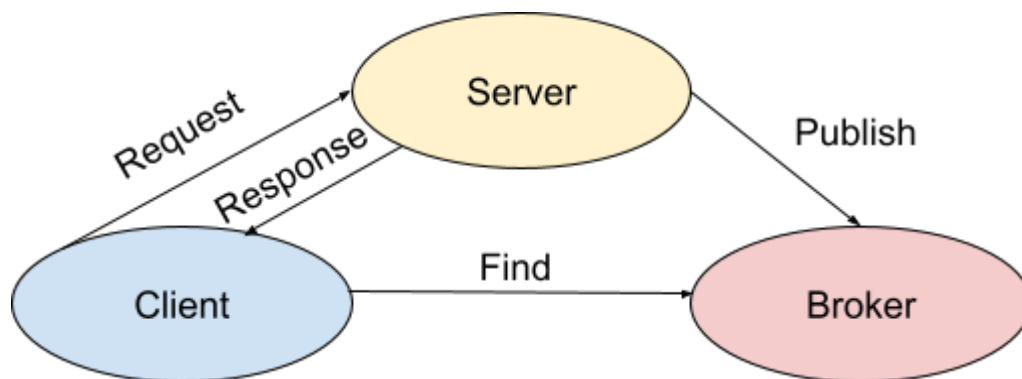
Javascript Object Notation, also known as JSON, is the most popular data exchange format. In JSON, we would format the data as follows -

```
{ "Student" :  
  [ { "ID" : 123,  
      "Name" : "Tester"  
    } ]  
}
```

## Key Terminologies

- **Request and Response** - The input which comes to the web service is known as a Request, and the output given by the web service is known as a Response.
- **Message exchange format** - It is the format in which data exchange takes place. There are two most popular exchange formats - XML and JSON.
- **Service Definition** - It ensures that the process of data transfer is standardized.
- **Service Consumer or Client** - The application which sends the request is known as the service consumer or client.
- **Service Provider or Server** - The application which processes the request and gives back the Response is known as Service Provider or Server.
- **Transport** - It is responsible for the data exchange between the service provider and consumer. The data transfer can take place via HTTP or Queues.

## Web Services Architecture



1. As soon as the server is available, it will publish a service description to the Broker.
2. Then the Service consumer or client can retrieve the service description from the Broker.
3. Finally, the client can send the Request to the Server and the Server can send the response.

**All web services are APIs, but not all APIs are web services.**

## Introduction to RESTful Web Services

RESTful web services are loosely coupled, lightweight web services that are particularly well suited for creating APIs for clients spread out across the internet.

## RESTful Key Elements

- **Resources** - In RESTful web services, a resource is a specific piece of information or data that can be accessed and manipulated using standard HTTP methods (such as GET, POST, PUT, and DELETE). The information or data is typically represented in a format such as JSON or XML and is identified by a unique URI (Uniform Resource Identifier).
- **Request Headers** - The header is an additional piece of information provided with the request. It contains authorization data or the type of response required by the client.
- **Request Body** - The data which is sent by the client in the request to be processed by the server is known as Request Body.
- **Response Body** - If the client asks for some piece of information, the data is sent in the response body by the server.
- **Response Status Code** - The server makes use of various status codes to notify if the request was successful or if not, then the type of error encountered.
- **Request Methods** - Request methods tell about the type of request to the server.



- GET method is used to fetch data from the server.
- POST method is used to store new data on the server.
- PUT method is used to update existing data in the server.
- DELETE method is used to delete the data which exists on the server.

## SOAP vs REST web services

SOAP Protocol	RESTful Web Services
SOAP is a protocol.	REST is an architectural approach.
The data exchange format is always XML.	There is no strict data exchange format.
SOAP cannot use RESTful services because it is a protocol.	RESTful services can use SOAP web services because it is an architectural approach that can use any protocol like HTTP or SOAP.

Cloud-based architecture work on the REST principle, it makes more sense for web services to be programmed on the REST services based.

# RESTful Web Services with Spring Boot

---

## REST API - Important Annotations

- **@RestController** - It is a Spring annotation used to create a RESTful web service using Spring Boot. It is used to create web services that return JSON or XML data instead of HTML views. When a request is made to a URL that is mapped to a method annotated with **@RestController**, the method is invoked and the return value is automatically converted to JSON or XML and sent back to the client as the HTTP response.
- **@RequestMapping** - **@RequestMapping** is a Spring annotation that is used to map a web request to a specific method in a controller class.
  - It is used to handle requests to a specific URL and can be applied at the class level or the method level.
  - The **@RequestMapping** annotation can also be used to specify the HTTP method (such as GET, POST, PUT, DELETE) that the mapped method should handle, and can be used to map requests to specific HTTP headers or request parameters.
- **@GetMapping** - **@GetMapping** is a Spring MVC annotation that is used to handle HTTP GET requests. It maps a specific URI (Uniform Resource Identifier) pattern to a controller method, allowing it to handle the incoming request and return a response.

## Creation of Hello World REST API

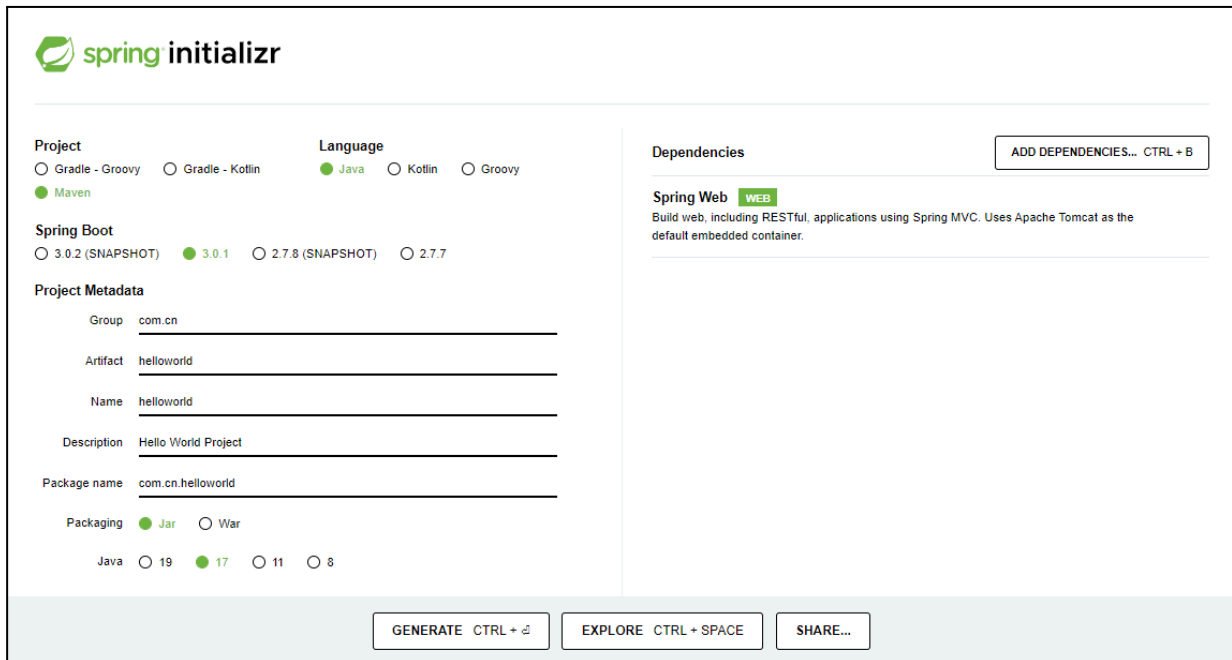
We would be creating a Maven project for the Hello World REST API. Thus, it is required that we have the following tools and technologies for our project.

- Spring Boot version 2.2.2+
- Java 8+
- Maven 3.0+
- An IDE of your choice (The tutorial uses Eclipse IDE)
- Dependency - Spring Web

## @RestController, @RequestMapping and @GetMapping

1. Visit <https://start.spring.io> (This is a web-based Spring Initializer).

Add Spring Web Dependency to your project.



The screenshot shows the Spring Initializr web interface. The 'Project' section has 'Maven' selected. The 'Language' section has 'Java' selected. The 'Spring Boot' section has '3.0.1' selected. The 'Project Metadata' section has fields for Group (com.cn), Artifact (helloworld), Name (helloworld), Description (Hello World Project), and Package name (com.cn.helloworld). The 'Packaging' section has 'Jar' selected. The 'Java' section has '17' selected. The 'Dependencies' section has 'Spring Web' selected. At the bottom, there are buttons for 'GENERATE', 'EXPLORE', and 'SHARE...'. The 'GENERATE' button has a tooltip that says 'CTRL + G'.

In the end, the selections should look like this, and then click on **GENERATE** button on the bottom. It would download a zip file.

2. Extract the zip in a folder.
3. Import the project in Eclipse by going to -  
File -> Open projects from File System -> Directory -> Select the extracted folder -> Finish.
4. Create a new Controller class in the project as follows.

```
@RestController
@RequestMapping("/project")
public class HelloWorldController {
    @RequestMapping("/hello")
    public String hello()
    {
        return "Hello Coding Ninjas !";
    }
}
```

```
}

@GetMapping("/hello-world")
public String helloWorld()
{
    return "Hello from Earth !";
}
}
```

5. Run the project and go to your browser and visit the URL - <http://localhost:8080/project/hello-world> to see the output.
6. In this example, we made use of `@RestController` on the class which helps us to return JSON from the methods, `@RequestMapping` to set a base URL for the class, and then `@GetMapping` to handle get requests on a URL.

## @PathVariable

Now, let's add another method to our controller class, which can grab a value from the URL dynamically. This can be done with the help of `@PathVariable` annotation in the method parameters.

```
@RestController
@RequestMapping("/project")
public class HelloWorldController {

    @RequestMapping("/hello")
    public String hello()
    {
        return "Hello Coding Ninjas !";
    }

    @GetMapping("/hello-world")
    public String helloWorld()
    {
        return "Hello from Earth !";
    }

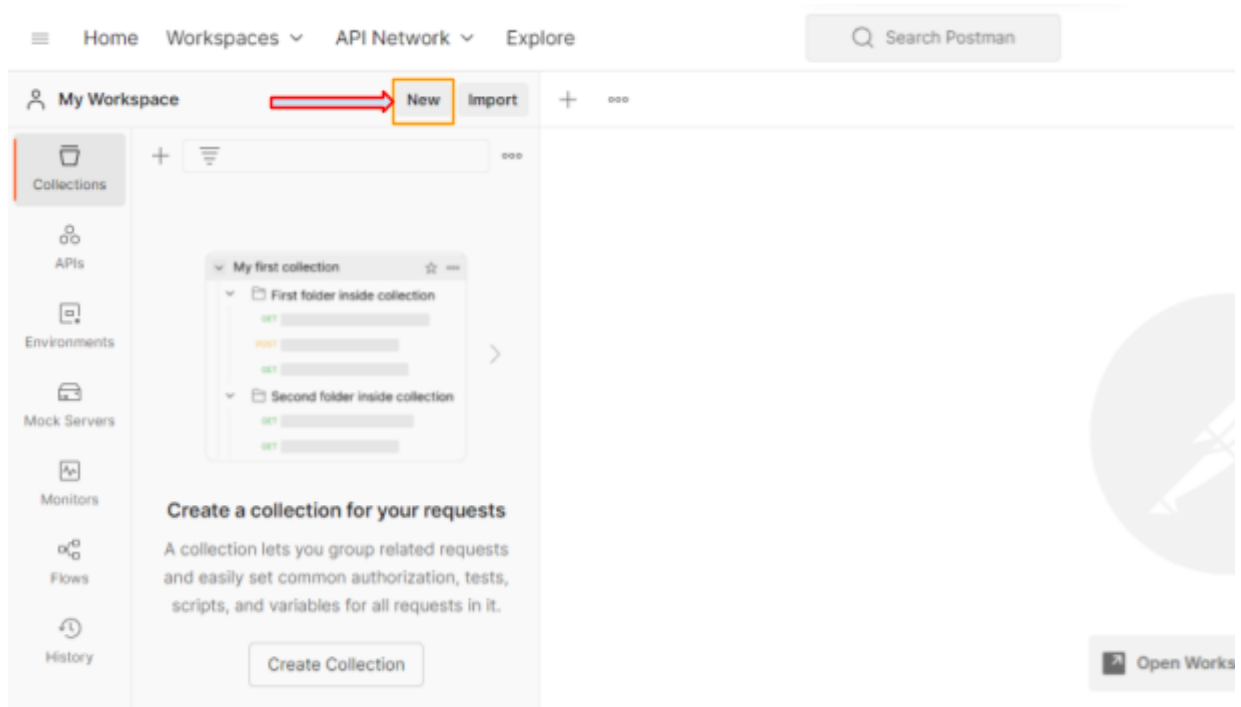
    @GetMapping("/hello-mars/{name}")
    public String helloFromMars(@PathVariable String name)
    {
        return "Hello from Mars "+name;
    }
}
```

Now, the controller class would look something like this. Here `@PathVariable` takes the `{name}` from the URL and attaches it to the 'name' parameter of the 'helloFromMars' method. This parameter can then be used in the method body normally.

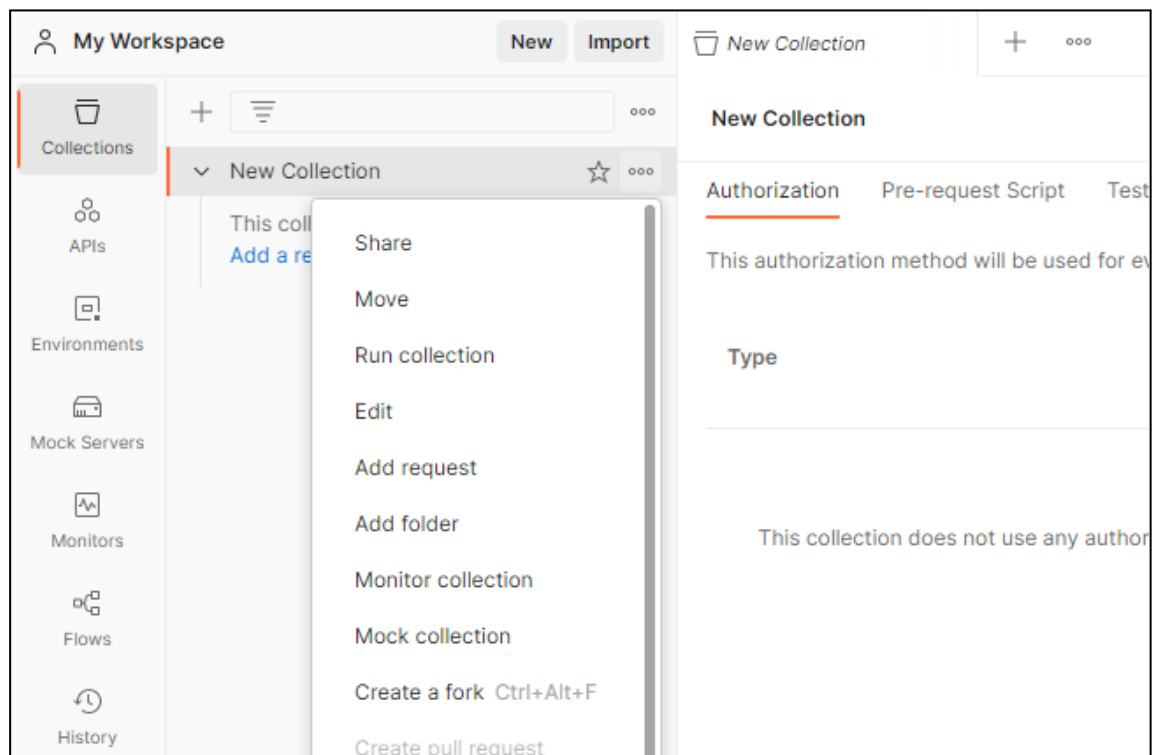
Now visit the URL <http://localhost:8080/hello-mars/Elon> , you would see an output that says - “Hello from Mars Elon”.

## Post Man Setup

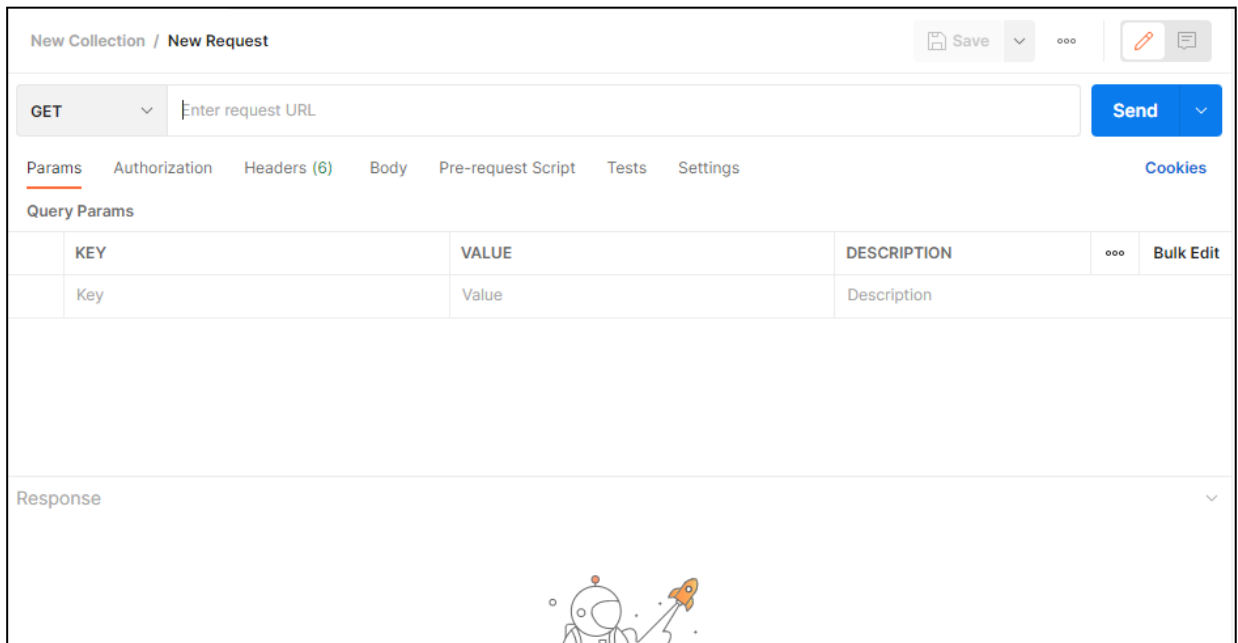
1. Visit <https://www.postman.com/downloads/> and download the PostMan app.
2. Install the Postman app and open it. You would see a screen like this.



3. Click on New, and select “Collection” from the screen.
4. Right-click on “New Collection” and select ‘New Request’ from the menu.



- Now you have added a request, and you are ready to test APIs.



- You can enter the URL to be tested, and also select from a range of request methods such as GET, POST, PUT, DELETE etc and see the response in the bottom section.

## Implementing the POST method

- Create a new Project from the web [spring.initialzr](https://spring.io/guides-topics/docs/spring-initializr/).
- Create a Hotel class as follows

```
public class Hotel {
    private String id;
    private String name;
    private long rating;
    private String city;

    public Hotel(String id, String name, long rating, String city) {
        super();
        this.id = id;
        this.name = name;
        this.rating = rating;
        this.city = city;
    }

    public String getId() {
        return id;
    }

    public void setId(String id) {
        this.id = id;
    }

    public String getName() {
```



```
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public long getRating() {
        return rating;
    }
    public void setRating(long rating) {
        this.rating = rating;
    }
    public String getCity() {
        return city;
    }
    public void setCity(String city) {
        this.city = city;
    }
}
```

3. Create a HotelService class which will contain business logic, and methods to add, get or delete Hotels, which will be called by Controller class.

We will make use of ArrayList of Hotel to store the hotels, and also map the Hotels to their ID's in a HashMap, so that they can be fetched easily later on.

```
@Service
public class HotelService {

    List<Hotel> hotelList = new ArrayList<>();
    Map<String,Hotel> hotelMap= new HashMap<>();

    public void createHotel(Hotel hotel) {
        hotelList.add(hotel);
        hotelMap.put(hotel.getId(), hotel);
    }
}
```

4. Create a new Controller class and annotate it as @RestController.

```
@RestController
@RequestMapping("/hotel")
public class HotelController {
    @Autowired
    HotelService hotelService;

    @PostMapping("/create")
    public void createHotel(@RequestBody Hotel hotel)
```

```
    {  
        hotelService.createHotel(hotel);  
    }  
}
```

## Implementing the GET method

In the Hotel project's service class, let's add a service method which will return the list of all Hotels, and a service method which returns a Hotel by its ID.

So, in the Service class add two methods as follows:

```
@Service  
public class HotelService {  
  
    List<Hotel> hotelList = new ArrayList<>();  
    Map<String,Hotel> hotelMap= new HashMap<>();  
  
    public Hotel getHotelById(String id) {  
        Hotel hotel = hotelMap.get(id);  
        return hotel;  
    }  
  
    public List<Hotel> getAllHotels() {  
        return hotelList;  
    }  
  
    public void createHotel(Hotel hotel) {  
        hotelList.add(hotel);  
        hotelMap.put(hotel.getId(), hotel);  
    }  
}
```

Also add respective controller methods in the Controller class with the help of `@PathVariable` to get the HotelId from the URL.

```
@RestController  
@RequestMapping("/hotel")  
public class HotelController {  
    @Autowired  
    HotelService hotelService;  
  
    @GetMapping("/id/{id}")  
    public Hotel getHotelById(@PathVariable String id)  
    {  
        return hotelService.getHotelById(id);  
    }  
  
    @GetMapping("/getAll")
```

```
public List<Hotel> getAllHotels()
{
    return hotelService.getAllHotels();
}

@PostMapping("/create")
public void createHotel(@RequestBody Hotel hotel)
{
    hotelService.createHotel(hotel);
}
}
```

## Implementing the DELETE Method

Now let's add a method to delete the Hotel by a given ID, so add a method in the service class as follows:

```
@Service
public class HotelService {

    List<Hotel> hotelList = new ArrayList<>();
    Map<String,Hotel> hotelMap= new HashMap<>();

    public void deleteHotelById(String id) {
        Hotel hotel = getHotelById(id);
        hotelList.remove(hotel);
        hotelMap.remove(id);
    }

    public Hotel getHotelById(String id) {
        Hotel hotel = hotelMap.get(id);
        return hotel;
    }

    public List<Hotel> getAllHotels() {
        return hotelList;
    }

    public void createHotel(Hotel hotel) {
        hotelList.add(hotel);
        hotelMap.put(hotel.getId(), hotel);
    }
}
```

And similarly in the Controller class, make use of `@DeleteMapping` and `@PathVariable` to map a URL to delete a Hotel by Id as follows :

```
@RestController
@RequestMapping("/hotel")
public class HotelController {
    @Autowired
    HotelService hotelService;

    @DeleteMapping("/remove/id/{id}")
    public void deleteHotelById(@PathVariable String id)
    {
        hotelService.deleteHotelById(id);
    }

    @GetMapping("/id/{id}")
    public Hotel getHotelById(@PathVariable String id)
    {
        return hotelService.getHotelById(id);
    }

    @GetMapping("/getAll")
    public List<Hotel> getAllHotels()
    {
        return hotelService.getAllHotels();
    }

    @PostMapping("/create")
    public void createHotel(@RequestBody Hotel hotel)
    {
        hotelService.createHotel(hotel);
    }
}
```

## Implementing the PUT Method

Let's add a method in the service class which takes a new Hotel object and replaces the old Hotel object with the same ID.

So, add a new update method in the existing service class as follows:

```
@Service
public class HotelService {

    List<Hotel> hotelList = new ArrayList<>();
    Map<String,Hotel> hotelMap= new HashMap<>();

    public void updateHotel(Hotel updatedHotel) {
```

```
//1. Get the previous data of the hotel
//2. remove this old data from list
//3. Add the updated data to the list.

Hotel existingHotel= getHotelById(updatedHotel.getId());
hotelList.remove(existingHotel);
hotelList.add(updatedHotel);

//4. update the previous data with new data.
//5. Update the map with new data.

hotelMap.put(updatedHotel.getId(), updatedHotel);
}

public void deleteHotelById(String id) {
    Hotel hotel = getHotelById(id);
    hotelList.remove(hotel);
    hotelMap.remove(id);
}

    public Hotel getHotelById(String id) {
        Hotel hotel = hotelMap.get(id);
        return hotel;
    }

    public List<Hotel> getAllHotels() {
        return hotelList;
    }

    public void createHotel(Hotel hotel) {
        hotelList.add(hotel);
        hotelMap.put(hotel.getId(), hotel);
    }
}
```

Accordingly add a update method in the controller class with the help of `@PutMapping` as follows :

```
@RestController
@RequestMapping("/hotel")
public class HotelController {
    @Autowired
    HotelService hotelService;

    @PutMapping("/update")
    public void updateHotel(@RequestBody Hotel hotel)
    {
```

```
        hotelService.updateHotel(hotel);
    }

    @DeleteMapping("/remove/id/{id}")
    public void deleteHotelById(@PathVariable String id)
    {
        hotelService.deleteHotelById(id);
    }

    @GetMapping("/id/{id}")
    public Hotel getHotelById(@PathVariable String id)
    {
        return hotelService.getHotelById(id);
    }

    @GetMapping("/getAll")
    public List<Hotel> getAllHotels()
    {
        return hotelService.getAllHotels();
    }

    @PostMapping("/create")
    public void createHotel(@RequestBody Hotel hotel)
    {
        hotelService.createHotel(hotel);
    }
}
```

## Exception Handling

Let's say in our Hotel Application, we have a list of Hotels with ID's 1,2,3,4 and 5 respectively. And a user sends a get request with an ID of 7 which does not exist in the list. So, we should be able to handle such requests and also respond with proper error messages in such cases. This is known as exception handling, and there are several HTTP response status codes which indicate whether a given request was successful or not.

These are some of the most common response codes -

- **404** - This indicates that the requested resource was not found on the server.
- **400** - It is used to indicate that the server did not process the request due to an error in the request sent by the user.
- **500** - This status code indicates that the server encountered an internal error and could not process the request.

In Spring Boot we can annotate an exception class with `@ResponseStatus`, so if that exception is thrown, then a response status is sent to the client.

## Implementing Exception Handling

We will start by creating a new class which extends the `RuntimeException` class and also annotate it with `@ResponseStatus`. In the `@ResponseStatus`, pass the `HttpStatus.NOT_FOUND`, as the argument. There are many options which are available to us, such as `HttpStatus.OK`, `HttpStatus.BAD_REQUEST` etc. Then add a constructor which takes a `String` as an input and passes it to the super class constructor.

So, your new Class would look like this -

```
@ResponseStatus(HttpStatus.NOT_FOUND)
public class HotelNotFoundException extends RuntimeException{

    public HotelNotFoundException(String message) {
        super(message);
    }
}
```

Our custom exception class is ready, now we can throw this exception in our service method. So, in the `getHotelById` method, add a condition which will check if a Hotel exists by the given id. We can do this with the help of `isEmpty` method of `ObjectUtils` class. It is a static method which takes an object as argument and returns true if it is null. The `getHotelById` method should look like this now:

```
public Hotel getHotelById(String id) {
    if(ObjectUtils.isEmpty(hotelMap.get(id)))
    {
        throw new HotelNotFoundException("Hotel not found for
id: "+id);
    }
    Hotel hotel = hotelMap.get(id);
    return hotel;
}
```

## Validations for RESTful Services

Validating the user input is a common requirement, and thus Java provides with several annotations to help constraint the Entity according to various criterias.

Some of the most common annotations are:

- `@Size` validates that the property has a size or length as specified by the min and max attributes. It can be applied to Collections, Map, Strings, and arrays.
- `@Min` is used to validate that the value of field should not be lesser than that given in annotation attribute.
- `@Max` validates that the annotated property has a value not greater than the value attribute.

We will make use of these annotations to restrict the Hotel field values as follows:

```
private String id;

@Size(min=3)
private String name;

@Min(value = 1)
@Max(value = 10)
private long rating;
private String city;
```

After these restrictions, the 'name' will have a minimum length of 3, rating will have a minimum value of 1 and maximum value of 10.

These annotations only provide a restrictions on the fields, but the validation is taken care by another annotation which is `@Valid`. Let's have a look at the implementation of `@Valid`.

## @Valid

`@Valid` can be is used against a method parameter in the controller method which will accept the input from the user. So, in our hotel project we will use the `@Valid` in our `createHotel` controller method. So, after using `@Valid`, our controller method would look like this -

```
@PostMapping("/create")
public void createHotel(@Valid @RequestBody Hotel hotel)
{
    hotelService.createHotel(hotel);
}
```

## BindingResult Class

`@Valid` will validate our inputs, but we do need to throw an exception if the input is not valid. So we make use of `BindingResult` class to store the result or the errors that may have occurred. This can be passed as a parameter to the controller method. Then we can make use of `hasErrors` method of `BindingResult` class to detect if any error has occurred, while validating the input.

So after passing `BindingResult` class as a parameter and adding a if condition to check errors and throw exception, our `createHotel` method would be as follows:

```
@PostMapping("/create")
public void createHotel(@Valid @RequestBody Hotel hotel,
BindingResult
bindingResult)
{
    if(bindingResult.hasErrors())
    {
        throw new RuntimeException("Request Not Valid");
    }
}
```



```
        hotelService.createHotel(hotel);  
    }
```

## Instructor Codes

- [Hotel Application](#)