SPRING BOOT

1. INTRODUCTION

What is Spring Boot?

Spring Boot is a very popular Java-based framework for building web and enterprise applications. Spring Boot framework provides a wide variety of features addressing the modern business. It provides a simpler and faster way to set up, configure, and run both simple and web-based applications.

Advantages:

Spring Boot offers the following advantages to its developers:

1. Easy to understand and develop spring applications
2. Increases productivity
3. Reduces the development time
4. Reduces the Boiler Plate Code

Why Spring Boot?

You can choose Spring Boot because of the features and benefits it offers as given here:

1. It provides a flexible way to configure Java Beans, XML configurations, and Database Transaction.
2. It provides a powerful batch processing and manages REST endpoints.
3. In Spring Boot, everything is auto configured using annotations.
4. Eases dependency management
5. It includes Embedded Servlet Container

Why do we use Spring Boot instead of spring?

In the Spring core framework, you need to configure all the things manually. Hence, you can have a lot of configuration files, such as XML descriptors. That's one out of the main problems that Spring Boot solves for you.

It smartly chooses your dependencies, auto-configures all the features you will want to use, and you can start your application with one click. Furthermore, it also simplifies the deployment process of your application.

Project Work:

Steps:

1. Applications Required: JDK, JRE, Apache Maven and GitBash.
2. Set the path for JDK, go to window search for Edit Environment Variables ➡ Environment Variables ➡ New ➡ VariableName: JAVA_HOME and VariableValue: path of the jdk C:\Program Files\Java\jdk1.8.0_191\bin
3. Download Spring tool suite from the following link for your respective operating system: https://spring.io/tools/sts/all

2. BUILD SYSTEM

In Spring Boot, choosing a build system is an important task. Gradle and Maven are options available. We recommend Maven as they provide a good support for dependency management.

Dependency Management:

Spring Boot provides a list of dependencies to support the Spring Boot version for its every release. You do not need to provide a version for dependencies in the build configuration file. Spring Boot automatically configures the dependencies version based on the release. Remember that when you upgrade the Spring Boot version, dependencies will also upgrade automatically.

Note: If you want to specify the version for dependency, you can specify it in your configuration file. However, the Spring Boot team highly recommends that it is not needed to specify the version for dependency.

Maven Dependency:

For Maven configuration, we should inherit the Spring Boot Starter parent project to manage the Spring Boot Starters dependencies. For this, simply we can inherit the starter parent in our pom.xml file as shown below.

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>1.4.2. RELEASE</version>
</parent>
```

We should specify the version number for Spring Boot Parent Starter dependency. Then for other starter dependencies, we do not need to specify the Spring Boot version number. Observe the code given below:

```
<dependencies>
  <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-web</artifactId>
  </dependency>
</dependencies>
```

What is groupId and artifactId?

```
<groupId>com.companyname.project-group</groupId>
<artifactId>springboot</artifactId>
<version>1.0</version>
```

groupId:

The groupId is the id of the project's group. Generally, it is unique amongst an organization. According to the above, the groupId is com. companyname.project-group.

artifactId:

The artifactId is the id of the project. It specifies the name of the project. According to the above, the artifactId is springboot.

Naming Conventions on groupId and artifactId:

groupId will identify your project uniquely across all projects, so we need to enforce a naming schema. It must follow the package name rules, what means that has to be at least as a domain name you control, and you can create as many subgroups as you want.

eg. com.capgemini

artifactId is the name of the jar without version. If you created, it then you can choose whatever name you want with lowercase letters and no strange symbols. If it's a third party jar you have to take the name of the jar as it's distributed.

eg. employee

Maven pom.xml file:

POM is an acronym for **Project Object Model**. The pom.xml file contains information of project and configuration information for the maven to build the project such as dependencies, build directory, source directory, test source directory, plugin, goals etc.

Structure of POM:

```
<project>
        <properties>…</properties>
        <parent>…<parent>
        <dependencies>…</dependencies>
        <build>
            <plugins>…<plugins>
        </build>
        <repositories>…</repositories>
</project>
```

Project: It is the root element of the pom.xml file. You need to specify the basic schema settings such as apache schema and w3.org specification.

Parent: It is used to structure the project to avoid redundancies or duplicate configurations using inheritance between pom files. It helps in easy maintenance in long term.

Properties: Maven properties are value placeholder, like properties in Ant. Their values are accessible anywhere within a POM by using the notation ${X}, where X is the property. Or they can be used by plugins as default values.

Dependencies: In order to use Maven, it is necessary to explicitly add dependencies to the pom.xml file. Once added to the Maven pom.xml file, dependencies will be automatically downloaded, updated, and have their sub-dependencies managed by Maven.

Plugins: Plugins are generally used to create jar files, create war files, compile code files and unit testing of code. There are two types of plugins

- **Build plugins:** They execute during the build process and should be configured in the <build/> element of pom.xml.
- **Reporting plugins:** They execute during the site generation process and they should be configured in the <reporting/> element of the pom.xml.

Repositories: A repository in Maven holds build artifacts and dependencies of varying types. There are three types of repositories

- **Local Repository:** Maven local repository is a folder location on your machine. It gets created when you run any maven command for the first time.

  Local repository keeps your project's all dependencies (library jars, plugin jars etc.). When you run a Maven build, then Maven automatically downloads all the dependency jars into the local repository. It helps to avoid references to dependencies stored on remote machine every time a project is build.

- **Central Repository:** Maven central repository is repository provided by Maven community. It contains a large number of commonly used libraries.

  When Maven does not find any dependency in local repository, it starts searching in central repository using following URL − https://repo1.maven.org/maven2/

- **Remote Repository:** Sometimes, Maven does not find a mentioned dependency in central repository as well. It then stops the build process and output error message to console. To prevent such situation, Maven provides concept of **Remote Repository**, which is developer's own custom repository containing required libraries or other project jars.

Dependency and It's Use:

Spring Boot Starter Parent dependency is used for providing dependency and plugin management for Spring-Boot based applications. Its code is shown below:

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>1.4.2. RELEASE</version>
</parent>
```

Spring Boot Starter Web dependency is used to write a Rest Endpoints. Its code is shown below:

```
<dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-web</artifactId>
 </dependency>
```

Spring Boot Starter Actuator dependency is used to monitor and manage your application. Its code is shown below:

```
<dependency>
      <groupId>org.springframework.boot</groupId>
       <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

Spring Boot Starter Security dependency is used for Spring Security. Its code is shown below:

```
<dependency>
       <groupId>org.springframework.boot</groupId>
       <artifactId>spring-boot-starter-security</artifactId>
</dependency>
```

Spring Boot Starter Test dependency is used for writing Test cases. Its code is shown below:

```
<dependency>
        <groupId>org.springframework.boot </groupId>
       <artifactId>spring-boot-starter-test</artifactId>
</dependency>
```

Spring Boot Starter Data JPA dependency is used to provide database access to Spring and JPA. Its code is shown below:

```
<dependency>
        <groupId>org.springframework.boot </groupId>
      <artifactId>spring-boot-data-jpa</artifactId>
</dependency>
```

Spring Boot Starter Tomcat dependency is used to enable an embedded Apache Tomcat 7 instance, by default. Its code is shown below:

```
<dependency>
        <groupId>org.springframework.boot </groupId>
       <artifactId>spring-boot-starter-tomcat</artifactId>
       <scope>provided</scope>
</dependency>
```

Project work:

1. Create a Maven project in STS
2. Add the parent and required dependencies in the pom.xml.

3. KEYPOINTS & FEATURES

Spring Boot does not have any code layout to work with. However, there are some best practices that will help us.

- **Default package**

  A class that does not have any package declaration is considered as a default package. Note that generally a default package declaration is not recommended. Spring Boot will cause issues such as malfunctioning of Auto Configuration or Component Scan, when you use default package.

  Note: Java's recommended naming convention for package declaration is reversed domain name.

Code Structure to be Maintained:

```
com
+- tutorialspringboot
     +-myproject
          +- springboot
          | +- Application.java
          +- model
          | +- Product.java
          +- dao
          | +- ProductRepository.java
          +- controller
          | +- ProductController.java
          +- service
          | +- ProductService.java
```

@AutoConfiguration

SpringBootAutoConfiguration, automatically configures your Spring application based on the JAR dependencies you added in the project. By adding **@EnableAutoConfiguration** annotation or **@SpringBootApplication** annotation to your main class file. Then, your Spring Boot application will be automatically configured.

Observe the following code for a better understanding:

```
@EnableAutoConfiguration
public class SpringBootMainApplication {

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

	}
}
```

Snippet: 3.1

@ComponentScan

Spring Boot application scans all the beans and package declarations when the application initializes. You need to add the **@ComponentScan** annotation for your class file to scan your components added in your project.

Note: Scan with in packages and sub packages.

Observe the following code for a better understanding:

```
@ComponentScan
public class SpringBootApplication {

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

Snippet: 3.2

@SpringBootApplication

The entry point of the Spring Boot Application is the class contains **@SpringBootApplication** annotation. This class should have the main method to run the Spring Boot application. **@SpringBootApplication annotation** includes Autoconfiguration, Component Scan, and Spring Boot Configuration.

If you added **@SpringBootApplication** annotation to the class, you do not need to add the **@EnableAutoConfiguration**, **@ComponentScan** and **@SpringBootConfiguration** annotation. The **@SpringBootApplication** annotation includes all other annotations.

```
@SpringBootApplication
public class SpringBootMainApplication {

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

	}

}
```

Snippet: 3.3

Project Work:

1. Create project hierarchy by creating packages and subpackages

   Ex: com.cg.<artificatId>(Root package),

   com.cg.<artificatId>.controller

   com.cg.<artificatId>.service

   com.cg.<artificatId>.exception and

   com.cg.<artificatId>.logging
2. Create main class in root package and paste Snippet:3.3
3. Now, Rightclick on project-> Maven->UpdateProject
4. Thus our SpringBoot project is created

4. SPRING IOC & DEPENDENCY INJECTION

In Spring Boot, we can use Spring Framework to define our beans and their dependency injection. The **@ComponentScan** annotation is used to find beans and the corresponding injected with **@Autowired** annotation.

If you followed the Spring Boot typical layout, no need to specify any arguments for @ComponentScan annotation. All component class files are automatically registered with Spring Beans. The following example provides an idea about Auto wiring the Rest Template object and creating a Bean for the same:

```
@Bean
public RestTemplate getRestTemplate() {
      return new RestTemplate();
}
```

The following code shows the code for auto wired Rest Template object and Bean creation object in main Spring Boot Application class file:

```
package com.tutorialspoint.demo;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;
import org.springframework.web.client.RestTemplate;
@SpringBootApplication
public class DemoApplication {
@Autowired
     RestTemplate restTemplate;
     public static void main(String[] args) {
           SpringApplication.run(DemoApplication.class, args);
     }
     @Bean
     public RestTemplate getRestTemplate() {
           return new RestTemplate();}
}
```

5. APPLICATION PROPERTIES

Application Properties support us to work in different environments

Properties File:

Properties files are used to keep 'N' number of properties in a single file to run the application in a different environment. In Spring Boot, properties are kept in the **application. properties** file under the class path.

The application. properties file is located in the **src/main/resources** directory. The code for sample application. properties file is given below:

```
server.port=9090
spring.application.name=demoservice
```

Note: that in the code shown above the Spring Boot application demoservice starts on the port 9090.

*# JACKSON (*JacksonProperties)

- spring.jackson.deserialization.*= *# Jackson on/off features that affect the way Java objects are deserialized.*
- spring.jackson.serialization.*= *# Jackson on/off features that affect the way Java objects are serialized.*

Some of the other properties used in spring boot are:

Reference: https://docs.spring.io/spring-boot/docs/current/reference/html/common-application-properties.html

Project Work:

1. Create an application.properties file in src/main/resources Source Folder.
2. Add server.port and the other properties which we require.

6. DATA VALIDATION

Implementing Validations on the Bean

Let's add a few validations to the Employee bean.

```
@Id
@NotNull
private int empId;
@Size(min = 4, message = "Length between 4-8 characters")
private String name;
@Size(min = 1, message = "Length between 1-6 characters")
private String gender;
@Min(21)
private int age;
@NotNull
private double salary;
```

@NotNull – validates that the annotated property value is not null

@AssertTrue – validates that the annotated property value is true

@Size – validates that the annotated property value has a size between the attributes min and max; can be applied to String, Collection, Map, and array properties

@Min – Validates that the annotated property has a value no smaller than the value attribute

@Max – validates that the annotated property has a value no larger than the value attribute

@Email – validates that the annotated property is a valid email address

Note: If the validation does not match it throws an exception and *message* attribute is common to all of them.

Project Work:

1. Add the validations to the bean class for the API developed under API Design and Document for EmployeeManagement.

7. LOMBOK

Lombok is used to reduce boilerplate code for model/data objects, e.g., it can generate getters and setters for those objects automatically by using Lombok annotations. The easiest way is to use the @Data annotation.

Enable Lombok for Spring Tool Suite:

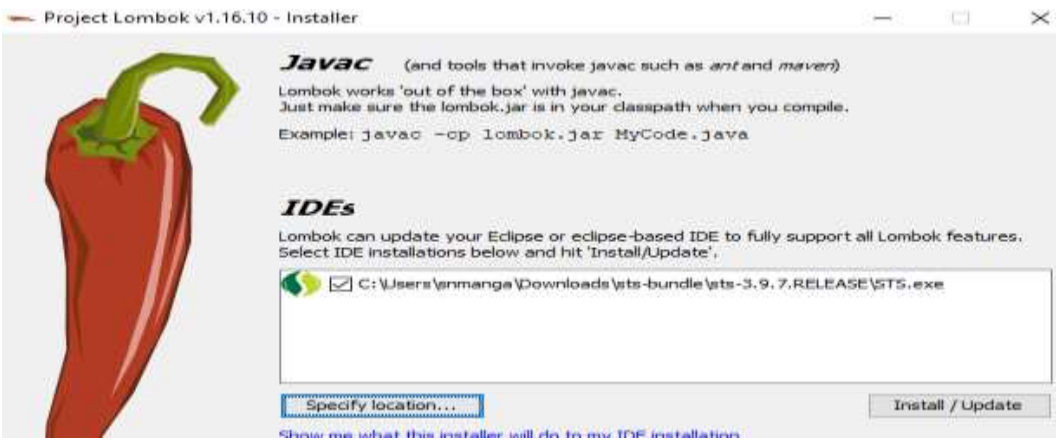
You can download the latest Lombok version by using Maven.

Steps:

1. Add Lombok dependency in pom.xml and update the project.

```
<dependency>
        <groupId>org.projectlombok<groupId>
        <artifactId>lombok</artifactId>
         <version>${lombok.version}<version>
</dependency>
```

2. Then go to System (c:) driver ➡ Users ➡ Yourname ➡ .m2 ➡ repository ➡ org ➡ projectlombok ➡ lombok ➡ version (1.16.10) and run the lombok (1.16.10) jar.
   Example: C:\Users\snmanga\.m2\repository\org\projectlombok\lombok\1.16.10
3. Then it will open a UI, where the location of the Eclipse or STS installation can be specified.



To specify the location press the **specify location** button and select the location of STS IDE then click on Install/Update.

Lombok provides a variety of annotations that we can use and manipulate according to our needs. Some of these annotations are:

@Getter/@Setter: Can be used on a class or field to generate getters and setters automatically for every field inside the class or for a particular field respectively.

```
@Getter
@Setter
public class Employee {
}
```

@NoArgsConstructor, **@RequiredArgsConstructor**, and **@AllArgsConstructor**: Generates constructors that take no arguments, one argument per final/non-null field, or one argument for every field.

@EqualsAndHashCode: Generates hashCode and equals implementations from the fields of your object.

@Data: A shortcut for **@ToString, @EqualsAndHashCode**, **@Getter** on all fields, and **@Setter** on all non-final fields, and **@RequiredArgsConstructor**.

For more annotations refer: https://dzone.com/articles/project-lombok-the-boilerplate-code-extractor-1

Project Work:

1. Install Lombok to STS and add the lombok annotations needed.
2. Thus our POJO classes are ready.
3. Now let's create Controller and Service classes

8. BUILDING A RESTFUL WEBSERVICE

Before starting to build a RESTful web service, let's look into the following annotations:

- **@RestController**

The **@RestController** annotation is used to define the RESTful web services. It serves JSON, XML and custom response. Its syntax is shown below

```
@RestController
public class EmployeeServiceController {
}
```

Note: For building a RESTful Web Services, we need to add the Spring Boot Starter Web dependency into the build configuration file.

```
<dependency>
   <groupId>org.springframework. boot</groupId>
   <artifactId>spring-boot-starter-webservices</artifactId>
</dependency>
```

- **@RequestMapping**

The **@RequestMapping** annotation is used to provide routing information. This annotation maps HTTP requests to handler methods of MVC and REST controllers. It tells to the Spring that any HTTP request should map to the corresponding method. We need to import org.springframework.web. annotation package in our file. It is both class level and method level annotation.

```
@RequestMapping(value = "/employee", method = RequestMethod.GET)
        public List<Employee> getEmployees() {}
```

The value attribute indicates the URL to which the handler method is mapped and the method attribute defines the service method to handle HTTP GET request.

- **@Request Body**

The @RequestBody annotation is used to define the request body content type.

```
public Employee addEmployee(@RequestBody Employee employee) {
}
```

- **@RequestParam and @PathVariable**

@RequestParam and @PathVariable annotations are used for accessing the values from the request.

The key difference between @RequestParam and @PathVariable is that @RequestParam used for accessing the values of the query parameters whereas @PathVariable used for accessing the values from the URI template.

Examples:

For RequestParam:

```
public ResponseEntitygetEmployee (@RequestParam (value="name",
    required=false, defaultValue="honey") String name) {
 }
```

For Path variable:

```
@RequestMapping(value = "/employee/{id}", method = RequestMethod.GET)
        public Employee getEmployeeDetails(@PathVariable int id) {
}
```

- **@ResponseBody** and **@RequestHeader** few other available annotations

***HTTP Methods**:

The following HTTP methods are most commonly used in a REST based architecture.

@GetMapping

This annotation is used for mapping HTTP GET requests onto specific handler methods. @GetMapping is a composed annotation that acts as a shortcut for @RequestMapping(method = RequestMethod.GET)

@PostMapping

This annotation is used for mapping HTTP POST requests onto specific handler methods. @PostMapping is a composed annotation that acts as a shortcut for @RequestMapping(method = RequestMethod.POST)

@PutMapping

This annotation is used for mapping HTTP PUT requests onto specific handler methods. @PutMapping is a composed annotation that acts as a shortcut for @RequestMapping (method = RequestMethod.PUT)

@DeleteMapping

This annotation is used for mapping HTTP DELETE requests onto specific handler methods. @DeleteMapping is a composed annotation that acts as a shortcut for @RequestMapping(method = RequestMethod.DELETE)

Project Work:

1. Create a controller class under controller package and annotate with @RestController.
2. Add methods to implement the Employee Management API's and annotate with relevant annotations.
3. Now let's create Service interface and implementation classes required.
4. Autowire service interface with the controller class (Recall Chapter:4)
5. Run the application and test API calls
6. In the next chapter, Let's see how to implement Logging and Exception Handling using AOP

9. AOP WITH SPRING FRAMEWORK

Aspect Oriented Programming (AOP) compliments OOPs in the sense that it also provides modularity. But the key unit of modularity is aspect than class. AOP breaks the program logic into distinct parts (called concerns). It is used to increase modularity by **cross-cutting concerns**.

A **cross-cutting concern** is a concern that can affect the whole application and should be centralized in one location in code as possible, such as logging, security etc.

AOP Terminologies:

Before we start working with AOP, let us become familiar with the AOP concepts and terminology. These terms are not specific to Spring, rather they are related to AOP.

- **Pointcut**

It is an expression language of AOP that matches join points. Advice is associated with a pointcut expression and runs at any join point matched by the pointcut (for example, the execution of a method with a certain name). The concept of join points as matched by pointcut expressions is central to AOP, and Spring uses the AspectJ pointcut expression language by default.

- **Join point**

A point during the execution of a program, such as the execution of a method or the handling of an exception. In Spring AOP, a join point always represents a method execution.

- **Aspect**

A combination of defining when you want to intercept a method call (Pointcut) and what to do (Advice) is called an Aspect.

- **Advice**

It is the logic that you would want to invoke when you intercept a method. In the below example, it is the code inside the before (Join Point join Point) method.

```
public void before(JoinPoint joinPoint){
//Advice
logger.info(" Check for user access ");
logger.info(" Allowed execution for {}", joinPoint);
}
```

Types of Advice:

Spring aspects can work with five kinds of advice mentioned as follows –

- **@Before**

  Run advice before the a method execution.

  Eg:@Before("execution(* com.example.bean.Register.*(..))")

- **@After**

  Run advice after the method execution, regardless of its outcome.

Eg:@After("execution(* com.example.controller.ExamController.*(..))").

- **@After-returning**

  Run advice after the method execution only if method completes successfully.

Eg:@afterReturning(pointcut= "execution(* com.example.controller.ExamController.*(..))", returning = "retRal")

- **@After-throwing**

  Run advice after the method execution only if method exits by throwing an exception.

eg: @AfterThrowing (pointcut= "execution (* com.example.controller.ExamController.*(..))", throwing = "exc")

- **@Around**

  Run advice before and after the advised method is invoked.

Dependency Needed for Spring AOP

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-aop</artifactId>
    <version>${org.springframework.version}</version>
</dependency>
```

Note: Loggers are the Boiler Plate Code, to reduce these Boilerplates codes we use AOP.

10. LOGGING WITH AOP

Spring Boot uses Apache Common logging for all internal logging. Spring Boot's default configurations provides a support for the use of Java Util Logging, Log4j2, Slf4j and Logback.

Log Format: The default spring boot Log Format is given below

```
2019-02-01 13:03:36.313  INFO 11500 --- [           main] o.s.w.s.handler.SimpleUrlHandlerMapping  : Mapped URL path [/**] onto handl
2019-02-01 13:03:36.313  INFO 11500 --- [           main] .m.m.a.ExceptionHandlerExceptionResolver : Detected @ExceptionHandler metho
2019-02-01 13:03:36.344  INFO 11500 --- [           main] o.s.w.s.handler.SimpleUrlHandlerMapping  : Mapped URL path [/**/favicon.ico
2019-02-01 13:03:36.735  INFO 11500 --- [           main] o.s.j.e.a.AnnotationMBeanExporter        : Registering beans for JMX exposu
2019-02-01 13:03:36.774  INFO 11500 --- [           main] s.b.c.e.t.TomcatEmbeddedServletContainer : Tomcat started on port(s): 8090
2019-02-01 13:03:36.774  INFO 11500 --- [           main] c.c.t.s.SpringBootMainApplication        : Started SpringBootMainApplicatio
```

which gives you the following information -

- **Date** and **Time** that gives the date and time of the log
- **Log level** shows INFO, ERROR or WARN
- **Process ID**
- --- which is a separator
- **Thread name** is enclosed within the square brackets []
- **Logger Name** that shows the Source class name
- The Log message

Console Log Output:

The default log messages will print to the console window. By default, "INFO", "ERROR" and "WARN" log messages will print in the log file.

SLF4J:

The Simple Logging Facade for Java (SLF4J) serves as a simple facade or abstraction for various logging frameworks, allowing the end user to plug in the desired logging framework at deployment time. It is very simple to use slf4j logging in your application. You just need to create a slf4j logger and invoke its methods.

Best practice is to use slf4j for your own log statements, and then choose the appropriate backend for it (including log4j as well by configuring to use log4j as a logging backend).

Here you will see how to add logging of method calls with Spring. On every call you can log method name, method arguments and returned object.

```
        private final Logger slf4jLogger = LoggerFactory.getLogger(this.getClass());
        private ObjectMapper mapper = new ObjectMapper();
        @Before("execution(*com.cg.tutorial.employeemanagement.service.EmployeeServiceImpl..*(..))")
        public void serviceLogging(JoinPoint joint) throws EmployeeNotFoundException {
                mapper.configure(SerializationFeature.FAIL_ON_EMPTY_BEANS, false);
                try {
                        slf4jLogger.info("Begin of - " + joint.getStaticPart().getSignature().getName()
+ " method");
                        slf4jLogger.info("Info Input Parameters -:\n " +
mapper.writerWithDefaultPrettyPrinter().writeValueAsString(joint.getArgs()));
                } catch (JsonProcessingException e) {
                        throw new EmployeeManagementException(400, e.getMessage());
                }
          }
```

In this example, method calls are logged with execution time. If you want to log what method is returning, use @AfterReturnung Advice:

```
@AfterReturning(pointcut = "execution(*
com.cg.tutorial.employeemanagement.service.EmployeeServiceImpl.*(..))", returning = "result")
        public void serviceSetterMethodLogging(JoinPoint joint, Object result) throws
EmployeeNotFoundException {
                mapper.configure(SerializationFeature.FAIL_ON_EMPTY_BEANS, false);
                try {
```

```
                    slf4jLogger.info("Info Output Parameters -: \n "
                                    +
mapper.writerWithDefaultPrettyPrinter().writeValueAsString(null != result ? result : ""));
                    slf4jLogger.debug("end of  - " + joint.getStaticPart().getSignature().getName()
+ " method");
                } catch (JsonProcessingException e) {
                    throw new EmployeeManagementException(400, e.getMessage());
                }
```

In this application we have used two log levels i.e. INFO and DEBUG

INFO: INFO messages correspond to normal application behavior. They provide the skeleton of what happened. A service started or stopped. You added a new user to the database. That sort of thing.

DEBUG: Information that is diagnostically helpful to people more than just developers (IT, sysadmins, etc.).

Project Work:

1. Create LoggingHandler class under logging package and add the methods in the above snippet. Edit the pointcut expression as needed. By this way we log the input (@Before)and output (@AfterReturning) from Service methods
2. Run the application and check the logs by making API calls.

11. EXCEPTION HANDLING

Handling exceptions and errors in APIs and sending the proper response to the client is good for enterprise applications.

Before proceeding with exception handling, let us gain an understanding on the following annotations.

@ControllerAdvice:

The @ControllerAdvice is an annotation, to handle the exceptions globally.

@ExceptionHandler:

The @ExceptionHandler is an annotation used to handle the specific exceptions and sending the custom responses to the client.

You can use the following code to create @ControllerAdvice class to handle the exceptions globally −

```
import org.springframework.web.bind.annotation.ControllerAdvice;

@ControllerAdvice
  public class EmployeeExceptionController {}
```

Define a class that extends the RuntimeException class.

```
public class EmployeeManagementException extends RuntimeException {
   private static final long serialVersionUID = 1L;}
```

You can define the @ExceptionHandler method to handle the exceptions as shown. This method should be used for writing the Controller Advice class file.

```
@ExceptionHandler(value = EmployeeManagementException.class)
public ResponseEntity<Object> exception(EmployeeManagementException exception) {
}
```

Now, use the code given below to throw the exception from the API.

```
@RequestMapping(value = "/employees/{id}", method = RequestMethod.PUT)
public ResponseEntity<Object> updateEmployee() {
  throw new EmployeeManagementException ();
}
```

In this example, we used the PUT API to update the employee. Here, while updating the employee, if the employee is not found, then return the response error message as "Employee not found".

Note: The EmployeeManagementException exception class should extend the **RuntimeException**.

```
public class EmployeeManagementException extends RuntimeException {
private static final long serialVersionUID = 1L;
}
```

The Controller Advice class to handle the exception globally is given below. We can define any Exception Handler methods in this class file.

```
@ControllerAdvice
public class EmployeeExceptionController {
  @ExceptionHandler(value = EmployeeManagementException.class)
  public ResponseEntity<Object> exception(EmployeeManagementException exception) {
     return new ResponseEntity<>("Employee not found", HttpStatus.NOT_FOUND);
 }
}
```

Project Work:

1. Create three Exception classes i.e. ExceptionHandler, ExceptionResult and EmployeeManagementException under Exception package.
2. As mentioned above, add the @ControllerAdvice and @ExceptionHandler annotations to the Exception Handler class.
3. Then add throws to the method that you want to check in the controller class.
4. Run the application and check the exception by making API calls for inconsistent data.

12. UNIT TEST CASES

Unit Testing is a one of the testing done by the developers to make sure individual unit or component functionalities are working fine.

The important dependency for unit testing is **spring-boot-starter-test**

```
<dependency>
        <groupId>org.springframework. boot </groupId>
        <artifactId>spring-boot-starter-test</artifactId>
</dependency>
```

Spring Boot Test Starter is Starter for testing Spring Boot applications with libraries including JUnit, AssertJ and Mockito.

Unit Testing with Mockito using MockitoRunner

Code below shows a unit test with Mockito using MockitoJUnitRunner.

```
@RunWith(MockitoJUnitRunner.class)
public class EmployeeTest {

        @Mock
        EmployeeRepository empRepo;

        @InjectMocks
        private EmployeeServiceImpl employeeService;

        @Test
        public void getEmployeeDetails() throws Exception {
                Employee = new Employee();

                employee.setEmpId(1);
                employee.setName("Sneha");
                employee.setAge(21);
                employee.setGender("female");
                employee.setSalary(20000);
                when(empRepo.findOne(1)).thenReturn(employee);
                Employee empExpected =
employeeService.getEmployeeDetails(employee.getEmpId());
                assertEquals(employee,empExpected);
```

```
        }

        @Test
        public void getEmployees() {

                List<Employee> allEmployees = Arrays.asList();
                when(empRepo.findAll()).thenReturn(allEmployees);
                List<Employee> empExpec = employeeService.getEmployees();
                assertEquals(allEmployees, empExpec);
        }
}
```

Snippet: 12.1

Note:

- @RunWith (MockitoJUnitRunner.class) public class EmployeeTest {
- The JUnit Runner which causes all the initialization magic with @Mock and @InjectMocks to happen before the tests are run.
- @Mock

  EmployeeRepository prodRepo;

  Creates a mock for EmployeeRepository.

- @InjectMocks

  Private EmployeeServiceImpl employeeService;

  Inject the mocks as dependencies into EmployeeServiceImpl.

- There are two test methods testing two different scenarios –

  getEmployeeDetails and getEmployees.

- **@Mock** creates a mock implementation for the classes you need.
- **@InjectMock** creates an instance of the class and injects the mocks that are marked with the annotations @Mock into it.
- **Junit Assert** Assert is a method useful in determining Pass or Fail status of a test case.

- **Assert Equals** If you want to test equality of two objects, you have the following methods **assertEquals(expected, actual)**

  It will return true if: **expected.equals(actual)** returns true.

Project Work:

1. Create a junit test case under src/test/java source folder.
2. Then write the test cases for the methods of service class.
3. Use the annotations as mentioned in the above snippet 12.1.
4. Run the application and check the test cases.