SPRING MVC

Spring is widely used for creating scalable applications. For web applications Spring provides Spring MVC framework which is a widely used module of spring which is used to create scalable web applications. Spring MVC framework enables the separation of modules namely Model View, Controller, and seamlessly handles the application integration. This enables the developer to create complex applications also using plain java classes. The model object can be passed between view and controller using maps.

The Spring MVC framework consists of the following components:

- **Model –**
  A model can be an object or collection of objects which basically contains the data of the application.
- **View –**
  A view is used for displaying the information to the user in a specific format. Spring supports various technologies like **freemarker**, **velocity**, and **thymeleaf**.
- **Controller –**
  It contains the logical part of the application.
  @Controller annotation is used to mark that class as a controller.

- **Front Controller –**
  It remains responsible for managing the flow of the web application. Dispatcher Servlet acts as a front controller in Spring MVC.

Difference between Spring MVC and Spring Boot:

| S.No.	SPRING MVC	SPRING BOOT
1.	Spring MVC is a Model View, and Controller based web framework widely used to develop web applications.	Spring Boot is built on top of the conventional spring framework, widely used to develop REST APIs.
2.	If we are using Spring MVC, we need to build the configuration manually.	If we are using Spring Boot, there is no need to build the configuration manually.
3.	Spring MVC specifies each dependency separately.	It wraps the dependencies together in a single unit.
4.	It takes more time in development.	It reduces development time and increases productivity.

Spring Boot Architecture:

Spring Boot is an advanced version or project of the [Spring framework](#). Along with the Spring framework, it also consists of third-party libraries and Embedded HTTP servers. It easily creates a production-grade, less time-consuming, and stand-alone applications based on the Spring framework.

The aim of Spring Boot is to completely remove the use of XML-based and annotation-based configuration in the applications. Using [Spring Boot](#), we can also create an application with minimal fuss (less time and effort). By default, it offers most of the things, such as functions, procedures, etc.

In this tutorial, we are going to learn about the architecture of the Spring Boot framework. It follows the layered architecture and consists of four layers, as shown below.

- Presentation Layer
- Business Layer
- Persistence Layer
- Database Layer
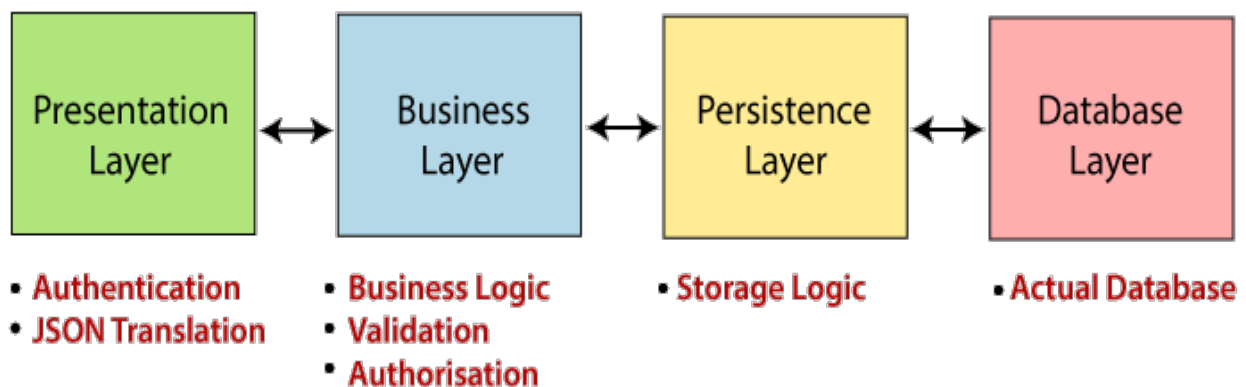
The above diagram shows that each layer of the architecture is directly communicating with the layer just above or below it, is because of the workflow. It means each layer only depends on its adjacent layer, so if we change the API of one layer, we just need to update the layers adjacent to it.

The brief description of the layers is given below.

1. Presentation layer

It is the front layer or top layer of the architecture, as it consists of *views*. It is used to translate the JSON fields to objects and vice-versa, and also handles authentication and HTTP requests. After completing the authentication, it passes it to the business layer for further processes.

2. Business Layer

It handles all the business logic and also performs validation and authorization as it is a part of business logic. For example, only admins are allowed to modify the user's account.

3. Persistence Layer

It contains all the storage logic, such as database queries of the application. It also translates the business objects from and to database rows.

4. Database Layer

The database layer consists of the database such as MySQL, Postgre, MongoDB, etc. It may contain multiple databases. All the database related operations like CRUD (Create, Read/Retrieve, Update, and Delete) are performed in this layer.

The implementation of the above layered architecture is performed in such a way:
The HTTP request or web requests are handled by the Controllers in the *presentation layer*, the *services* control the business logic, and the repositories handle persistence (storage logic). A controller can handle multiple services, a service can handle multiple repositories, and a repository can handle multiple databases.

Spring Boot Work Flow

The **Spring Boot architecture** is based on the Spring framework. So, it mostly uses all the features and modules of Spring-like Spring MVC, Spring Core, etc., except that there is no need for the DAO and DAOImpl classes.
The following diagram shows the workflow of Spring Boot.
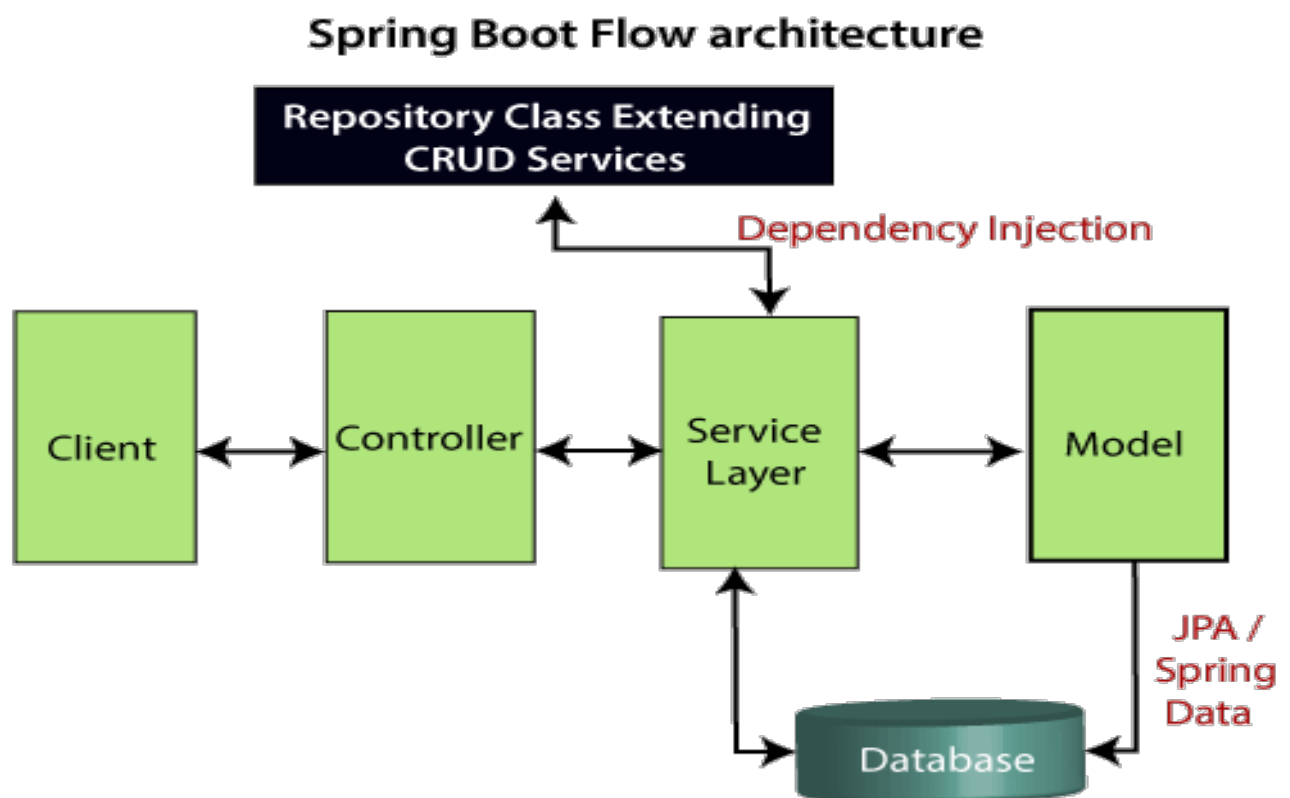
- The client makes an HTTP request (GET or POST).
- The request is forwarded to the controller, and it maps the request and processes it. It also calls the service logic if needed.
- The business logic is performed in the service layer, and the logic is performed on the data from the database that is mapped with the model or entity class through JPA.
- A JSP page is returned as a response to the client if no error has occurred.

Annotation Basics
Spring Boot Specific Annotations
@SpringBootApplication (@Configuration + @ComponentScan + @EnableAutoConfiguration)

Everyone who worked on Spring Boot must have used this annotation. When we create a Spring Boot Starter project, we receive this annotation as a gift. This annotation applies at the main class which has main () method. The Main class serves two purposes in a Spring Boot application: *configuration* and *bootstrapping*. In fact *@SpringBootApplication* is a combination of three annotations with their default values. They are *@Configuration,*
@ComponentScan, and *@EnableAutoConfiguration.* Therefore, we can also say that **@SpringBootApplication** is a 3-in-1 annotation.

@EnableAutoConfiguration: enables the auto-configuration feature of Spring Boot.
@ComponentScan: enables @Component scan on the package to discover and register components as beans in Spring's application Context.
@Configuration: allows to register extra beans in the context or imports additional configuration classes.

We can also use above three annotations in place of @SpringBootApplication if we want any customized behavior of them.

@EnableAutoConfiguration

@EnableAutoConfiguration enables auto-configuration of beans present in the classpath in Spring Boot applications. In a nutshell, this annotation enables Spring Boot to auto-configure the application context. Therefore, it automatically creates and registers beans that are part of the included jar file in the classpath and also the beans defined by us in the application. For example, while creating a Spring Boot starter project when we select Spring Web and Spring Security dependency in our classpath, Spring Boot auto-configures Tomcat, Spring MVC and Spring Security for us.

Moreover, Spring Boot considers the package of the class declaring the *@EnableAutoConfiguration* as the default package. Therefore, if we apply this annotation in the root package of the application, every sub-packages & classes will be scanned. As a result, we won't need to explicitly declare the package names using *@ComponentScan*.

Furthermore, *@EnableAutoConfiguration* provides us two attributes to manually exclude classes from auto-configurations. If we don't want some classes to be auto-configured, we can use exclude attribute to disable them. Another attribute is excludeName to declare a fully qualified list of classes to exclude. For example, below are the codes.

Use of 'exclude' in *@EnableAutoConfiguration*

@Configuration

```
@EnableAutoConfiguration(exclude={WebSocketMessagingAutoConfiguration.class})
public class MyWebSocketApplication {
    public static void main(String[] args) {
        ...
    }
}
```

Use of 'excludeName' in @EnableAutoConfiguration

```
@Configuration
@EnableAutoConfiguration(excludeName =
{"org.springframework.boot.autoconfigure.websocket.servlet.WebSocketMessagingAutoCon
figuration"})
public class MyWebSocketApplication {
    public static void main(String[] args) {
        ...
    }
}
```

@ConfigurationProperties

Spring Framework provides various ways to inject values from the properties file. One of them is by using @*Value* annotation. Another one is by using @*ConfigurationProperties* on a configuration bean to inject properties values to a bean. But what is the difference among both ways and what are the benefits of using @*ConfigurationProperties*, you will understand it at the end. Now Let's see how to use @*ConfigurationProperties* annotation to inject properties values from the application.properties or any other properties file of your own choice.

First, let's define some properties in our application.properties file as follows. Let's assume that we are defining some properties of our development working environment. Therefore, representing properties name with prefix 'dev'.

```
dev.name=Development Application
dev.port=8090
dev.dburl=mongodb://mongodb.example.com:27017/
dev.dbname=employeeDB
dev.dbuser=admin
dev.dbpassword=admin
```

Now, create a bean class with getter and setter methods and annotate it with @ConfigurationProperties.

```
@ConfigurationProperties(prefix="dev")
public class MyDevAppProperties {
    private String name;
    private int port;
    private String dburl;
    private String dbname;
    private String dbuser;
    private String dbpassword;

    //getter and setter methods
}
```

Here, Spring will automatically bind any property defined in our property file that has the prefix '*dev*' and the same name as one of the fields in the MyDevAppProperties class.

Next, register the *@ConfigurationProperties* bean in your *@Configuration* class using the *@EnableConfigurationProperties* annotation.

```
@Configuration
@EnableConfigurationProperties(MyDevAppProperties.class)
public class MySpringBootDevApp { }
```

Finally create a Test Runner to test the values of properties as below.

```
@Component
public class DevPropertiesTest implements CommandLineRunner {

    @Autowired
    private MyDevAppProperties devProperties;

    @Override
    public void run(String... args) throws Exception {
        System.out.println("App Name = " + devProperties.getName());
        System.out.println("DB Url = " + devProperties.getDburl());
        System.out.println("DB User = " + devProperties.getDbuser());
    }
}
```

We can also use the *@ConfigurationProperties* annotation on *@Bean*-annotated methods.

@Configuration

We apply this annotation on classes. When we apply this to a class, that class will act as a configuration by itself. Generally the class annotated with *@Configuration* has bean definitions as an alternative to <bean/> tag of an XML configuration. It also represents a configuration using Java class. Moreover the class will have methods to instantiate and configure the dependencies. For example :

```
@Configuration
public class AppConfig {
    @Bean
    public RestTemplate getRestTemplate() {
        RestTemplate restTemplate = new RestTemplate();
        return restTemplate();
    }
}
```

♥ The benefit of creating an object via this method is that you will have only one instance of it. You don't need to create the object multiple times when required. Now you can call it anywhere in your code.

@Bean

We use *@Bean* at method level. If you remember the xml configuration of a Spring, It is a direct analog of the XML <bean/> element. It creates Spring beans and generally

used with *@Configuration*. As aforementioned, a class with *@Configuration (we* can call it as a Configuration class) will have methods to instantiate objects and configure dependencies. Such methods will have *@Bean* annotation. By default, the bean name will be the same as the method name. It instantiates and returns the actual bean. The annotated method produces a bean managed by the Spring IoC container.

```
@Configuration
public class AppConfig {
    @Bean
     public Employee employee() {
       return new Employee();
    }
    @Bean
    public Address address() {
      return new Address();
    }
}
```

For comparison sake, the configuration above is exactly equivalent to the following Spring XML:

```
<beans>
    <bean name="employee" class="com.dev.Employee"/>
    <bean name="address" class="com.dev.Address"/>
</beans>
```

The annotation supports most of the attributes offered by <bean/>, such as: init-method, destroy-method, autowiring, lazy-init, dependency-check, depends-on and scope.

@Component

This is a generic stereotype annotation which indicates that the class is a Spring-managed bean/component. *@Component* is a class level annotation. Other stereotypes are a specialization of *@Component*. During the component scanning, Spring Framework automatically discovers the classes annotated with *@Component*, It registers them into the Application Context as a Spring Bean.

Applying *@Component* annotation on a class means that we are marking the class to work as Spring-managed bean/component. For example, look at the code below:

```
@Component
class MyBean { }
```

On writing a class like above, Spring will create a bean instance with name 'myBean'. Please keep in mind that, By default, the bean instances of this class have the same name as the class name with a lowercase initial. However, we can explicitly specify a different name using the optional argument of this annotation like below.

```
@Component("myTestBean")
class MyBean { }
```

@Controller

@Controller tells Spring Framework that the class annotated with *@Controller* will work as a controller in the Spring MVC project.

@RestController

@RestController tells Spring Framework that the class annotated with *@RestController* will work as a controller in a Spring REST project.

@Service

@Service tells Spring Framework that the class annotated with *@Service* is a part of service layer and it will include business logics of the application.

@Repository

@Repository tells Spring Framework that the class annotated with @Repos*itory* is a part of data access layer and it will include logics of accessing data from the database in the application.

Configuration Annotations

Next annotations in our article 'Spring Boot Annotations with Examples' are for Configurations. Since Spring Framework is healthy in configurations, we can't avoid learning annotations on configurations. No doubt, they save us from complex coding effort.

@ComponentScan

Spring container detects Spring managed components with the help of *@ComponentScan*. Once you use this annotation, you tell the Spring container where to look for Spring components. When a Spring application starts, Spring container needs the information to locate and register all the Spring components with the application context. However It can auto scan all classes annotated with the stereotype annotations such as *@Component, @Controller, @Service*, and *@Repository* from pre-defined project packages.

The *@ComponentScan* annotation is used with the *@Configuration* annotation to ask Spring the packages to scan for annotated components. *@ComponentScan* is also used to specify base packages and base package classes using basePackages or basePackageClasses attributes of *@ComponentScan*. For example :

```
import com.springframework.javatechonline.example.package2.Bean1;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
@Configuration
@ComponentScan(basePackages =
{"com.springframework.javatechonline.example.package1",
                "com.springframework.javatechonline.example.package3",
                "com.springframework.javatechonline.example.package4"},
        basePackageClasses = Bean1.class
            )
public class SpringApplicationComponentScanExample {
    ....
}
```

Here the *@ComponentScan* annotation uses the basePackages attribute to specify three packages including their subpackages that will be scanned by the Spring

container. Moreover the annotation also uses the basePackageClasses attribute to declare the Bean1 class, whose package Spring Boot will scan.

Moreover, In a Spring Boot project, we typically apply the @SpringBootApplication annotation on the main application class. Internally, *@SpringBootApplication* is a combination of the *@Configuration,*

@ComponentScan, and *@EnableAutoConfiguration* annotations. Further, with this default setting, Spring Boot will auto scan for components in the current package (containing the SpringBoot main class) and its sub packages.

@Import

Suppose we have multiple Java configuration classes annotated by *@Configuration*. *@Import* imports one or more Java configuration classes. Moreover It has the capability to group multiple configuration classes. We use this annotation where one *@Configuration* class logically imports the bean definitions defined by another. For example:

```
@Configuration
@Import({ DataSourceConfig.class, TransactionConfig.class })
public class AppConfig extends ConfigurationSupport {
    // @Bean methods here that can reference @Bean methods in DataSourceConfig or TransactionConfig
}
```

List of Essential Spring Boot Annotations:

There are many annotations you can use to control and define your applications. Here are some of the most useful, sorted by category.
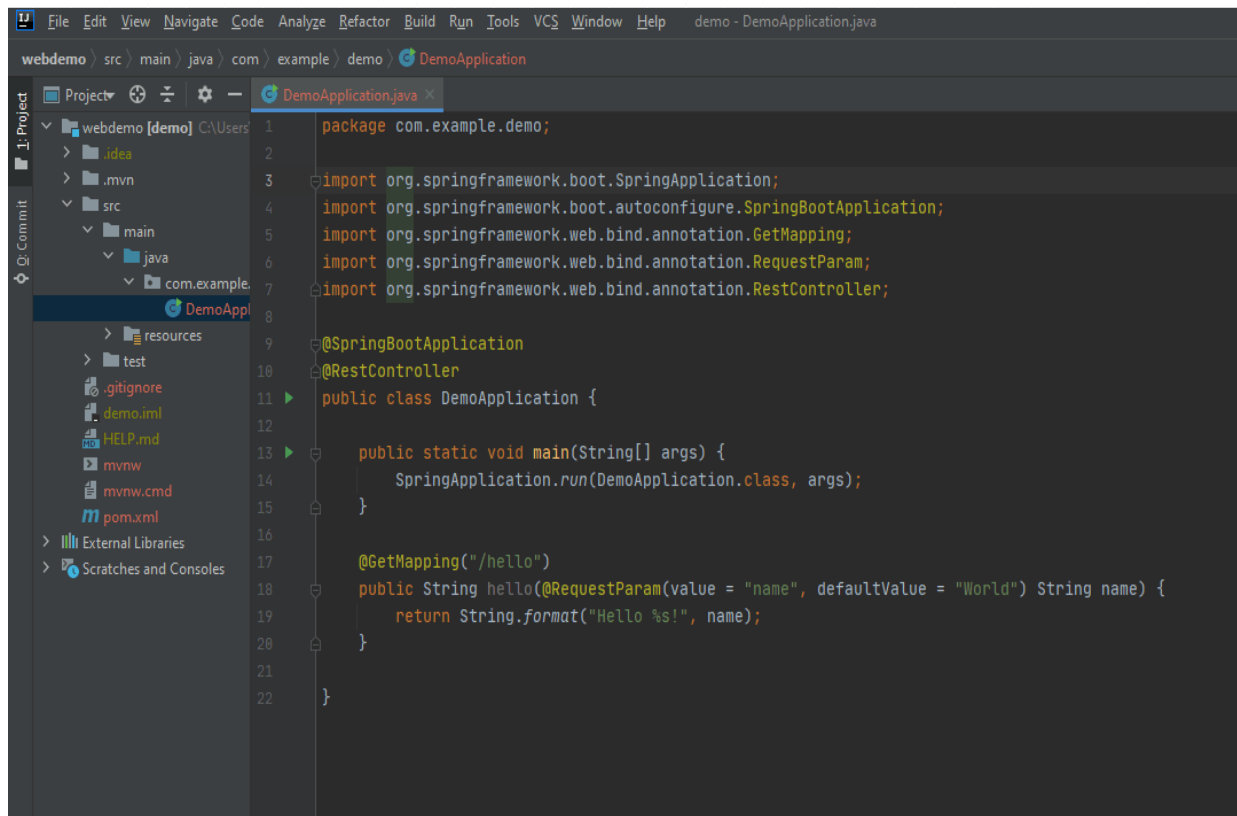
1. Basic Setup

```
@SpringBootApplication
```
Java

Copy

Available since version 1.2, @SpringBootApplication replaces several other key annotations and as such, is essential to nearly all Spring Boot applications.

The 1.2 version delivers the same functionality as the near ubiquitous @Configuration, @ComponentScan, and @EnableAutoConfiguration.

This is a good place for a simple `Hello World`, based on Spring Boot's demo application, so here goes:

```java
package com.example.demo;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class DemoApplication {

    public static void main(String[] args) {
        SpringApplication.run(DemoApplication.class, args);
    }

}
```

It's not much of a `Hello World` though, as it doesn't actually display anything. For that, let's introduce @RestController and @GetMapping. But before that, let's talk about the annotations that @SpringBootApplication replaces.

@Configuration

Now superseded by @SpringBootApplication, @Configuration [enables Java configuration](#) and lets you use [Spring Beans](#) in the class.

@ComponentScan

Also superseded by @SpringBootApplication, @ComponentScan enables component scanning and means controller classes and components you create can be discovered by the framework. It marks classes to be discovered with @Controller.

If you include the @ComponentScan annotation, then all application components will be registered as Spring Beans automatically. That includes @Service, @Component, @Repository, @Controller, and others.

@EnableAutoConfiguration

The final annotation replaced by @SpringBootApplication, @EnableAutoConfiguration enables Spring Boot's autoconfiguration. Spring Boot makes over 200 decisions for you. These can be overridden if you want to make your own choices, but it will pick sensible defaults for you, [saving you a lot of time](#) at the beginning of projects.

The exclude attribute is used to disable autoconfiguration for specific classes.

2. Request Responses

The next few annotations show how easy it is to respond to HTTP requests using Spring. There are several ways to mark classes and functions to control how they behave and what they return.

@GetMapping

Take a look at your earlier `Hello World` and get it returning a response.

In this example, again from Spring's demo code, the @GetMapping class combines with the @RestController to deliver a response to calls like `http://yoururl/hello`.

Here it's a simple `Hello World` by default, though it also shows how to take a parameter, which can replace the default *World* in the response. `http://yoururl/?name=Dave` will return `Hello Dave`, for example.

```
package com.example.demo;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;

@SpringBootApplication
@RestController
```

```
public class DemoApplication {


public static void main(String[] args) {

SpringApplication.run(DemoApplication.class, args);

}


@GetMapping("/hello")

public String hello(@RequestParam(value = "name",
defaultValue = "World") String name) {

return String.format("Hello %s!", name);

}

}

@RequestMapping
```

This can combine with @Controller to create a class that returns simple requests, such as the following code:

```
@Controller

@RequestMapping("users")

public class UserController {


    @GetMapping("/{id}", produces =
"application/json")

    public @ResponseBody User getUser(@PathVariable
int id) {

        return findUsersById(id);

    }


    private User findUsersById(int id) {

            // return user specific data

    }

@RequestParam
```

As seen in the code above, @RequestParam allows you to send parameters in the get request and use them in Java. It also supplies a default value.

3. Component Types

There are several annotations to let you label components in your application. Aside from @RestController, these are functionally identical, but allow you to organize your application and mark classes for specific roles, helping to keep your application [modular](#).

@Component

Application components and their variants are automatically registered as Spring Beans, providing dependency injection, provided you use either @SpringBootApplication or @ComponentScan.

@Repositry, @Controller, and @Service are move specific alternatives to @Componenet.

@Service

This is an alternative to @Component that specifies you intend to use the class as part of your service layer. However, it doesn't actually implement anything differently than @Component.

@Repository

This annotation marks a class as part of your data layer, for [handling storage, retrieval, and search](#). This can be especially useful for targeting your tests and generating the [right exceptions](#).

@Controller

@Controller is a specialized @Component marked as a controller in [MVC architecture](#).

@RestController

@RestController combines the @Controller and @ResponseBody into a single annotation. @RestController classes return domains instead of views.

4. Testing

As it should, Spring Boot makes it [very easy to write tests](#). [Identifying and fixing errors](#) is much easier when the framework helps you.

@SpringBootTest

Though Spring Boot can use regular Spring tests using the @Test annotation, it has a special annotation—@SpringBootTest—that lets you test using Spring Boot specific features.

@SpringBootTest works best when testing [the whole application together](#). There are other options, such as @WebMvcTest and @DataJpaTest to use, if you're looking at those specific areas.

You can mark a test class as follows:

```
@SpringBootTest(properties = "spring.main.web-application-type=reactive")
class YourTests {
    // code here
}
@MockBean
```

The @MockBean annotation allows you to create a temporary version of a service for [testing](#). It's useful if you have a [web service](#) you connect to that isn't suitable for testing, or if you want to test against specific results.

Here's an example, based on [Spring Boot's documentation](#).

```java
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.*;
import org.springframework.boot.test.context.*;
import org.springframework.boot.test.mock.mockito.*;

import static org.assertj.core.api.Assertions.*;
import static org.mockito.BDDMockito.*;

@SpringBootTest
class ServiceTests {

    @MockBean
    private RemoteService remoteService;

    @Autowired
    private Capitalizer capitalizer;

    @Test
    void exampleTest() {
        // RemoteService has been injected into the capitalizer bean

 given(this.remoteService.testCall()).willReturn("test");
        String caps = capitalizer.capitalizeTestCall();
        assertThat(caps).isEqualTo("TEST");
    }


}
```

To validate input for methods, you apply the @Validated annotation to the class. For nested properties, apply the @Valid tag.

Here's an example of using the @Validated tag along with javax.validation constraints.

```java
import javax.validation.constraints.Size;
import javax.validation.constraints.NotNull;
import org.springframework.stereotype.Service;
import org.springframework.validation.annotation.Validated;

@Service
@Validated
public class TestBean {
public Archive findByCopiesAndTitle(@Size(min = 1, max = 100) String code, @NotNull Title title) {
return …
}
}
```

Jackson ObjectMapper Tutorial

In this tutorial, you will learn how to use Jackson ObjectMapper with Spring Boot application to serialize and deserialize Java objects.

Overview

In today's time, the JSON format is one of the most popular formats to transfer and exchange data on the world wide web. Almost all RESTful web services support JSON as input and output structure. Additionally, It is also widely used by various NoSQL databases such as MongoDB to store records as well.

In this tutorial, we'll deep-dive into Jackson ObjectMapper and discuss how to serialize JSON requests and responses in the Spring Boot application with various Jackson configurations.

Preparation:

In this example, I'll be using the following tools:

- JDK 1.8
- Intellij Idea for IDE
- Maven
- Springboot Application version 2.5.5
- Lombok

Structure of the post :

The development outline of this blog will be as follow:

1. Project Setup
2. Introducing Jackson ObjectMapper

Step 1: Project setup

At first, we will create a simple Spring Boot based application. You can use the [Spring Initializr page](#) to create an initial project template.

After importing the project into IDE, we will be creating the following sub-packages inside our main package '*com.appsdeveloperblog.ObjectMapperTutorial*' :

- **Config:** This folder contains configuration class with filename '*config.java*'

- **Controller:** This folder contains class *'EmployeeController'* with filename *'EmployeeController.java'*
- **Model:** This folder contains Employee DTO structure with filename *'Employee.java'*

In the end, the final project structure will look like this:
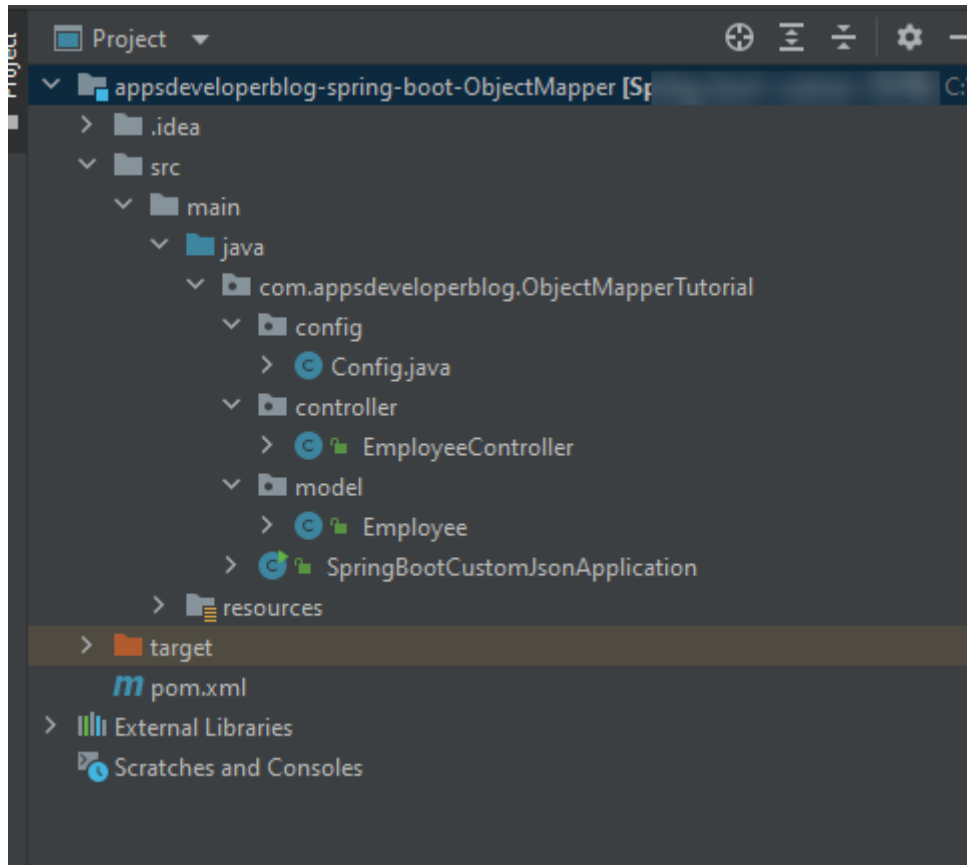
In the following sections, we will learn the components of *config*, *controller* classes in detail.

Maven Dependencies

We will only include *spring-boot-starter-web* and *Lombok* dependencies in our pom.xml. As, Spring Boot already provides managed dependencies for various Jackson modules. For example,

- jackson-core
- jackson-databind
- jackson-annotations

Hence, we don't need to include this in pom.xml explicitly.

```
<?xml version="1.0" encoding="UTF-8"?>
```

```xml
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<parent>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-parent</artifactId>

<version>2.5.0</version>

<relativePath/>

</parent>

<groupId>com.appsdeveloperblog.ObjectMapperTutorial</groupId>

<artifactId>Spring-boot-custom-JSON</artifactId>

<version>0.0.1-SNAPSHOT</version>

<name>Jackson ObjectMapper Tutorial</name>

<description>Jackson ObjectMapper Tutorial</description>

<properties>

<java.version>11</java.version>

<maven.compiler.source>11</maven.compiler.source>

<maven.compiler.target>11</maven.compiler.target>

</properties>

<dependencies>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

<dependency>

<groupId>org.projectlombok</groupId>

<artifactId>lombok</artifactId>

<version>1.18.20</version>

</dependency>

</dependencies>

<build>

<plugins>
```

```xml
<plugin>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-maven-plugin</artifactId>

</plugin>

</plugins>

</build>

</project>
```

Create Employee POJO

Now, we create our simple POJO class which we will be using in our project. Here we are using the "Employee" object with basic member fields like ID, FirstName etc.

Here is the code for 'Employee.java'

```java
package com.appsdeveloperblog.ObjectMapperTutorial.model;

import lombok.AllArgsConstructor;

import lombok.Getter;

import lombok.Setter;

import java.io.Serializable;

import java.util.ArrayList;

import java.util.List;

@AllArgsConstructor

@NoArgsConstructor

@Getter

@Setter

public class Employee {

private int id;

private String firstName;

private String jobTitle;

private String gender;

public static java.util.List<Employee> getEmployee() {

List<Employee> employees = new ArrayList<>();

employees.add(new Employee(1, "damian", "Developer", "Male"));
```

```java
employees.add(new Employee(2, "noel", "Developer", "Male"));

employees.add(new Employee(3, "amena", "CEO", "Female"));

employees.add(new Employee(4, "eugenia", "Developer", "Male"));

employees.add(new Employee(5, "mckee", "Developer", "Female"));

employees.add(new Employee(6, "ayers", "Developer", "Female"));

return employees;

}

}
```

Create Controller

Next, we need the controller for GET and POST services.
The complete source code for *"EmployeeController.java"* is as follow:

```java
package com.appsdeveloperblog.ObjectMapperTutorial.controller;

import com.appsdeveloperblog.ObjectMapperTutorial.model.Employee;

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

import java.util.List;

@RestController

public class EmployeeController {

@RequestMapping("/getEmployees")

public List<Employee> getEmployees() {

return Employee.getEmployee();

}

@PostMapping("/createEmployees")

@ResponseBody

public Employee createProduct(@RequestBody Employee employee) {

/* sample logic to show employee object enhancement*/

employee.setID(employee.getID() + 100);

employee.setJobTitle("software " + employee.getJobTitle());

return employee;

}

}
```

Running the Application

Now, we can create the main class to start the *Spring Boot Application*:

```
package com.appsdeveloperblog.ObjectMapperTutorial;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication

public class SpringBootCustomJsonApplication {

public static void main(String[] args) {

SpringApplication.run(SpringBootCustomJsonApplication.class, args);

}

}
```

In the end, we will run our application either via executing the following command or using the 'Run' button from IDE. You will find that the application started on Tomcat port 8080.
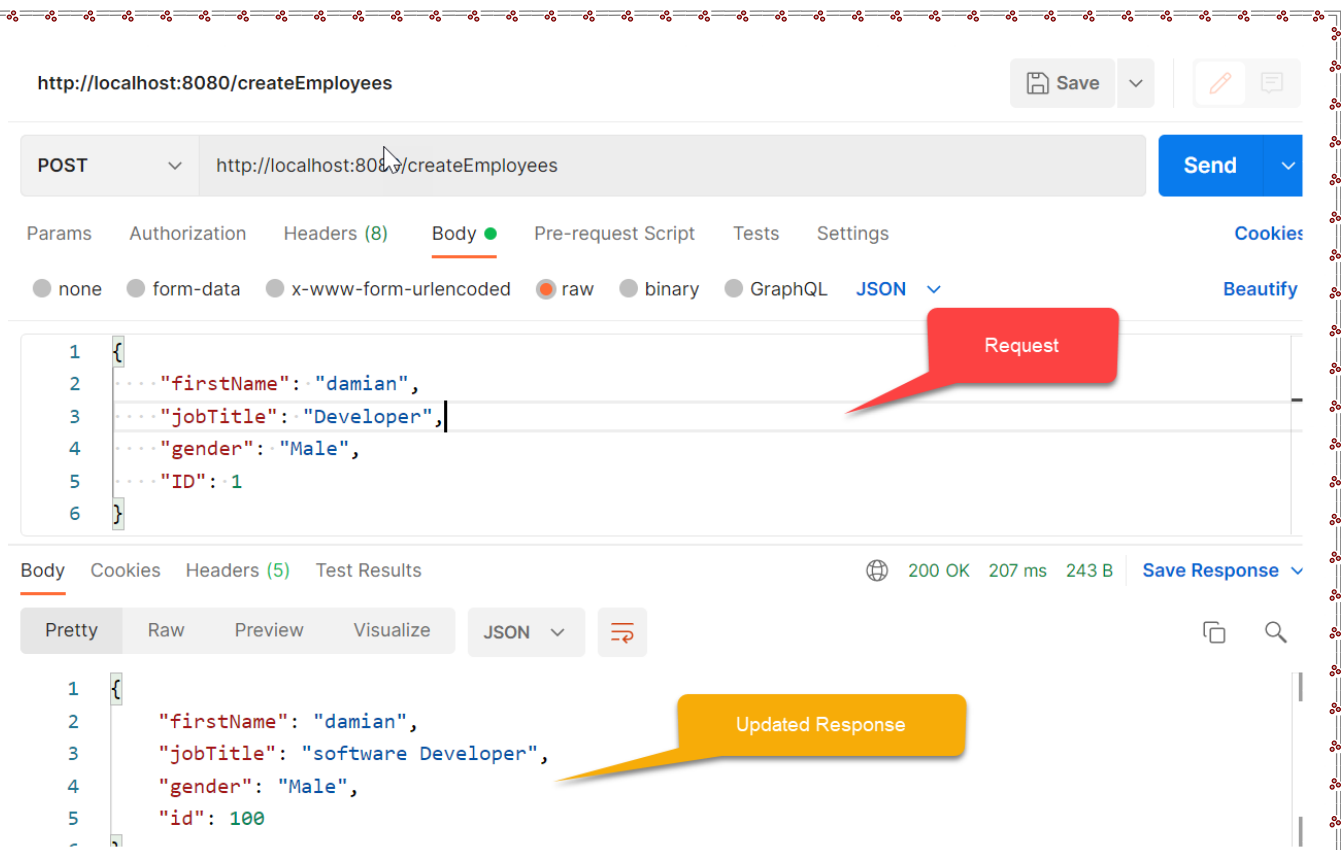
To test GET we can simply execute the below URL:

http://localhost:8080/getEmployees

Subsequently, we will get the following response:

```
[{"firstName":"damian","jobTitle":"Developer","gender":"Male","id":1},
{"firstName":"noel","jobTitle":"Developer","gender":"Male","id":2},{"firstName":"amena","jobTitle":"CEO","gender":"Female",   ":3},
{"firstName":"eugenia","jobTitle":"Developer","gender":"Male","id":4},
{"firstName":"mckee","jobTitle":"Developer","gender":"Female","id":5},
{"firstName":"ayers","jobTitle":"Developer","gender":"Female","id":6}]
```

Similarly, if we do POST call using the below URL via Postman we should get the following response.

http://localhost:8080/createEmployees

Here is a sample request and response.

```
http://localhost:8080/createEmployees                    [🖫 Save ▾]  [✎ ▭]  </>

POST  ▾   http://localhost:8080/createEmployees                    [Send ▾]

Params  Authorization  Headers (8)  Body ●  Pre-request Script  Tests  Settings    Cookies

○ none  ○ form-data  ○ x-www-form-urlencoded  ● raw  ○ binary  ○ GraphQL  JSON ▾    Beautify

1  {
2      "firstName": "damian",                         [Request]
3      "jobTitle": "Developer",
4      "gender": "Male",
5      "ID": 1
6  }

Body  Cookies  Headers (5)  Test Results        🌐 200 OK  207 ms  243 B  Save Response ▾

Pretty  Raw  Preview  Visualize   JSON ▾   ⇥

1  {
2      "firstName": "damian",          [Updated Response]
3      "jobTitle": "software Developer",
4      "gender": "Male",
5      "id": 100
```

Now, we will see how to work with the Jackson object mapper in our project to convert a DTO object to a JSON and vice-versa.

Step2: Introducing Jackson ObjectMapper

In the previous example, we already saw a basic version of ObjectMapper functionality even though we have not created an instance of it. Let see how it has happened sequentially as following:

During HTTP GET request:

1. We requested a GET call via "/getEmployees" in the browser.
2. Controller returned a list of Employee DTO objects.
3. Spring internally created ObjectMapper instance and converted the employee objects to JSON.

This process is called Serialization because spring converted the list of employee object to JSON.

During HTTP POST request:

1. We send a POST request via "/createEmployees" from Postman using JSON request payload.

2. In Controller, Spring automatically converted the requested JSON into Employee DTO object.
3. Employee DTO was modified as per logic and returned to the controller.
4. Spring again converted returned received "Employee" object DTO to JSON and presented to the client as JSON response.

This process of converting JSON payload to java object is called Deserialization.

Customizing ObjectMapper

In this section, we'll learn how to customize the default *ObjectMapper* behaviour so based on requirement we can change the Serialization/Deserialization.

Example 1:

Firstly, we will create a bean inside our config folder as "Config.java"

```java
package com.appsdeveloperblog.ObjectMapperTutorial.config;

import com.fasterxml.jackson.annotation.JsonInclude;

import com.fasterxml.jackson.databind.ObjectMapper;

import com.fasterxml.jackson.databind.PropertyNamingStrategy;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import org.springframework.context.annotation.Primary;

import org.springframework.http.converter.json.Jackson2ObjectMapperBuilder;

@Configuration

class WebConfig {

@Bean

@Primary

public ObjectMapper customJson(){

return new Jackson2ObjectMapperBuilder()

.indentOutput(true)

.serializationInclusion(JsonInclude.Include.NON_NULL)

.propertyNamingStrategy(PropertyNamingStrategy.UPPER_CAMEL_CASE)

.build();

}
```

}

Now if we run our project again and execute GET */getEmployees* , we will see the following output.

```
[ {
  "FirstName" : "damian",
  "JobTitle" : "Developer",
  "Gender" : "Male",
  "Id" : 1
}, {
  "FirstName" : "noel",
  "JobTitle" : "Developer",
  "Gender" : "Male",
  "Id" : 2
}, {
  "FirstName" : "amena",
  "JobTitle" : "CEO",
  "Gender" : "Female",
  "Id" : 3
}, {
  "FirstName" : "eugenia",
  "JobTitle" : "Developer",
  "Gender" : "Male",
  "Id" : 4
}, {
  "FirstName" : "mckee",
  "JobTitle" : "Developer",
  "Gender" : "Female",
  "Id" : 5
}, {
  "FirstName" : "ayers",
  "JobTitle" : "Developer",
  "Gender" : "Female",
  "Id" : 6
} ]
```

Keys changed to UpperCamelCase

If you have noticed, we have configured ObjectMapper to indent the output response. Additionally, we have requested ObjectMapper to convert names to upper case. Hence, now all keys are in camelcase format and output is indented as well after POJO conversion.

Example 2:

Sometimes we have a use-case, where API send additional field JSON than expected in the internal java objects structure. In this case, we can use the

ObjectMapper feature to have stricter checks during serialization to throw exceptions on such cases.

To achieve this, we can modify our config class a below:

```java
package com.appsdeveloperblog.ObjectMapperTutorial.config;

import com.fasterxml.jackson.annotation.JsonInclude;

import com.fasterxml.jackson.databind.DeserializationFeature;

import com.fasterxml.jackson.databind.ObjectMapper;

import com.fasterxml.jackson.databind.PropertyNamingStrategy;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import org.springframework.context.annotation.Primary;

import org.springframework.http.converter.json.Jackson2ObjectMapperBuilder;

@Configuration

class WebConfig {

@Bean

@Primary

public ObjectMapper objectMapper(Jackson2ObjectMapperBuilder builder) {

ObjectMapper objectMapper = builder.createXmlMapper(false).build();

objectMapper.configure(DeserializationFeature.FAIL_ON_UNKNOWN_PROPERTIES, true);

return objectMapper;

}

}
```
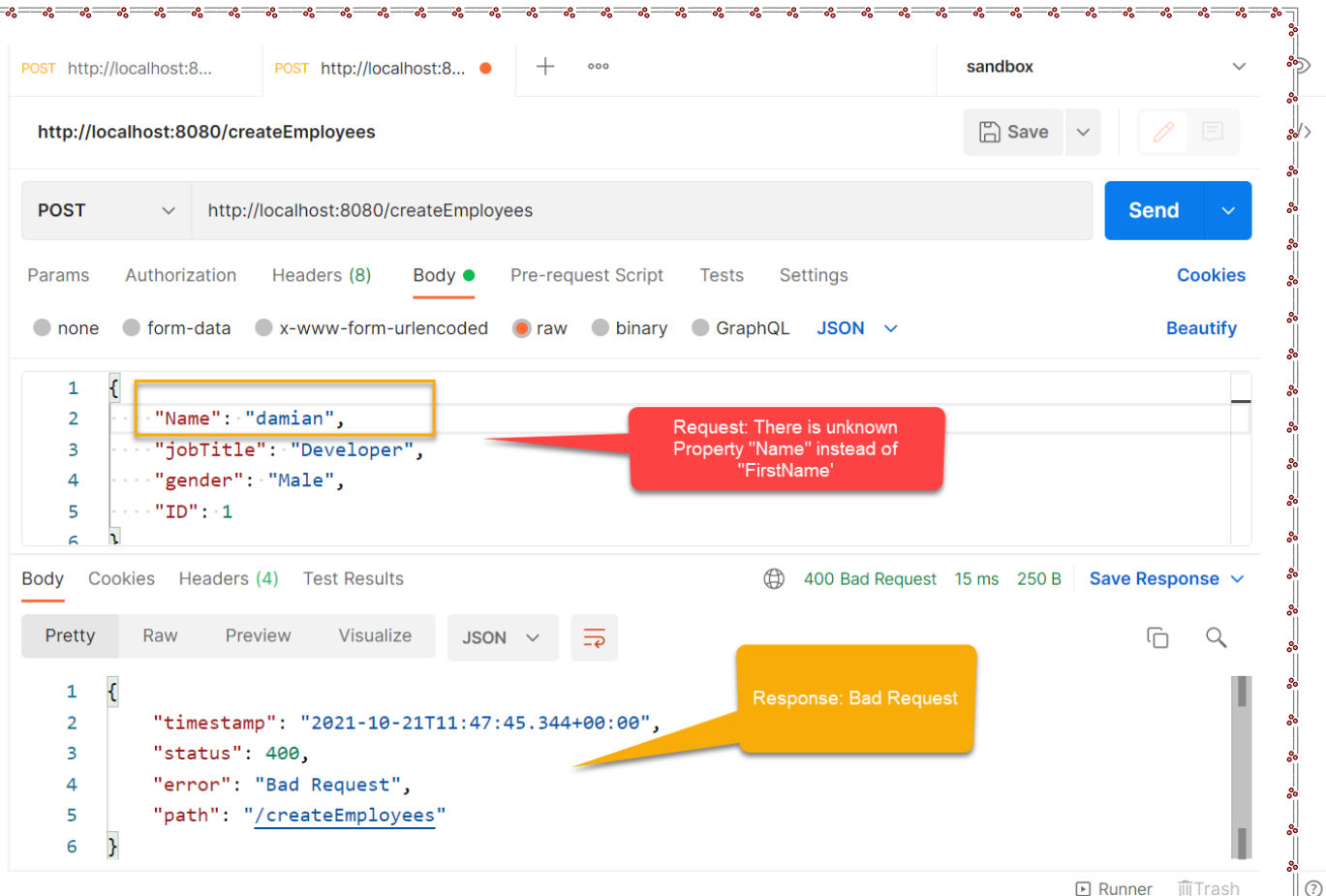
Now, we have introduced the object mapper property "DeserializationFeature.FAIL_ON_UNKNOWN_PROPERTIES" . Consequently, if we send any JSON keys which are not in the Employee object then ObjectMapper will throw an error as below.

Error in the console is as follow:

2021-10-21 17:17:45.343 WARN 28108 --- [nio-8080-exec-4] .w.s.m.s.DefaultHandlerExceptionResolver : Resolved [org.springframework.http.converter.HttpMessageNotReadableException: JSON parse error: Unrecognized field "Name" (**class** com.appsdeveloperblog.ObjectMapperTutorial.model.Employee), not marked as ignorable; nested exception is com.fasterxml.jackson.databind.exc.UnrecognizedPropertyException: Unrecognized field "Name" (**class** com.appsdeveloperblog.ObjectMapperTutorial.model.Employee), not marked as ignorable (4 known properties: "gender", "firstName", "jobTitle", "id"])

at [Source: (PushbackInputStream); line: 2, column: 14] (through reference chain: com.appsdeveloperblog.ObjectMapperTutorial.model.Employee["Name"])]

There are many such properties that can be configured on ObjectMapper, you can follow official Javadoc for *Jackson-databind* features here.

Interacting with ObjectMapper Directly

Usually, with Spring Boot we don't need to create the ObjectMaper instance manually as the Spring Boot framework inherently do it for you. As a good practice, we should avoid creating ObjectMapper instances manually as much as possible. However in certain cases if we need to manually create an ObjectMapper instance to serialize and deserialize our way then we can do as below.

This example shows how to use JACKSON API to convert a Java object into a JSON String. We can use the ObjectMapper class provided by the Jackson API for our conversion.

- **Serialization** : writeValueAsString() is used to convert java object to JSON

```java
ObjectMapper mapper = new ObjectMapper();
try {

String json = mapper.writeValueAsString(cat);

System.out.println("ResultingJSONstring = " + json);

//System.out.println(json);

} catch (JsonProcessingException e) {

e.printStackTrace();

}
```

- **Deserialization** : readValue() is used to convert JSON into java object

```java
Employee emp = new Employee();
emp.setID(200);

emp.setfirstName("David");

emp.setjobTitle("Associate");

emp.gender("Male");

ObjectMapper mapper = new ObjectMapper();

try {

String json = mapper.writeValueAsString(Employee);

System.out.println("ResultingJSONstring = " + json);

//System.out.println(json);

} Catch (JsonProcessingException e) {

e.printStackTrace();

}
```

1. Lambda Expressions

[Lambda expressions](#) are known to many of us who have worked on other popular programming languages like Scala. In Java programming language, a Lambda expression (or function) is just an *anonymous function*, i.e., a **function with no name** and without being bound to an identifier.

Lambda expressions are written precisely where it's needed, typically as a parameter to some other function.

1.1. Syntax

A few basic syntaxes of lambda expressions are:

```
(parameters) -> expression



(parameters) -> { statements; }



() -> expression
```

A typical lambda expression example will be like this:

```
//This function takes two parameters and return their sum
(x, y) -> x + y
```

Please note that based on the type of `x` and `y`, we may use the method in multiple places. Parameters can match to `int`, or `Integer` or simply `String` also. Based on context, it will either add two integers or concatenate two strings.

1.2. Rules for writing lambda expressions

1. A lambda expression can have zero, one or more parameters.

2. The type of the parameters can be explicitly declared or it can be inferred from the context.

3. Multiple parameters are enclosed in mandatory parentheses and separated by commas. Empty parentheses are used to represent an empty set of parameters.

4. When there is a single parameter, if its type is inferred, it is not mandatory to use parentheses.

5. The body of the lambda expressions can contain zero, one, or more statements.

6. If the body of lambda expression has a single statement, curly brackets are not mandatory and the return type of the anonymous function is the same as that of the body expression. When there is more than one statement in the body then these must be enclosed in curly brackets.

Read More: [Java 8 Lambda Expressions Tutorial](#)

2. Functional Interfaces

Functional interfaces are also called *Single Abstract Method interfaces (SAM Interfaces)*. As the name suggests, a functional interface **permits exactly one abstract method** in it.

Java 8 introduces `@FunctionalInterface` annotation that we can use for giving compile-time errors it a functional interface violates the contracts.

2.1. Functional Interface Example

```
//Optional annotation
@FunctionalInterface
public interface MyFirstFunctionalInterface {
    public void firstWork();
}
```

Please note that a functional interface is valid even if the `@FunctionalInterface` annotation is omitted. It is only for informing the compiler to enforce a single abstract method inside the interface.

Also, since default methods are not `abstract` , we can add default methods to the functional interface as many as we need.

Another critical point to remember is that if a functional interface overrides one of the public methods of `java.lang.Object`, that also does not count toward the interface's abstract method count since any implementation of the interface will have an implementation from `java.lang.Object` or elsewhere.

For example, given below is a perfectly valid functional interface.

```
@FunctionalInterface

public interface MyFirstFunctionalInterface

{

    public void firstWork();



    @Override
```

```
    public String toString();                    //Overridden from Object class

    @Override
    public boolean equals(Object obj);           //Overridden from Object class
}
```

Read More: [Java 8 Functional Interface Tutorial](#)

3. Default Methods

Java 8 allows us to add non-abstract methods in the interfaces. These methods must be declared `default` methods. Default methods were introduced in java 8 to enable the functionality of lambda expression.

Default methods enable us to introduce new functionality to the interfaces of our libraries and ensure binary compatibility with code written for older versions of those interfaces.

Let's understand with an example:

```
public interface Moveable {

    default void move(){

        System.out.println("I am moving");

    }

}
```

`Moveable` interface defines a method `move()` and provided a default implementation as well. If any class implements this interface then it need not to implement its own version of `move()` method. It can directly call `instance.move()`. e.g.

```
public class Animal implements Moveable{

    public static void main(String[] args){

        Animal tiger = new Animal();

        tiger.move();

    }

}



Output: I am moving
```

If the class willingly wants to customize the behavior of `move()` method then it can provide its own custom implementation and override the method.

Reda More: [Java 8 Default Methods Tutorial](#)

Method references in java 8.

In [Java 8](#), we can refer a method from class or object using `class::methodName` type syntax.

Types of method references

Java 8 allows four types of method references.

Method Reference	Description	Method reference example
Reference to **static method**	Used to refer static methods from a class	`Math::max` equivalent to `Math.max(x,y)`
Reference to **instance method from instance**	Refer to an instance method using a reference to the supplied object	`System.out::println` equivalent to `System.out.println(x)`
Reference to **instance method from class type**	Invoke the instance method on a reference to an object supplied by the context	`String::length` equivalent to `str.length()`
Reference to **constructor**	Reference to a constructor	`ArrayList::new` equivalent to `new ArrayList()`