# UNIT 9 INTRODUCTION TO SPRING BOOT

**Structure** **Page No.**

9.0 Introduction
9.1Objectives
9.2 Spring Boot: An Overview
   9.2.1 Spring Boot Working and Annotations
   9.2.2 Spring Boot Starter
   9.2.3 Spring Boot Project Structure
   9.2.4 Spring Boot Runners
   9.2.5 Spring Boot Web Application Using Thymeleaf
9.3 Spring Boot DevTools and Spring Boot Actuator
   9.3.1 Spring Boot DevTools
   9.3.2 Spring Boot Actuator
   9.3.3 Spring Boot Actuator Example
9.4 Spring Boot- Application Properties
   9.4.1 Command Line Properties
   9.4.2 Properties File
   9.4.3 YAML File
   9.4.4 Externalized Properties
   9.4.5 @Value annotation
   9.4.6 Active Profile
   9.4.7 Spring Active Profile in application.yml
9.5 Running Spring Boot Apps from command line
   9.5.1 Running the Code with Maven in Exploded Form
   9.5.2 Running the Code as a Stand-Alone Packaged Application
   9.5.3 Application Packaging as WAR and Deployment on Tomcat
9.6 Summary
9.7 Solutions/Answer to Check Your Progress
9.8 References/Further Reading

## 9.0 INTRODUCTION

Fast, secured and cost-effective development of software plays a vital role in the growth of an organization. Java has been the popular choice for many enterprises and programmers since the mid 90's for designing and developing software. Hence, lots of frameworks are being developed to ease the software development using Java.

In order to make the software development easy, fast and effective several frameworks have been introduced such as Struts, Spring and ORM tools such as Toplink, Hibernate, Eclipse Link, Ibatis.

Spring based enterprise application requires many configurations and complicated dependency management which make the set up tedious and error-prone. Each time spring-based enterprise application requires repetition of the same configuration steps:

… Based on application type, import the required dependencies such as spring mvc, spring jpa, spring jdbc, spring rest etc.

- Import the specified Spring version compatible third-party libraries such as hibernate, Jackson etc.
- Configure web layer beans such as view resolver, resource manager etc.
- Configure DAO layers such as data source, transaction management, entity manager etc.
- Import web container libraries in the case of web applications.

The spring team considered simplifying the routine configuration and facilitating the developers with some utility that automates the configuration process and speeds up the spring-based application's build and deployment process. Spring has evolved a lot in recent years and introduced many new modules such as Spring Boot, Spring Security, Spring Cloud, Spring Data etc.

***Spring Boot*** is a utility project which enables an organization to develop production-ready spring-based applications and services with less effort, reduced cost and minimal configuration. ***Spring Boot*** facilitates the developers to create a Spring based web application with minimum lines of code since it provides auto-configuration out-of-the-box. In this unit Tomcat and Tomcat are used interchangeably.

## 9.1 OBJECTIVES

After going through this unit, you will be able to:

- describeSpring Boot and its features,
- describeSpring Boot starters and runners,
- use Spring Boot annotations,
- set application properties and profile-based application properties,
- use DevTools and Actuator,
- perform command line Spring Boot application execution, and
- perform step by step development and execution of Spring Boot web app.

## 9.2 SPRING BOOT:AN OVERVIEW

***Spring Boot*** is an extension of the Spring framework that takes one step ahead and simplifies the configuration in order to fasten the development and execution of the spring application. It eliminates the boilerplate configuration required for a spring application. It is a module that enriches the Spring framework with Rapid Application Development (RAD) feature. It provides an easy way to create a stand-alone and production ready spring application with minimum configurations. Spring Boot provides comprehensive infrastructure support for the development and monitoring of the enterprise-ready applications based on micro services. Spring Boot is a combination of **Spring framework with auto-configuration** and **embedded Servers**.

Spring Boot provides a vast number of features and benefits. A few of them are as follows:

1) Everything is auto-configured in Spring Boot.

2) Spring Boot starter eases the dependency management and application configuration
3) It simplifies the application deployment by using an embedded server
4) Production ready features to monitor and manage applications, such as health checks, metrics gathering etc.
5) Reduces the application development time and run the application independently
6) Very easy to understand and develop Spring application
7) Increases productivity and reduces the cost

## 9.2.1 Spring Boot Working and Annotations

A class with the main method and annotated with **@SpringBootApplication** is the entry point of the Spring Boot application. Spring Boot auto-configures all required configurations. It performs auto-configuration by scanning the classes in class-path annotated with @Component or @Configuration. @SpringBootApplication annotation comprising the following three annotations with their default values-

- @EnableAutoConfiguration
- @ComponentScan
- @Configuration

**@EnableAutoConfiguration**

As the name implies, it enables auto-configuration. It means **Spring Boot** looks for auto-configuration beans on its class-path and automatically applies them. For example, if H2 dependency is added and you have not manually configured any database connection beans, then Spring will auto-configure H2 as an in-memory database.

The *@EnableAutoConfiguration* annotation should be applied in the root package so that every sub-packages and class can be examined.

**@ComponentScan**

The ***@ComponentScan*** annotation enables Spring to scan for things like configurations, controllers, services, and other components we define. By default, the scanning starts from the package of the class declaring @ComponentScan. The component scanning can be started from specified packages by defining ***basePackageClasses***, ***basePackages*** attributes into **@ComponentScan**. The **@ComponentScan** annotation is used with **@Configuration.**

**@Configuration**

The @Configuration annotation is used for annotation-based configuration into the Spring application. A class annotated with **@Configuration** indicates that the class declares one or more **@Bean** methods and can be processed by Spring container to generate bean definitions and service requests for those beans at runtime.

Other available annotations in the Spring Boot are described below:

**@ConditionalOnClass and @ConditionalOnMissingClass**

Spring @ConditionalOnClass and @ConditionalOnMissingClass annotations let @Configuration classes be included based on the presence or absence of specific classes. Hence, @ConditionalOnClass loads a bean only if a certain class is on the classpath and @ConditionalOnMissingClass loads a bean only if a certain class is not on the classpath.

**@CoditionalOnBean and @ConditionalOnMissingBean**

These annotations are used to define conditions based on the **presence or absence** of a specific bean.

**@ConditionalOnProperty**

The annotation is used to conditionally create a Spring bean depending on the configuration of a property.

```
@Bean

@ConditionalOnProperty(name = "usepostgres", havingValue = "local")

DataSourcedataSource()

{

    // ...

}
```

The DataSource bean will be created only if property **usepostgres** exists and it has value as **local.**

**@ConditionalOnResource**

The annotation is used to conditionally create a Spring bean based on the presence or absence of resources. The SpringConfig class beans will be created if the resource log4j.properties file is present.

```
@Configuration

@ConditionalOnResource(resources = { "log4j.properties" })

class SpringConfig

{
        @Bean
        public Log4j log4j()
        {
                return new Log4j();
        }
}
```

## 9.2.2 Spring Boot Starter

Before describing Spring Boot Starter, the following section explains all the required things to develop a web application.

Development of a web application using Spring MVC will require identifying all the required dependencies, compatible versions and how to connect them together. Followings are some of the dependencies which are used in Spring MVC web application. For all the dependencies we need to choose the compatible version dependencies.

```
<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-webmvc</artifactId>

<version>4.2.2.RELEASE</version>

</dependency>



<dependency>

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

<artifactId>jackson-databind</artifactId>

<version>2.5.3</version>

</dependency>



<dependency>

<groupId>org.hibernate</groupId>

<artifactId>hibernate-validator</artifactId>

<version>5.0.2.Final</version>

</dependency>

<dependency>

<groupId>log4j</groupId>

<artifactId>log4j</artifactId>

<version>1.2.17</version>

</dependency>
```

Dependency management is a very important and complex process in any project. Manual dependency management is error-prone, non-ideal and non-suggestible. To focus more on aspects apart from dependency management in web application development will require some solution that addresses compatible dependency management problems.

Spring Boot starters are the dependency descriptor that addresses the problem of compatible versions of dependency management. Spring Boot Starter POMs are a set of convenient dependency descriptors that you can include in your application. You get a one-stop-shop for all the Spring and related technology that you need from Spring Boot starters. Spring Boot provides a number of starters that make development easier and rapid. Spring Boot starters follow a similar naming pattern as spring-boot-starter-*, where * denotes a particular type of application. For instance, developing a rest service that requires libraries like Spring MVC, Tomcat, Log and Jackson, a single Spring Boot starter named spring-boot-starter-web needs to be included. Spring Boot provides many numbers starter, and a few of them are listed below-

| | |
|---|---|
| **spring-boot-starter-web** | It is used for building web applications, including RESTful applications using Spring MVC. It uses Tomcat as the default embedded container. |
| **spring-boot-starter-jdbc** | It is used for JDBC with the Tomcat JDBC connection pool. |
| **spring-boot-starter-validation** | It is used for Java Bean Validation with Hibernate Validator. |
| **spring-boot-starter-security** | It is used for Spring Security. |
| **spring-boot-starter-data-jpa** | It is used for Spring Data JPA with Hibernate. |
| **spring-boot-starter** | It is used as a core starter, including auto-configuration support, logging etc. |
| **spring-boot-starter-test** | It is used to test Spring Boot applications with libraries, including JUnit, Hamcrest, and Mockito. |
| **spring-boot-starter-thymeleaf** | It is used to build MVC web applications using Thymeleaf views. |

## 9.2.3 Spring Boot Project Structure

Spring Boot provides a high degree of flexibility in code layout. However, there are certain best practices that will help to organize the code for better readability.

In Spring Boot, default package declaration is not recommended since it can cause issues such as malfunctioning of auto-configuration or Component Scan. Spring Boot annotationslike:

*@ComponentScan*,

*@EntityScan,*

*@ConfigurationPropertiesScan,* and

*@SpringBootApplication*

use packages to define scanning locations. Thus, the default package should be avoided.

The *@SpringBootApplication* annotation triggers component scanning for the current package and its sub-packages. Therefore, the main class of the project should reside in the base package to use the implicit components scan of Spring Boot.

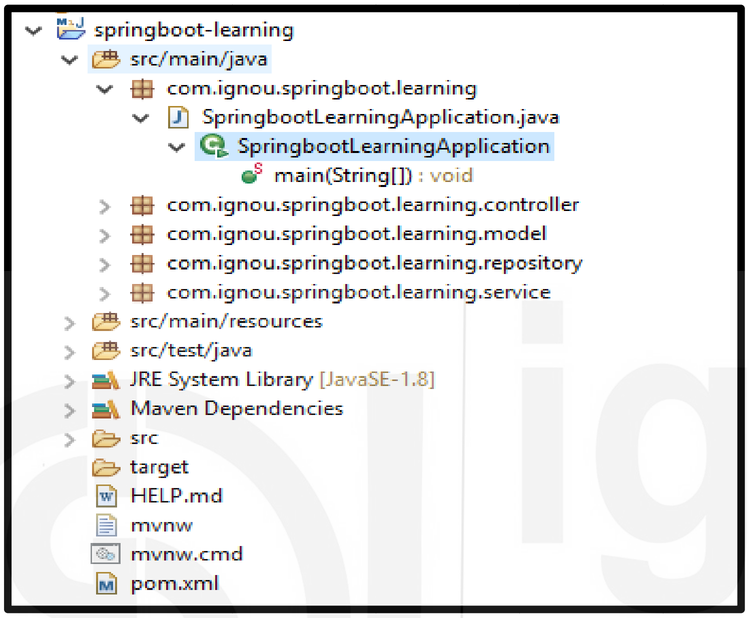
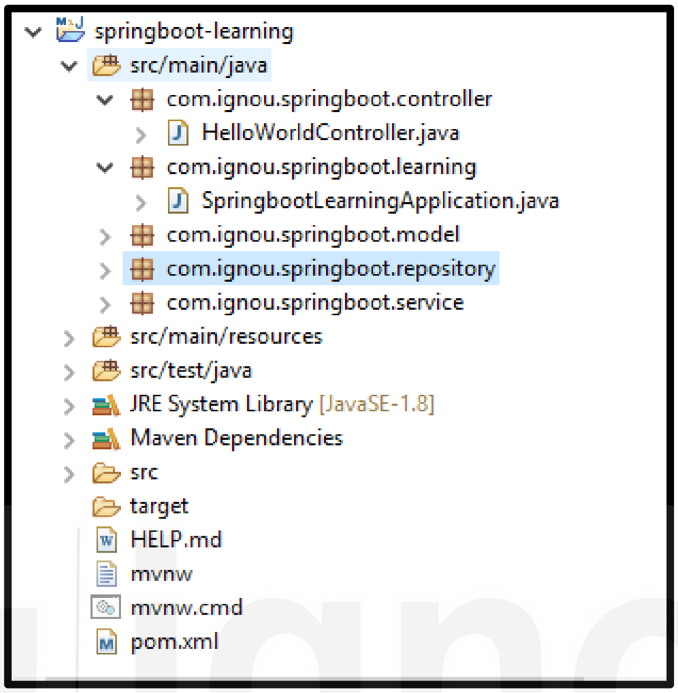

```
springboot-learning
  src/main/java
    com.ignou.springboot.learning
      SpringbootLearningApplication.java
        SpringbootLearningApplication
          main(String[]) : void
    com.ignou.springboot.learning.controller
    com.ignou.springboot.learning.model
    com.ignou.springboot.learning.repository
    com.ignou.springboot.learning.service
  src/main/resources
  src/test/java
  JRE System Library [JavaSE-1.8]
  Maven Dependencies
  src
  target
  HELP.md
  mvnw
  mvnw.cmd
  pom.xml
```

**Figure 9.1: SpringBoot Project Structure For Default Scanning**

In the above code structure, controller, model, repository and service are sub-package of base package **com.ignou.springboot.learning.** All the components and configurations are eligible for implicit scanning with main class as below:

```
package com.ignou.springboot.learning;

importorg.springframework.boot.SpringApplication;

importorg.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication

public class SpringbootLearningApplication

{
        public static void main(String[] args)
        {

        SpringApplication.run(SpringbootLearningApplication.class, 
args);
        }
}
```

Spring Boot has a high degree of flexibility. Thus, it allows us to relocate the scanning elsewhere by specifying the base package manually.



```
springboot-learning
  src/main/java
    com.ignou.springboot.controller
      HelloWorldController.java
    com.ignou.springboot.learning
      SpringbootLearningApplication.java
    com.ignou.springboot.model
    com.ignou.springboot.repository
    com.ignou.springboot.service
  src/main/resources
  src/test/java
  JRE System Library [JavaSE-1.8]
  Maven Dependencies
  src
  target
  HELP.md
  mvnw
  mvnw.cmd
  pom.xml
```

**Figure9.2: SpringBoot Project Structure For Custom Scan Package**

```
package com.ignou.springboot.learning;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication(scanBasePackages= {"com.ignou.springboot"})

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

In above package structure, component and configuration packages are not sub-package of main class. Thus, scanning location has been explicitly defined as @SpringBootApplication(scanBasePackages= {"com.ignou.springboot"})

### 9.2.4 Spring Boot Runners

Spring Boot provides two runner interfaces named as ApplicationRunner and CommandLineRunner. These interfaces enable you to execute a piece of code just after the Spring Boot application is started.

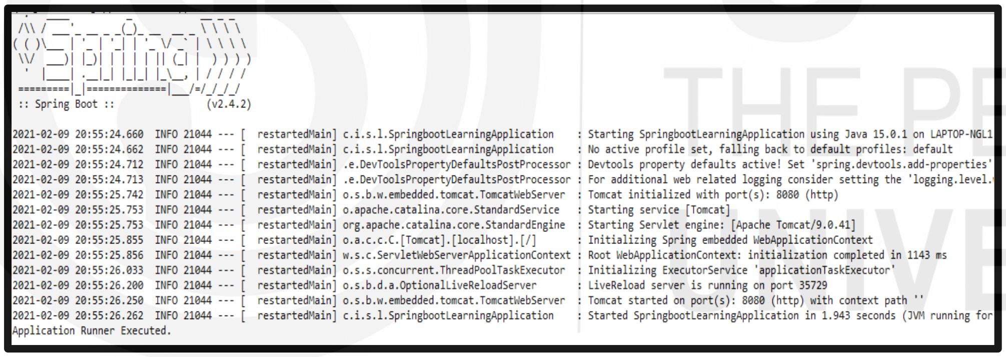
Both interfaces are Functional Interface. If any piece of code needs to be executed when Spring Boot Application starts, we can implement either of these functional interfaces and override the single method**run.**

## Application Runner

Spring bean which implements the **ApplicationRunner**interfaces in a Spring Boot Application, the bean gets executed when application is started. ApplicationRunner example is as following-

```
packagecom.ignou.springboot.learning;

importorg.springframework.boot.ApplicationArguments;

importorg.springframework.boot.ApplicationRunner;

importorg.springframework.stereotype.Component;

@Component

publicclassApplicationRunnerImplimplementsApplicationRunner
{
        @Override
        publicvoid run(ApplicationArgumentsargs) throws Exception
        {
                System.out.println("Application Runner Executed.");
        }
}
```

The execution result of Spring Boot application having the above component is shown below, and you can observe that the bean implementing ApplicationRunner is executed just after the application starts.



```
  /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
 ( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
  \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
   '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::                (v2.4.2)

2021-02-09 20:55:24.660  INFO 21044 --- [  restartedMain] c.i.s.l.SpringbootLearningApplication    : Starting SpringbootLearningApplication using Java 15.0.1 on LAPTOP-NGL1
2021-02-09 20:55:24.662  INFO 21044 --- [  restartedMain] c.i.s.l.SpringbootLearningApplication    : No active profile set, falling back to default profiles: default
2021-02-09 20:55:24.712  INFO 21044 --- [  restartedMain] .e.DevToolsPropertyDefaultsPostProcessor : Devtools property defaults active! Set 'spring.devtools.add-properties'
2021-02-09 20:55:24.713  INFO 21044 --- [  restartedMain] .e.DevToolsPropertyDefaultsPostProcessor : For additional web related logging consider setting the 'logging.level.
2021-02-09 20:55:25.742  INFO 21044 --- [  restartedMain] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat initialized with port(s): 8080 (http)
2021-02-09 20:55:25.753  INFO 21044 --- [  restartedMain] o.apache.catalina.core.StandardService   : Starting service [Tomcat]
2021-02-09 20:55:25.753  INFO 21044 --- [  restartedMain] org.apache.catalina.core.StandardEngine  : Starting Servlet engine: [Apache Tomcat/9.0.41]
2021-02-09 20:55:25.855  INFO 21044 --- [  restartedMain] o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing Spring embedded WebApplicationContext
2021-02-09 20:55:25.856  INFO 21044 --- [  restartedMain] w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initialization completed in 1143 ms
2021-02-09 20:55:26.033  INFO 21044 --- [  restartedMain] o.s.s.concurrent.ThreadPoolTaskExecutor  : Initializing ExecutorService 'applicationTaskExecutor'
2021-02-09 20:55:26.200  INFO 21044 --- [  restartedMain] o.s.b.d.a.OptionalLiveReloadServer       : LiveReload server is running on port 35729
2021-02-09 20:55:26.250  INFO 21044 --- [  restartedMain] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat started on port(s): 8080 (http) with context path ''
2021-02-09 20:55:26.262  INFO 21044 --- [  restartedMain] c.i.s.l.SpringbootLearningApplication    : Started SpringbootLearningApplication in 1.943 seconds (JVM running for
Application Runner Executed.
```

**Figure 9.3: Execution Result of ApplicationRunner**

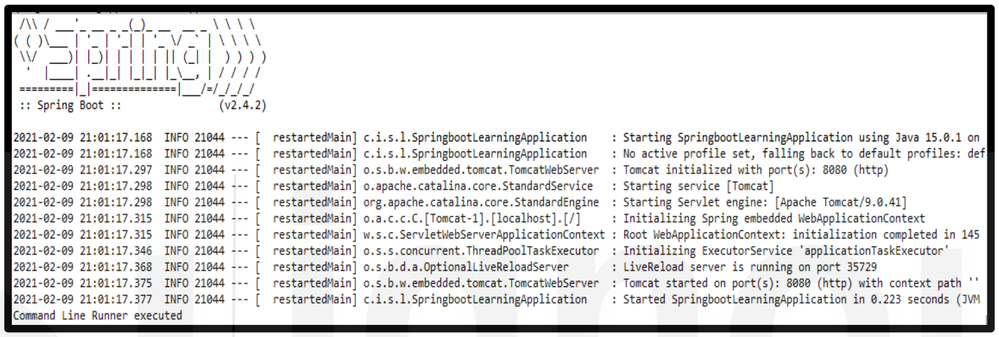
## Command Line Runner

The ***Command LineRunner*** interface is also meant for the same purpose as ApplicationRunner interface. Spring bean which implements the ***CommandLineRunner*** interface in a Spring Boot Application, the bean gets executed when the application is started. ***CommandLineRunner*** example is as following-

```
packagecom.ignou.springboot.learning;

importorg.springframework.boot.CommandLineRunner;

importorg.springframework.stereotype.Component;

@Component
```

```
publicclassCommandLineRunnerImplimplementsCommandLineRunner

{
        @Override
        publicvoid run(String... args) throws Exception
        {
                System.out.println("Command Line Runner executed");
        }
}
```

Execution result of Spring Boot application with the above component is shown below and you can observe that the bean implementing ApplicationRunner is executed just after application starts.



```
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::                (v2.4.2)

2021-02-09 21:01:17.168  INFO 21044 --- [  restartedMain] c.i.s.l.SpringbootLearningApplication    : Starting SpringbootLearningApplication using Java 15.0.1 on
2021-02-09 21:01:17.168  INFO 21044 --- [  restartedMain] c.i.s.l.SpringbootLearningApplication    : No active profile set, falling back to default profiles: def
2021-02-09 21:01:17.297  INFO 21044 --- [  restartedMain] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat initialized with port(s): 8080 (http)
2021-02-09 21:01:17.298  INFO 21044 --- [  restartedMain] o.apache.catalina.core.StandardService   : Starting service [Tomcat]
2021-02-09 21:01:17.298  INFO 21044 --- [  restartedMain] org.apache.catalina.core.StandardEngine  : Starting Servlet engine: [Apache Tomcat/9.0.41]
2021-02-09 21:01:17.315  INFO 21044 --- [  restartedMain] o.a.c.c.C.[Tomcat-1].[localhost].[/]     : Initializing Spring embedded WebApplicationContext
2021-02-09 21:01:17.315  INFO 21044 --- [  restartedMain] w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initialization completed in 145
2021-02-09 21:01:17.346  INFO 21044 --- [  restartedMain] o.s.s.concurrent.ThreadPoolTaskExecutor  : Initializing ExecutorService 'applicationTaskExecutor'
2021-02-09 21:01:17.368  INFO 21044 --- [  restartedMain] o.s.b.d.a.OptionalLiveReloadServer       : LiveReload server is running on port 35729
2021-02-09 21:01:17.375  INFO 21044 --- [  restartedMain] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat started on port(s): 8080 (http) with context path ''
2021-02-09 21:01:17.377  INFO 21044 --- [  restartedMain] c.i.s.l.SpringbootLearningApplication    : Started SpringbootLearningApplication in 0.223 seconds (JVM
Command Line Runner executed
```

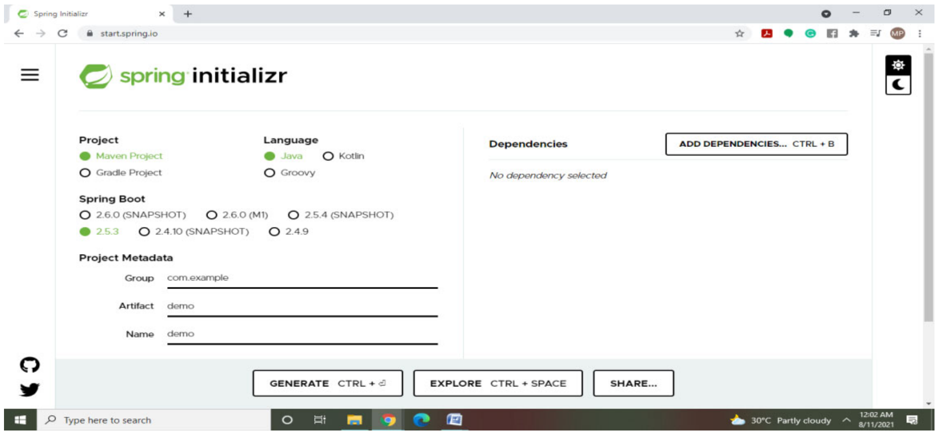
**Figure 9.4: Execution Result Of CommandLineRunner**

## 9.2.5 Spring Boot Web Application Using Thymeleaf

The previous sections have described the basic concepts of Spring Boot. This section explains the process of creating a "Hello World" website using the Spring Boot concepts. Required tools and software are as follows:

- … Eclipse
- … Maven
- … Spring Boot
- … JDK 8 or above

Step 1: Create a Spring Boot Project using Spring Initializr. Visit at **https://start.spring.io/** and generate the Spring Boot project with added dependency. For the project,**Spring Web** and **Thymeleaf** dependency will be used.



Project: [✓ Maven Project] Gradle Project
Language: [✓ Java] Kotlin Groovy
Spring Boot: 2.6.0 (SNAPSHOT) 2.6.0 (M1) 2.5.4 (SNAPSHOT) [✓ 2.5.3] 2.4.10 (SNAPSHOT) 2.4.9
Project Metadata
Group: com.example
Artifact: demo
Name: demo
Dependencies: No dependency selected

**Figure 9.5: SpringBoot Project Initialization Using Spring Initializr**

Step 2: Extract the generated project and Import this into eclipse as maven project. Project structure is shown below.

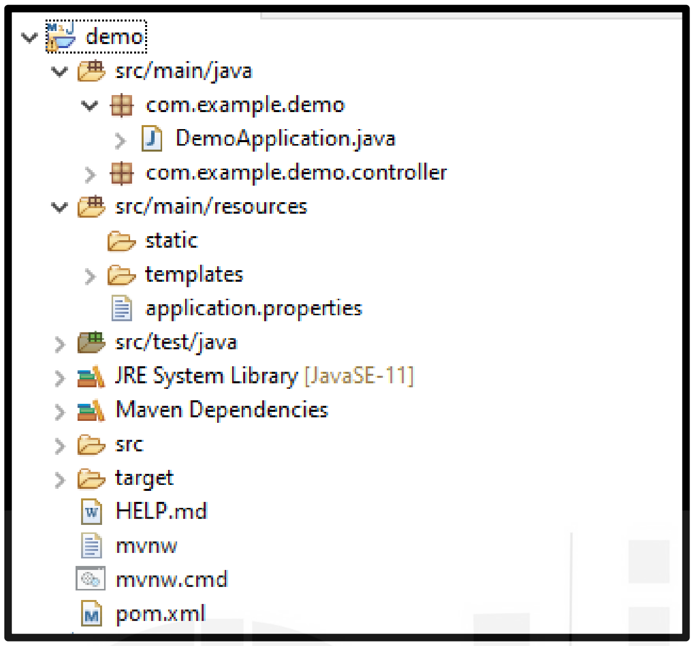


**Figure 9.6: SpringBoot Project Structure**

The controller is a sub-package of the base package. Thus, implicit scanning of components and configuration will be used.

Step 3: Add a HomeController into the controller package. HomeController has two endpoints:

… "/"

… "/greeting?name=Jack"

```
packagecom.example.demo.controller;
importorg.springframework.stereotype.Controller;
importorg.springframework.ui.Model;
importorg.springframework.web.bind.annotation.GetMapping;
importorg.springframework.web.bind.annotation.RequestMapping;
importorg.springframework.web.bind.annotation.RequestParam;
@Controller

publicclassHomeController
{
        @RequestMapping(value="/")
        public String home()
        {
                return"home";
        }
        @GetMapping("/greeting")
        public String greeting(@RequestParam(name="name",
required=false,defaultValue="World") String name, Model model)
        {
                model.addAttribute("name", name);
                return"greeting";
        }
}
```

Step 4: Thymeleaf is used as a view technology in this example. For both the end point, add the corresponding HTML template file into src/main/resources/templates

home.html

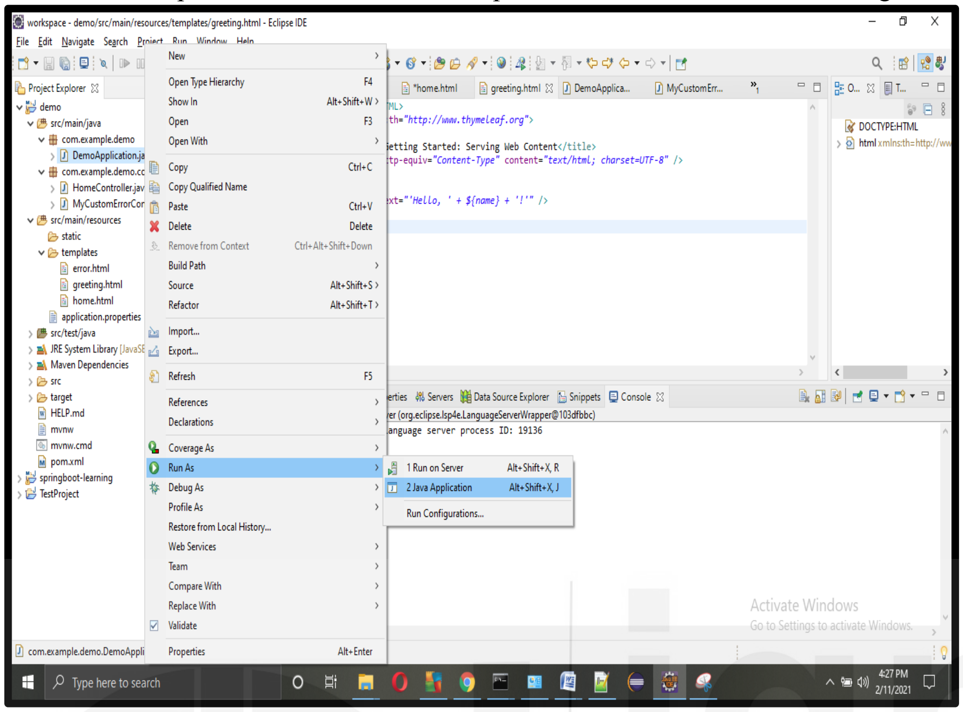
```
<!DOCTYPEhtml>

<html>

<head>

<metacharset="ISO-8859-1"/>

<title>Spring Boot Web Application</title>

</head>

<body>

<h1>Welcome to Thymeleaf Spring Boot web application</h1>

</body>

</html>
```

greeting.html

```
<!DOCTYPEHTML>

<htmlxmlns:th="http://www.thymeleaf.org">

<head>

<title>Getting Started: Serving Web Content</title>

<metahttp-equiv="Content-Type"content="text/html; charset=UTF-8"/>

</head>

<body>

<pth:text="'Hello, ' + ${name} + '!'"/>

</body>

</html>
```

Step 5: Now the project is ready to run. Run **DemoApplication.java** file as Java Application in eclipse as shown below.

Access the endpoints, as mentioned in step 3, into the browser. You will get the

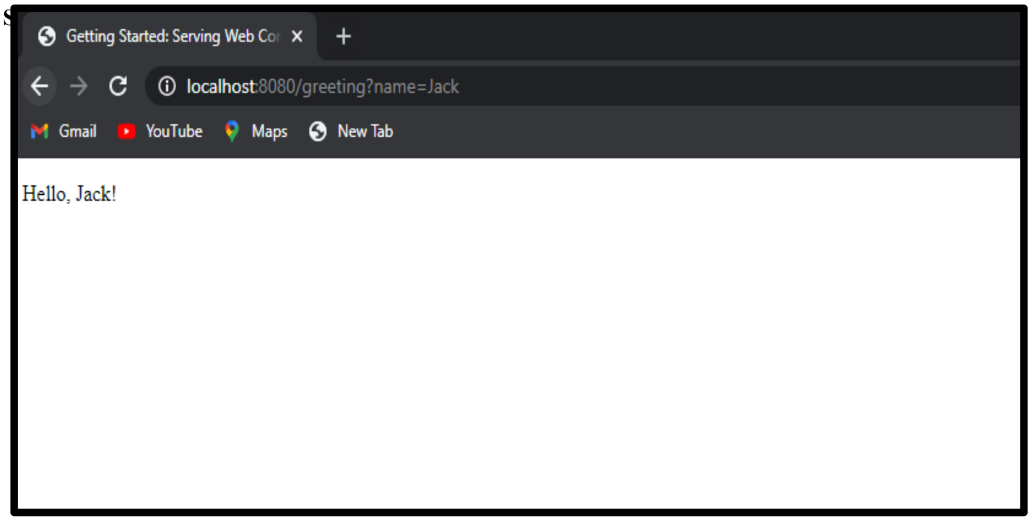


```
ML>
th="http://www.thymeleaf.org">
etting Started: Serving Web Content</title>
tp-equiv="Content-Type" content="text/html; charset=UTF-8" />

xt="'Hello, ' + ${name} + '!'" />
```

```
er (org.eclipse.lsp4e.LanguageServerWrapper@103dfbbc)
anguage server process ID: 19136
```

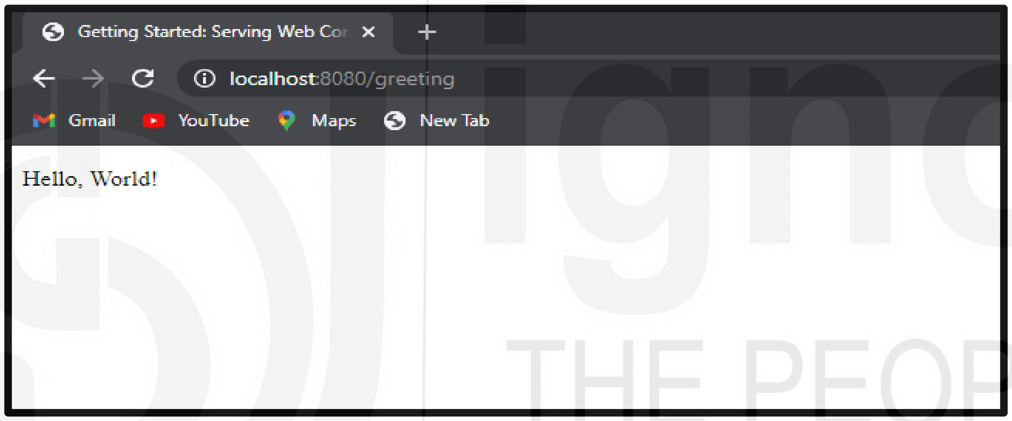
**Figure 9.6: SpringBoot Project Execution Into Eclipse**

following output in the browser.


Getting Started: Serving Web Cor
localhost:8080/greeting?name=Jack
Gmail
YouTube
Maps
New Tab

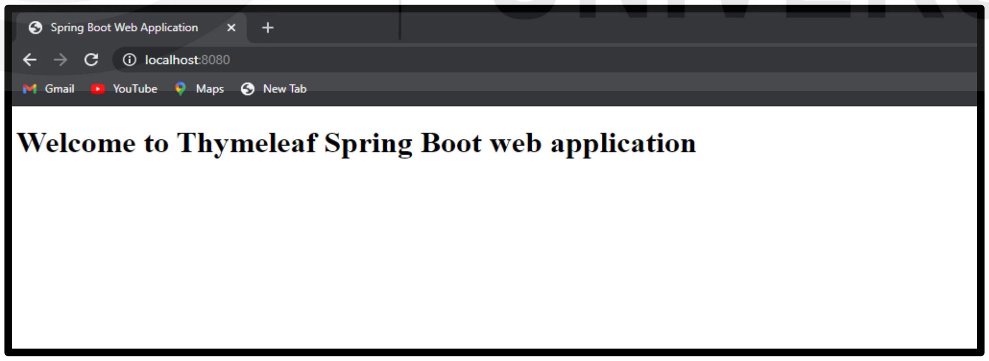
Hello, Jack!

Figure9.7: Execution Result 1



Hello, World!

Figure 8.9: Execution Result 2



Welcome to Thymeleaf Spring Boot web application

Figure 9.9: Execution Result 3

**✒Check Your Progress 1**

1) What is Spring Boot?Explain its need.

……………………………………………………………………………………
……………………………………………………………………………………
……………………………………………………………………………………
……………………………………………………………………………………
……………………………….

2) Describe Spring Boot working with @SpringBootApplication.

……………………………………………………………………………………
……………………………………………………………………………………
……………………………………………………………………………………
……………………………………………………………………………………
……………………………….

3) What are the Spring Boot starters, and what are the available starters?

……………………………………………………………………………………
……………………………………………………………………………………
……………………………………………………………………………………
……………………………………………………………………………………
……………………………….

4) What do you understand by auto-configuration in Spring Boot and how to disable the auto-configuration?

……………………………………………………………………………………
……………………………………………………………………………………
……………………………………………………………………………………
……………………………………………………………………………………
……………………………….

5) What is the need of a Spring Boot runner?

……………………………………………………………………………………
……………………………………………………………………………………
……………………………………………………………………………………
……………………………………………………………………………………
……………………………….

## 9.3 SPRING BOOT DEVTOOLS AND SPRING BOOT ACTUATOR

Spring Boot DevTools and Spring Boot Actuators are very important modules that enhance the development experience and increase productivity. These modules reduce the developers' effort and thus reducethe cost of the project.

### 9.3.1 Spring Boot DevTools

Spring Boot DevTools was released with Spring Boot 1.3. DevTools stands for developer tool. The aim of DevTools module is to enhance the application development experience by reducing the development time of the Spring Boot Application. During a web application development, a developer changes the code many times and then restarts the application to verify the changed code. DevToolsreduces the developer effort. It detects the code changes and restarts the application automatically. DevTools can be integrated into a Spring Boot application just by adding the following dependency into pom.xml.

```
<dependency>

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

       <artifactId>spring-boot-devtools</artifactId>

       <scope>runtime</scope>

       <optional>true</optional>

</dependency>
```

Spring Boot DevTools provides the following important features –

- Automatic Restart
- Live Reload
- Property defaults

**Automatic Restart:** Auto-restart refers to the reloading of Java classes and configures it at the server-side. While developing an application, a developer changes the code frequently and to verify these changes, the steps build, deploy and restart of the server is required. Spring Boot DevTools automates the build and deploys process, which increases productivity. Files change in the class-path triggers Spring Boot DevTools to restart the application. This auto-restart process reduces the time significantly to verify the changes. Spring Boot uses two types of ClassLoaders:

- **Base ClassLoader:** This ClassLoader loads the classes which do not change. E.g. Third-party jars
- **Restart ClassLoader:** This ClassLoader loads the classes which we are actively developing

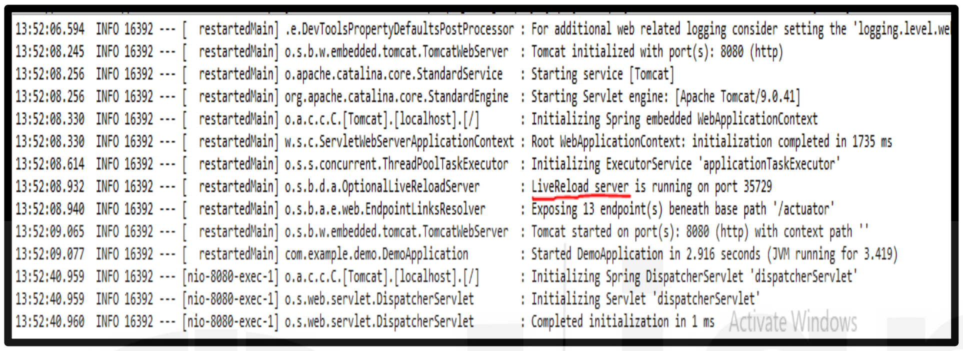
**Live Reload:** Live Reload or auto-refresh is also a very important feature provided by Spring Boot DevTools. Spring Boot DevTools module also comes with an embedded LiveReload server that can be used to trigger a browser refresh whenever a resource is changed. For example, when a developer makes the changes into templates or other resources, he/she has to refresh the browser to verify the changes. With the Live Reload/Auto Refresh feature, developers do not need to press F5 to refresh the browser. Thus, it enhances the development experience and increases productivity.

Enabling Live Reload in Spring Boot Application is pretty easy. Following steps are required to enable it.

1) Add the dependency spring-boot-devtools to your project's build file (pom.xml).

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-devtools</artifactId>
    <scope>runtime</scope>
    <optional>true</optional>
</dependency>
```

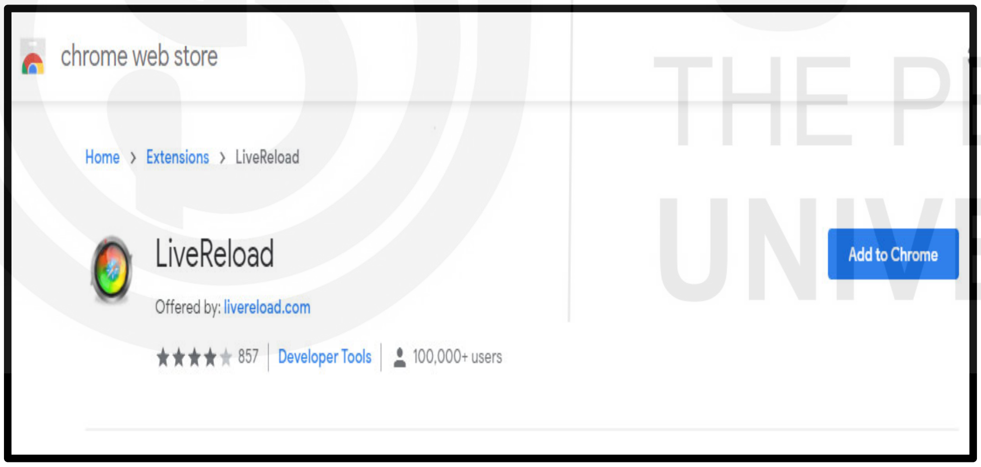
Once the application is started, one can verify the LiveReload server into log.



```
13:52:06.594  INFO 16392 --- [  restartedMain] .e.DevToolsPropertyDefaultsPostProcessor : For additional web related logging consider setting the 'logging.level.we
13:52:08.245  INFO 16392 --- [  restartedMain] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat initialized with port(s): 8080 (http)
13:52:08.256  INFO 16392 --- [  restartedMain] o.apache.catalina.core.StandardService   : Starting service [Tomcat]
13:52:08.256  INFO 16392 --- [  restartedMain] org.apache.catalina.core.StandardEngine  : Starting Servlet engine: [Apache Tomcat/9.0.41]
13:52:08.330  INFO 16392 --- [  restartedMain] o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing Spring embedded WebApplicationContext
13:52:08.330  INFO 16392 --- [  restartedMain] w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initialization completed in 1735 ms
13:52:08.614  INFO 16392 --- [  restartedMain] o.s.s.concurrent.ThreadPoolTaskExecutor  : Initializing ExecutorService 'applicationTaskExecutor'
13:52:08.932  INFO 16392 --- [  restartedMain] o.s.b.d.a.OptionalLiveReloadServer       : LiveReload server is running on port 35729
13:52:08.940  INFO 16392 --- [  restartedMain] o.s.b.a.e.web.EndpointLinksResolver      : Exposing 13 endpoint(s) beneath base path '/actuator'
13:52:09.065  INFO 16392 --- [  restartedMain] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat started on port(s): 8080 (http) with context path ''
13:52:09.077  INFO 16392 --- [  restartedMain] com.example.demo.DemoApplication         : Started DemoApplication in 2.916 seconds (JVM running for 3.419)
13:52:40.959  INFO 16392 --- [nio-8080-exec-1] o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing Spring DispatcherServlet 'dispatcherServlet'
13:52:40.959  INFO 16392 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet        : Initializing Servlet 'dispatcherServlet'
13:52:40.960  INFO 16392 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet        : Completed initialization in 1 ms
```

**Figure 9.10: Live Reload**

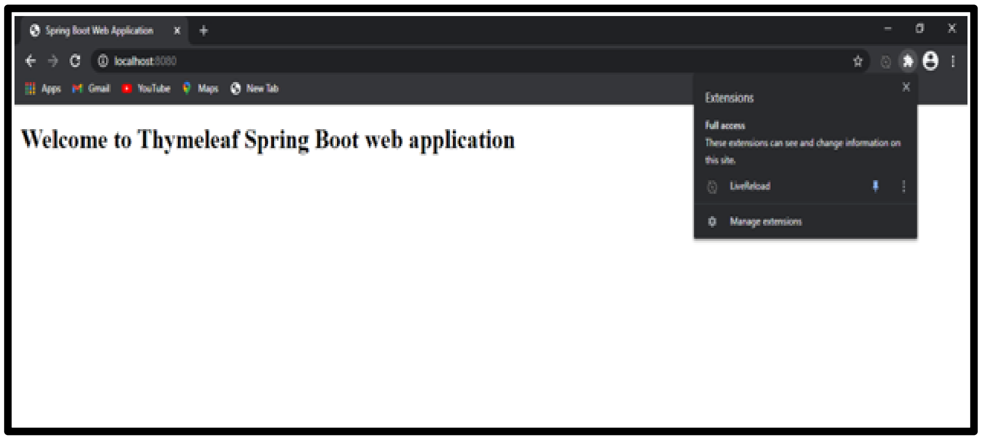
2) Install LiveReload extension for the browser. Go to http://livereload.com/extensions/ and click on the link that relates to your


chrome web store
Home › Extensions › LiveReload
LiveReload
Offered by: livereload.com
★★★★★ 857 | Developer Tools | 100,000+ users
Add to Chrome

browser.

**Figure 9.11: Live Reload Extension**

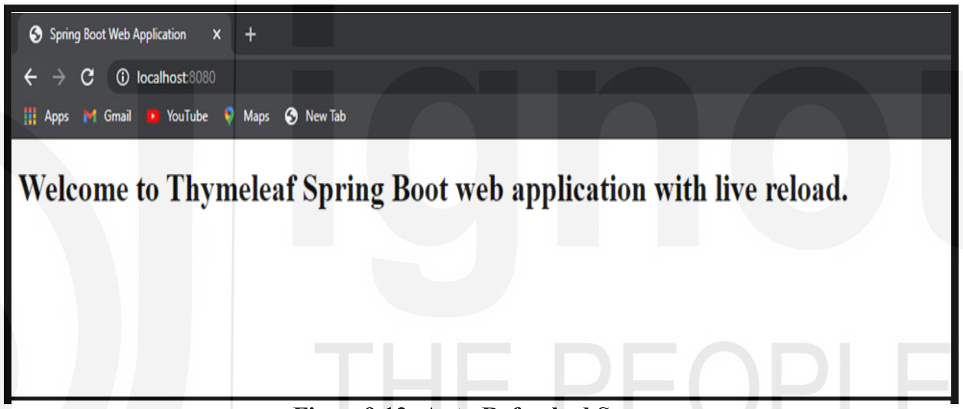
You can enable and disable the live reload by clicking on LiveReload as shown in screenshot.



Welcome to Thymeleaf Spring Boot web application

**Figure 9.12: Enable/Disable Live Relaod**

Update of home.html triggers the browser to auto-refresh. There is no need to press F5. Output is shown below:



Welcome to Thymeleaf Spring Boot web application with live reload.

**Figure9.13: Auto Refreshed Screen**

LiveReload works on the following path:

- /static
- /public
- /resources
- /templates
- /META-INF/maven
- /META-INF/resources

The following properties can be used to disable and enable the LiveReload feature.

```
# Disable LiveReload for following path

spring.devtools.restart.exclude=public/**, static/**, templates/**

# Enable LiveReload for additional path

spring.devtools.restart.additional-paths=/path-to-folder
```

**Property Defaults:** Spring Boot does a lot of auto configuration. It also includes caching for performance improvement. Template technology Thymeleaf contains the property **spring.thymeleaf.cache.** DevTools disables the caching and allows us to update pages without restarting the application. During the development, caching for Thymeleaf, Freemarker, Groovy Templates are automatically disabled by DevTools.

### 9.3.2 Spring Boot Actuator

Spring Boot Actuator is a sub-project of the Spring Boot framework. The Actuator provides production-ready features such as application monitoring, Network traffic, State of database and many more. Without any implementation, Actuator provides production-grade tools. In Spring Boot application, Actuator is primarily used to expose operational information about the running application such as info, health, dump, metrics, env etc. It provides **HTTP** and **JMX** endpoints to manage and monitor the Spring Boot application. There are three main features of Spring Boot Actuator-

- Endpoints
- Metrics
- Audit

**Endpoints:** The actuator endpoints enable us to monitor and interact with the application. In Spring Boot 2.x, most of the endpoints of the Actuator are disabled. By default, only two endpoints **/health and /info** are available. Other required endpoints can be enabled by adding ***management.endpoints.web.exposure.include*** property into application.properties. **By default, all Actuator endpoints are now placed under the */actuator* path.** Some of the important endpoints are listed below –

- **/health** provides the health status of the application
- **/info** provides general information about the application. It might be build information or the latest commit information.
- **/metrics** provides metrics of application. Returned metrics include generic metrics as well as a custom metric.
- **/env** provides the current environment properties.

**Metrics:** Spring Boot integrates the micrometer to provide dimensional metrics. The micrometer is the instrumentation library that empowers the delivery of application metrics from Spring. Metrics of the application can be accessed by **/metrics** endpoint.

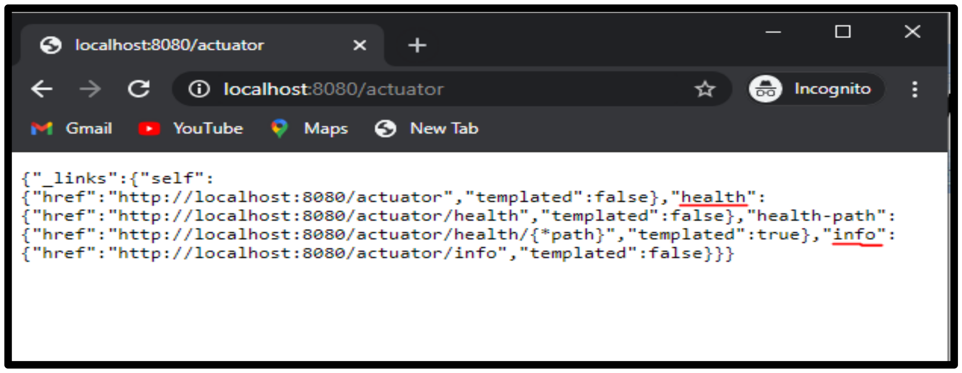
**Audit:** Spring Boot provides a flexible audit framework that publishes events to an AuditEventRepository. It automatically publishes the authentication events if spring-security is in execution.

### 9.3.3 Spring Boot Actuator Example

This section explains the required steps to integrate the Spring Boot Actuator into the previously developed website into section 9.2.5. Add the spring-boot-starter-actuator dependency into pom.xml –

```
<dependency>

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

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

</dependency>
```

Start the web application and access the **http://localhost:8080/actuator** endpoint. By default, only two end points are enabled. Result is as below:

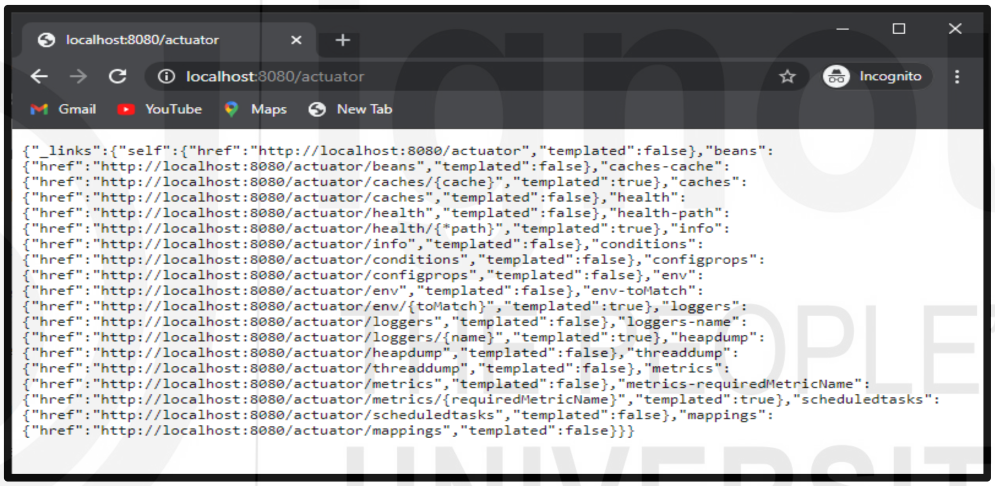


```
{"_links":{"self":
{"href":"http://localhost:8080/actuator","templated":false},"health":
{"href":"http://localhost:8080/actuator/health","templated":false},"health-path":
{"href":"http://localhost:8080/actuator/health/{*path}","templated":true},"info":
{"href":"http://localhost:8080/actuator/info","templated":false}}}
```

**Figure 9.14: Default Enabled Actuator Endpoints**

Add the following property in application.properties file and access the **http://localhost:8080/actuator** to get all available endpoints.

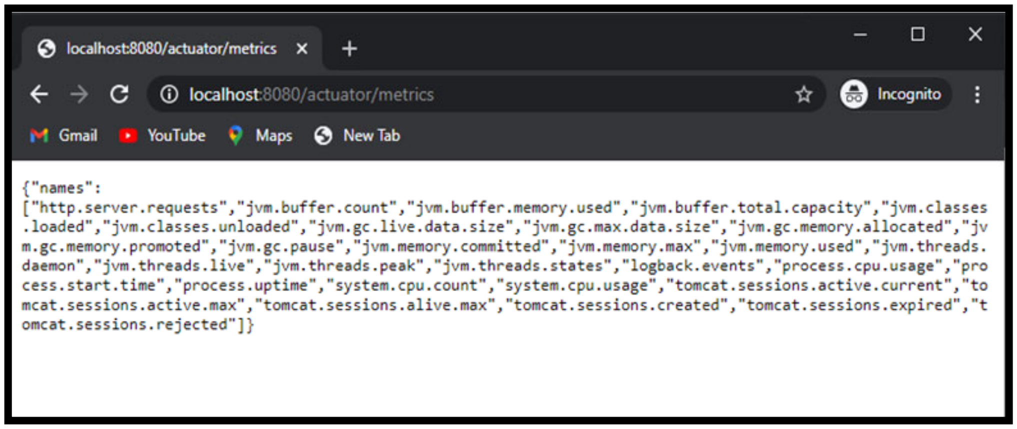
```
management.endpoints.web.exposure.include= *
```



```
{"_links":{"self":{"href":"http://localhost:8080/actuator","templated":false},"beans":
{"href":"http://localhost:8080/actuator/beans","templated":false},"caches-cache":
{"href":"http://localhost:8080/actuator/caches/{cache}","templated":true},"caches":
{"href":"http://localhost:8080/actuator/caches","templated":false},"health":
{"href":"http://localhost:8080/actuator/health","templated":false},"health-path":
{"href":"http://localhost:8080/actuator/health/{*path}","templated":true},"info":
{"href":"http://localhost:8080/actuator/info","templated":false},"conditions":
{"href":"http://localhost:8080/actuator/conditions","templated":false},"configprops":
{"href":"http://localhost:8080/actuator/configprops","templated":false},"env":
{"href":"http://localhost:8080/actuator/env","templated":false},"env-toMatch":
{"href":"http://localhost:8080/actuator/env/{toMatch}","templated":true},"loggers":
{"href":"http://localhost:8080/actuator/loggers","templated":false},"loggers-name":
{"href":"http://localhost:8080/actuator/loggers/{name}","templated":true},"heapdump":
{"href":"http://localhost:8080/actuator/heapdump","templated":false},"threaddump":
{"href":"http://localhost:8080/actuator/threaddump","templated":false},"metrics":
{"href":"http://localhost:8080/actuator/metrics","templated":false},"metrics-requiredMetricName":
{"href":"http://localhost:8080/actuator/metrics/{requiredMetricName}","templated":true},"scheduledtasks":
{"href":"http://localhost:8080/actuator/scheduledtasks","templated":false},"mappings":
{"href":"http://localhost:8080/actuator/mappings","templated":false}}}
```

**Figure 9.15: All Endpoints of Actuator**

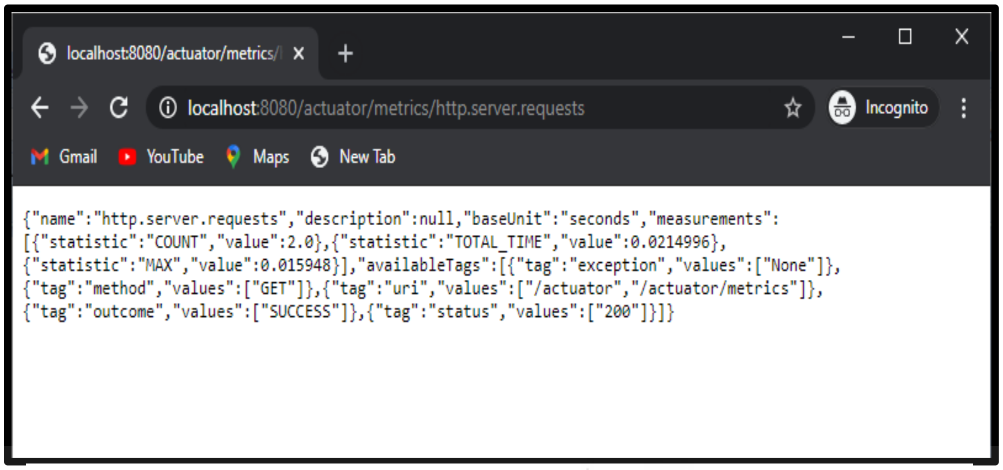
Output after adding above property is shown below:
Output for endpoint **http://localhost:8080/actuator/metrics** is shown below -



```
{"names":
["http.server.requests","jvm.buffer.count","jvm.buffer.memory.used","jvm.buffer.total.capacity","jvm.classes
.loaded","jvm.classes.unloaded","jvm.gc.live.data.size","jvm.gc.max.data.size","jvm.gc.memory.allocated","jv
m.gc.memory.promoted","jvm.gc.pause","jvm.memory.committed","jvm.memory.max","jvm.memory.used","jvm.threads.
daemon","jvm.threads.live","jvm.threads.peak","jvm.threads.states","logback.events","process.cpu.usage","pro
cess.start.time","process.uptime","system.cpu.count","system.cpu.usage","tomcat.sessions.active.current","to
mcat.sessions.active.max","tomcat.sessions.alive.max","tomcat.sessions.created","tomcat.sessions.expired","t
omcat.sessions.rejected"]}
```

**Figure 9.16: Autuator Metrics Result**

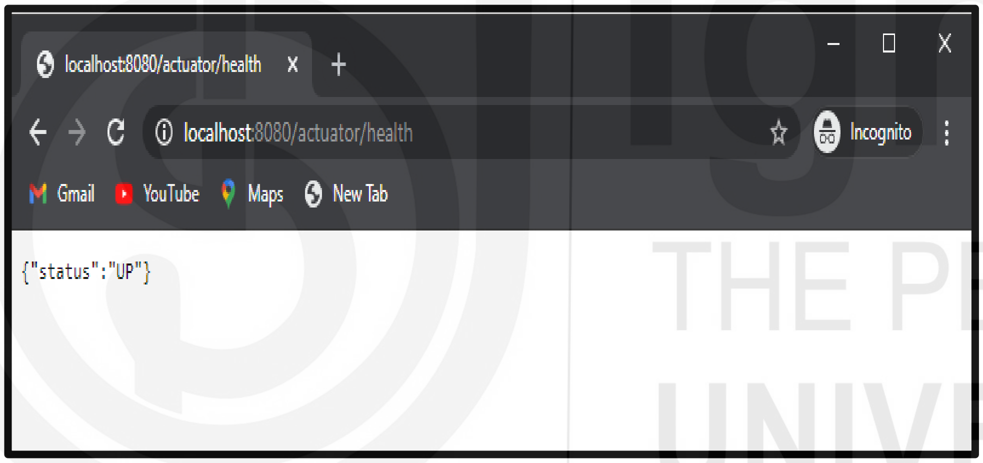
The above output shows all available metrics. Metric **http.server.requests**show the following output :


localhost:8080/actuator/metrics/
localhost:8080/actuator/metrics/http.server.requests
Incognito
Gmail
YouTube
Maps
New Tab

```
{"name":"http.server.requests","description":null,"baseUnit":"seconds","measurements":
[{"statistic":"COUNT","value":2.0},{"statistic":"TOTAL_TIME","value":0.0214996},
{"statistic":"MAX","value":0.015948}],"availableTags":[{"tag":"exception","values":["None"]},
{"tag":"method","values":["GET"]},{"tag":"uri","values":["/actuator","/actuator/metrics"]},
{"tag":"outcome","values":["SUCCESS"]},{"tag":"status","values":["200"]}]}
```

**Figure 9.17: Metric http.server.requests**

Output for **/health** endpoint is shown below-


localhost:8080/actuator/health
localhost:8080/actuator/health
Incognito
Gmail
YouTube
Maps
New Tab

```
{"status":"UP"}
```

**Figure9.18: Health Status of Server**

**✒Check Your Progress 2**

1) What is the need of Spring Boot DevTools?

…………………………………………………………………………………

…………………………………………………………………………………

…………………………………………………………………………………

…………………………………………………………………………………

………………………………

2) Explain Spring Boot Actuator and its advantages.

…………………………………………………………………………………

…………………………………………………………………………………

…………………………………………………………………………………

…………………………………………………………………………………

……………………………….

3) Write a simple "Hello World" Spring Boot rest application.

…………………………………………………………………………………

…………………………………………………………………………………

…………………………………………………………………………………

…………………………………………………………………………………

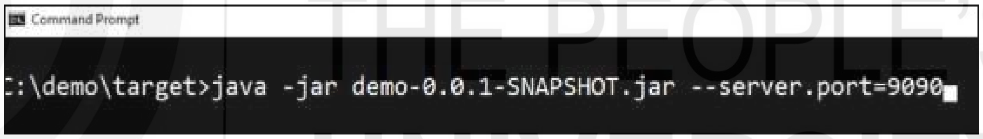
……………………………….

## 9.4 SPRING BOOT- APPLICATION PROPERTIES

Application Properties enable us to work in different environments such as Prod, Dev, Test. This section explains how to set up and use properties in Spring Boot via Java configuration and @PropertySource. Spring Boot application properties can be set into properties files, YAML files, command-line arguments and environment variables.

### 9.4.1 Command Line Properties

Spring Boot Environment properties can be passed via command-line properties. Command-line properties take precedence over the other property sources. The following screenshot shows the command line property ***–server.port=9090*** to change the port number



```
:\demo\target>java -jar demo-0.0.1-SNAPSHOT.jar --server.port=9090
```

***Note:*** Double hyphen (--) can be used as a delimiter to pass multiple command-line properties.

### 9.4.2 Properties File

By default, Spring Boot can access configurations available in an **application.properties** file kept under the classpath. The application.properties file should be kept in **src/main/resources** directory. Thesample application.properties file is shown below-

```
server.port = 9090

spring.application.name = demoservice
```

Property server.port changes the port number on which the web application runs.

### 9.4.3 YAML File

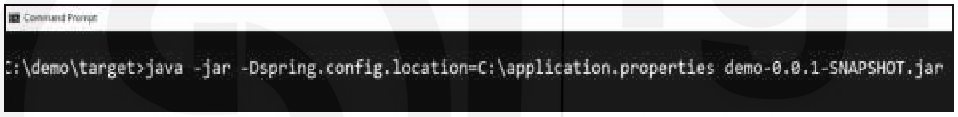
Spring Boot also supports YAML based properties configuration. Properties file application.yml can be used instead of application.properties. **YAML is a convenient format for specifying hierarchical configuration data**. The application.yml file also should be kept in **src/main/resources** directory. Sample application.yml file is shown below-

```
server:
port: 9090
spring:
application:
name: demoservice
```

### 9.4.4 Externalized Properties

Spring Boot supports keeping properties file at any location. We can externalize the properties file so that if any property is changed, a new build is not required. Just application restart will take effect on the changed property. While executing the application jar file, the following command-line argument is used to specify the location of the property file.

```
-Dspring.config.location = C:\application.properties
```



```
C:\demo\target>java -jar -Dspring.config.location=C:\application.properties demo-0.0.1-SNAPSHOT.jar
```

### 9.4.5 @Value annotation

Properties, defined for environment or application, can be fetched in java code using **@Value** annotation. Syntax to use @Value annotation is shown below –

```
@Value("${property_key}")
```

If a property **spring.application.name** is defined into application.properties or application.yml, it can be accessed into java code using as-

```
@Value("${spring.application.name}")

String appName;
```

While running, if the property is not found, an exception is thrown. The default value can be set while fetching the value using @Value annotation as follows-

```
@Value("${property_key:default_value}")
```

### 9.4.6 Active Profile

An application is executed in multiple environments such as test, development, production etc. To modify application.properties file based on the environment is not an ideal approach. There must be multiple properties files corresponding to each environment. At run time, Spring Boot must be able to select the desired environment properties file.

Spring Boot supports different properties based on the Spring active profile. Consider that there are separate properties file for each environment as shown below:

**application.properties**

```
server.port = 9090

spring.application.name = demoservice
```

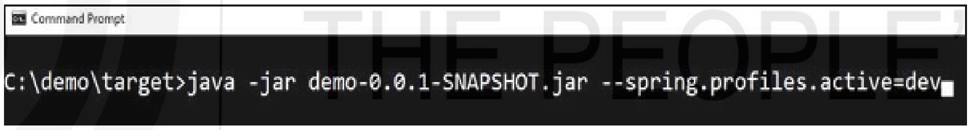
**application-dev.properties**

```
server.port = 9090

spring.application.name = demoservice
```

**application-prod.properties**

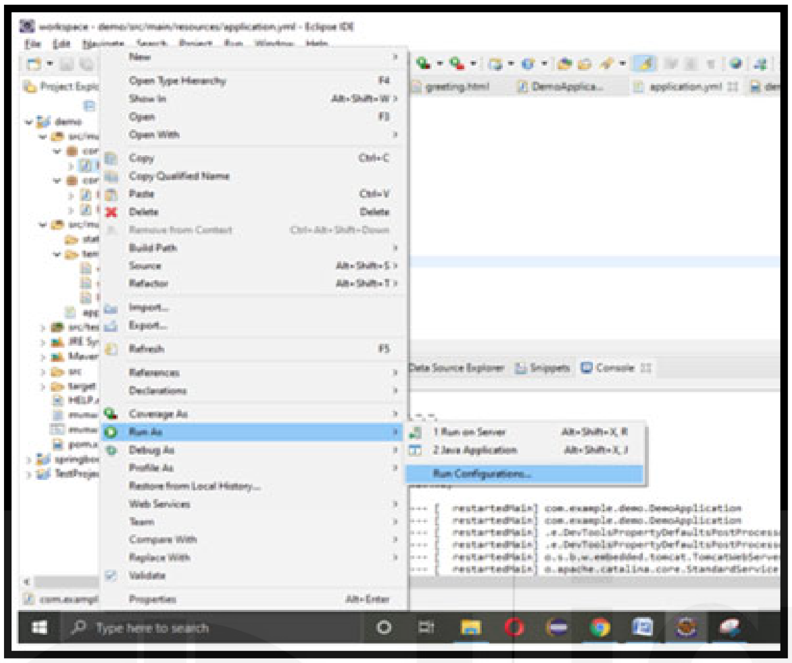
```
server.port = 9090

spring.application.name = demoservice
```

By default, Spring Boot uses application.properties file. But Spring Boot allows to set the active profile and corresponding properties file that is used by Spring Boot. The following command shows how to set an active profile while starting an application from the command- line.



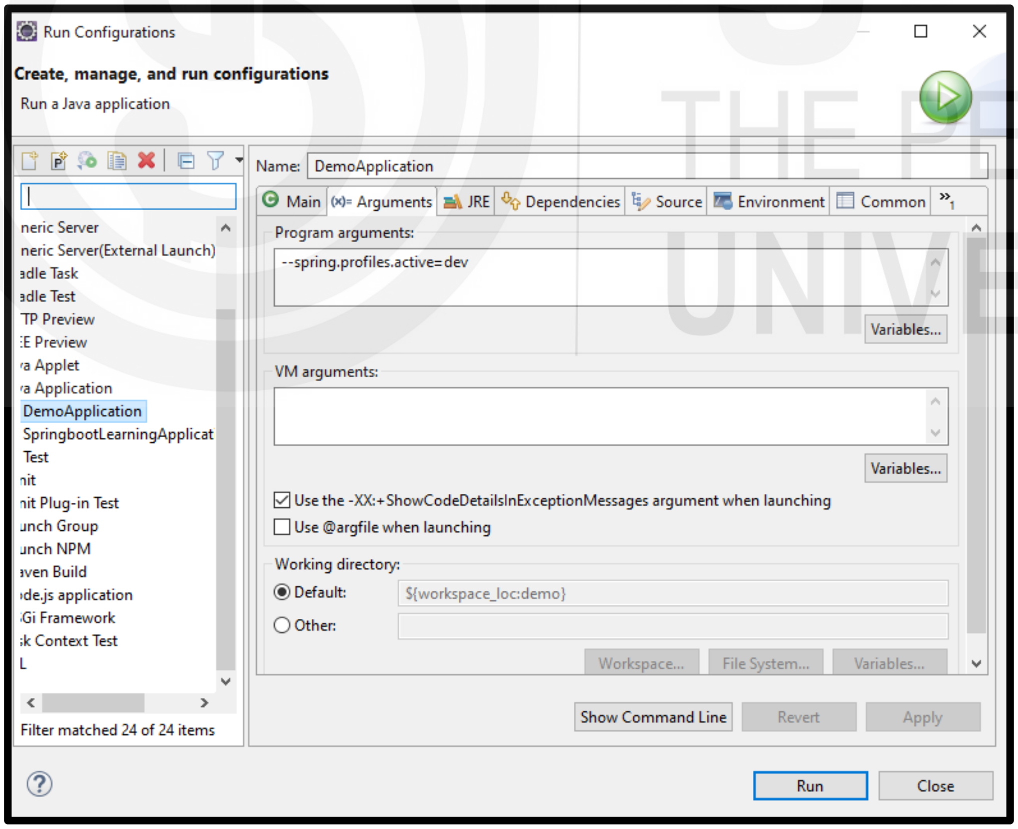
```
C:\demo\target>java -jar demo-0.0.1-SNAPSHOT.jar --spring.profiles.active=dev
```

While running an application from eclipse IDE, the active profile can be set as shown in the screenshot.



```
restartedMain] com.example.demo.DemoApplication
restartedMain] com.example.demo.DemoApplication
restartedMain] .e.DevToolsPropertyDefaultsPostProcess
restartedMain] .e.DevToolsPropertyDefaultsPostProcess
restartedMain] o.s.b.w.embedded.tomcat.TomcatWebServe
restartedMain] o.apache.catalina.core.StandardService
```

Figure 9.19: Set Active Profile in Eclipse Project 1



Name: DemoApplication

Program arguments:

```
--spring.profiles.active=dev
```

VM arguments:

[✓] Use the -XX:+ShowCodeDetailsInExceptionMessages argument when launching

[ ] Use @argfile when launching

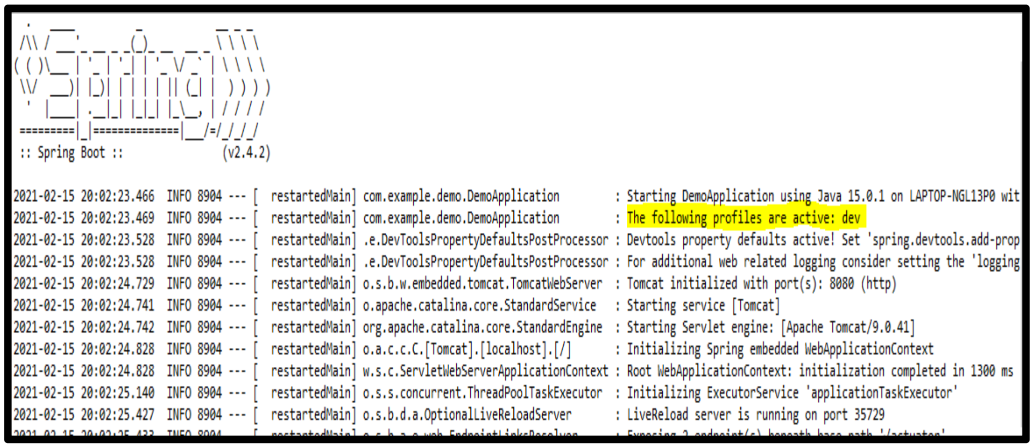
Working directory:

[✓ Default:] ${workspace_loc:demo}

Other:

Figure9.20: Set Active Profile in Eclipse Project 2

Active profile can be checked in the log while starting the application.



```
  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::                (v2.4.2)

2021-02-15 20:02:23.466  INFO 8904 --- [  restartedMain] com.example.demo.DemoApplication         : Starting DemoApplication using Java 15.0.1 on LAPTOP-NGL13P0 wit
2021-02-15 20:02:23.469  INFO 8904 --- [  restartedMain] com.example.demo.DemoApplication         : The following profiles are active: dev
2021-02-15 20:02:23.528  INFO 8904 --- [  restartedMain] .e.DevToolsPropertyDefaultsPostProcessor : Devtools property defaults active! Set 'spring.devtools.add-prop
2021-02-15 20:02:23.528  INFO 8904 --- [  restartedMain] .e.DevToolsPropertyDefaultsPostProcessor : For additional web related logging consider setting the 'logging
2021-02-15 20:02:24.729  INFO 8904 --- [  restartedMain] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat initialized with port(s): 8080 (http)
2021-02-15 20:02:24.741  INFO 8904 --- [  restartedMain] o.apache.catalina.core.StandardService   : Starting service [Tomcat]
2021-02-15 20:02:24.742  INFO 8904 --- [  restartedMain] org.apache.catalina.core.StandardEngine  : Starting Servlet engine: [Apache Tomcat/9.0.41]
2021-02-15 20:02:24.828  INFO 8904 --- [  restartedMain] o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing Spring embedded WebApplicationContext
2021-02-15 20:02:24.828  INFO 8904 --- [  restartedMain] w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initialization completed in 1300 ms
2021-02-15 20:02:25.140  INFO 8904 --- [  restartedMain] o.s.s.concurrent.ThreadPoolTaskExecutor  : Initializing ExecutorService 'applicationTaskExecutor'
2021-02-15 20:02:25.427  INFO 8904 --- [  restartedMain] o.s.b.d.a.OptionalLiveReloadServer       : LiveReload server is running on port 35729
2021-02-15 20:02:25.433  INFO 8904 --- [  restartedMain] o.s.b.a.e.web.EndpointLinksResolver      : Exposing 2 endpoint(s) beneath base path '/actuator'
```

**Figure 9.21: SpringBoot Application Execution Log**

### 9.4.7 Spring Active Profile in application.yml

YAML file allows active profile parameters to be kept into a single application.yml file. Unlike multiple application.properties files, there is a single application.yml. Delimiter (---) is used to separate each profile in an application.yml file. Sample application.yml file is shown with active profile dev and prod.

```
server:
port: 9090
spring:
application:
name: demoservice

---
server:
port: 8080
spring:
config:
activate:
on-profile: dev
application:
name: demoservice
---
server:
port: 8080
spring:
config:
activate:
on-profile: prod
application:
name: demoservice
```

## 9.5 RUNNING SPRING BOOT APPS FROM COMMAND LINE

This section describes a couple of ways to run a Spring Boot app from a command line in a terminal window. Later it explains how to package the app as war and which can be deployed on any application server such as Apache Tomcat, WebLogic, JBoss, etc.

The **Spring Boot Maven** plugin is the recommended tool to build, test and package the Spring Boot Application code. The plugin is configured by adding it into pom.xml.

```
<build>

<plugins>

    ...

<plugin>

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

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

</plugin>

    ...

</plugins>

</build>
```

The plugin comes with lots of convenient features such as –

- It can package all our dependencies (including an embedded application server if needed) in a single, runnable fat jar/war
- It resolves the correct dependency versions

### 9.5.1 Running the Code with Maven in Exploded Form

The Spring Boot Maven plugin has the ability to automatically deploy the web application in an embedded application server. If **spring-boot-starter-web** dependency has been included, plugin knows that Tomcat is required to run the code. On execution of **mvnspring-boot:run** command in the project root folder, the plugin reads the pom configuration. If a web application container is required, the plugin triggers the download of Apache Tomcat and initializes the startup of Tomcat. The command to run the Spring Boot application using maven plugin is as following –

```
mvnspring-boot:run
```

Execution of the above command starts the Spring Boot application and the produced log is shown.

## 9.5.2 Running the Code as a Stand-Alone Packaged Application

Once the development phase is over, the application is moved to production, and we need to package the application. Just include the **Spring Boot Maven** plugin and execute the following command in order to package the application.

```
mvn clean package spring-boot:repackage
```

The execution of the above command packages the application and produces the jar

**Figure 9.22: SpringBoot Application Execution From Command Line**

file into the target folder. The generated jar file can be executed using the following command.

```
java –jar <File Name>
```

As you can notice, that –cp option and main class has been skipped into java –jar command because the Spring Boot Maven plugin takes care of all these configurations into the manifest file.

## 9.5.3 Application Packaging as WAR and Deployment on Tomcat

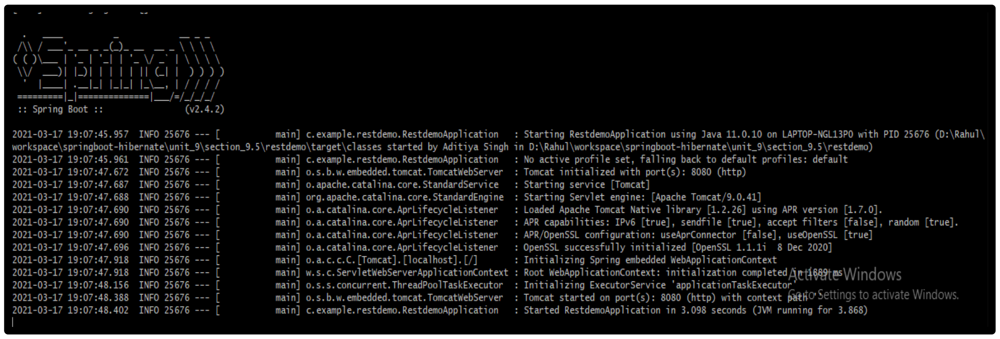
By default, Spring Boot builds a standalone Java application that can run as a desktop application. The standalone Java application is not suitable for the environment where installation of a new service or manual execution of an application is not allowed, such as Production environment.

The Servlet containers require the applications to meet some contracts to be deployed. For Tomcat the contract is the Servlet API 3.0. This section considers the example used in section 9.2.5 and explains how the application can be packaged as WAR and deployed into Tomcat.

First, change the packaging type as war into pom.xml with the following content.

```
<packaging>war</packaging>
```

Initialize the Servlet context required by Tomcat by extending



```
  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::                (v2.4.2)

2021-03-17 19:07:45.957  INFO 25676 --- [           main] c.example.restdemo.RestdemoApplication   : Starting RestdemoApplication using Java 11.0.10 on LAPTOP-NGL13PO with PID 25676 (D:\Rahul\
workspace\springboot-hibernate\unit_9\section_9.5\restdemo\target\classes started by Aditiya Singh in D:\Rahul\workspace\springboot-hibernate\unit_9\section_9.5\restdemo)
2021-03-17 19:07:45.961  INFO 25676 --- [           main] c.example.restdemo.RestdemoApplication   : No active profile set, falling back to default profiles: default
2021-03-17 19:07:47.672  INFO 25676 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat initialized with port(s): 8080 (http)
2021-03-17 19:07:47.687  INFO 25676 --- [           main] o.apache.catalina.core.StandardService   : Starting service [Tomcat]
2021-03-17 19:07:47.688  INFO 25676 --- [           main] org.apache.catalina.core.StandardEngine  : Starting Servlet engine: [Apache Tomcat/9.0.41]
2021-03-17 19:07:47.690  INFO 25676 --- [           main] o.a.catalina.core.AprLifecycleListener   : Loaded Apache Tomcat Native library [1.2.26] using APR version [1.7.0].
2021-03-17 19:07:47.690  INFO 25676 --- [           main] o.a.catalina.core.AprLifecycleListener   : APR capabilities: IPv6 [true], sendfile [true], accept filters [false], random [true].
2021-03-17 19:07:47.690  INFO 25676 --- [           main] o.a.catalina.core.AprLifecycleListener   : APR/OpenSSL configuration: useAprConnector [false], useOpenSSL [true]
2021-03-17 19:07:47.696  INFO 25676 --- [           main] o.a.catalina.core.AprLifecycleListener   : OpenSSL successfully initialized [OpenSSL 1.1.1i  8 Dec 2020]
2021-03-17 19:07:47.918  INFO 25676 --- [           main] o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing Spring embedded WebApplicationContext
2021-03-17 19:07:47.918  INFO 25676 --- [           main] w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initialization completed in 1889 ms
2021-03-17 19:07:48.156  INFO 25676 --- [           main] o.s.s.concurrent.ThreadPoolTaskExecutor  : Initializing ExecutorService 'applicationTaskExecutor'
2021-03-17 19:07:48.388  INFO 25676 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat started on port(s): 8080 (http) with context path ''
2021-03-17 19:07:48.402  INFO 25676 --- [           main] c.example.restdemo.RestdemoApplication   : Started RestdemoApplication in 3.098 seconds (JVM running for 3.868)
```

the *SpringBootServletInitializer* class:

```
@SpringBootApplication

publicclassDemoApplicationextendsSpringBootServletInitializer

{
	publicstaticvoid main(String[] args)
	{
		SpringApplication.run(DemoApplication.class, args);
	}
}
```

By default, generated war file name includes version number also. The Name of the generated war can be modified with following –

```
<build>

	<finalName>${artifactId}</finalName>

	...

</build>
```

Execute the following command in order to build the Tomcat deployable war if artifact id is **demo,** war file generated at target/demo.war. Follow the below steps in order to deploy the generated war file on Tomcat.

- Download Apache Tomcat and unpack it into the tomcat folder.
- Copy the generated war file from target/demo.war to tomcat/webapps/demo.war
- Go to bin dir of Tomcat and start the tomcat using ***catalina.bat start*** (on windows) or ***catalina.sh start*** (on Unix)
- Access the application using http://localhost:8080/demo

## Check Your Progress 3

1) Mention the possible sources of external configuration.

……………………………………………………………………………………
……………………………………………………………………………………
……………………………………………………………………………………
……………………………………………………………………………………
………………………………

2) Can we change the port of the embedded Tomcat server in Spring boot?

……………………………………………………………………………………
……………………………………………………………………………………
……………………………………………………………………………………
……………………………………………………………………………………
………………………………

3) Explain the concept of profile in Spring Boot and say how it is useful,

…………………………………………………………………………………
…………………………………………………………………………………
…………………………………………………………………………………
…………………………………………………………………………………
……………………………….

4) Explain the Spring Boot application execution with Maven.

…………………………………………………………………………………
…………………………………………………………………………………
…………………………………………………………………………………
…………………………………………………………………………………
……………………………….

5) What are the steps to deploy Spring Boot web applications as JAR and WAR files?

…………………………………………………………………………………
…………………………………………………………………………………
…………………………………………………………………………………
…………………………………………………………………………………
……………………………….

## 9.6 SUMMARY

This unit has described Spring Boot facilitates the developers to create a Spring based web application with minimum lines of code since it provides auto configuration **out-of-the-box**. This unit has explained the following:

- A class with the main method and annotated with **@SpringBootApplication** is the entry point of the Spring Boot application.
- Spring Boot starters are the dependency descriptor that addresses the problem of compatible versions of dependency management. It is also a one-stop-shop for all the Spring and related technology that you need.
- Spring Boot provides two runner interfaces named as ***ApplicationRunner*** and ***CommandLineRunner***. These functional interfaces enable you to execute a piece of code just after the Spring Boot application is started.
- Spring Boot ***DevTools*** important features such as Automatic restart, Live Reload, Property Defaults such as cache disable and allows us to update pages without restarting the application
- ***Actuator*** provides production-ready features such as application monitoring, Network traffic, State of database and many more. Without any implementation, Actuator provides production-grade tools.
- By default, only two endpoints ***/health*** and ***/info*** are available. Other required endpoints can be enabled by adding ***management.endpoints.web.exposure.include*** property into application.properties.
- Various ways to define application properties such as command-line argument, application.properties file and application.yml file.

- Spring Boot supports keeping properties files at any location. We can externalize the properties file so that if any property is changed, a new build is not required.
- Spring Boot supports the concept of an active profile. There can be multiple properties files corresponding to each environment. At run time, Spring Boot selects the desired environment properties file based **on active profile**.
- Spring Boot application can be executed from command line with Spring Boot Maven plugin using **mvnspring-boot:run** and can be executed as a standalone application by **java -jar <fileName>**

## 9.7 SOLUTIONS/ANSWER TO CHECK YOUR PROGRESS

**Check Your Progress 1**

1) ***Spring Boot*** is a utility project which enables an organization to develop production-ready spring-based applications and services with less effort, reduced cost and minimal configuration. ***Spring Boot***facilitates the developers to create a Spring based web application with minimum lines of code since it provides auto configuration out-of-the-box.

   It is a module that enriches the Spring framework with Rapid Application Development (RAD) feature. It provides an easy way to create a stand-alone and production ready spring application with minimum configurations. Spring Boot is a combination of Spring framework with auto-configuration and embedded Servers.

   Spring Boot provides a vast number of features and benefits. A few of them are as follows:

   - Everything is auto-configured in Spring Boot.
   - Spring Boot starter eases the dependency management and application configuration
   - It simplifies the application deployment by using an embedded server
   - Production-ready features to monitor and manage applications such as health checks, metrics gathering etc.
   - Reduces the application development time and run the application independently
   - Very easy to understand and develop Spring application

2) Spring Boot auto configures all required configurations. It performs auto-configuration by scanning the classes in classpath annotated with @Component or @Configuration. @SpringBootApplication annotation comprises the following three annotations with their default values-
   - @EnableAutoConfiguration
   - @ComponentScan
   - @Configuration

   For details, check section 9.2.1

3) Spring Boot starters are the dependency descriptor that addresses the problem of compatible versions of dependency management. Starter POMs are a set of convenient dependency descriptors that you can include in your application.

You get a one-stop-shop for all the Spring and related technology that you need from Spring Boot starters. Spring Boot provides a number of starters that make development easier and rapid. Spring Boot provides many numbers starter, and a few of them are listed below-

| | |
|---|---|
| **spring-boot-starter-web** | It is used for building web applications, including RESTful applications using Spring MVC. It uses Tomcat as the default embedded container. |
| **spring-boot-starter-jdbc** | It is used for JDBC with the Tomcat JDBC connection pool. |
| **spring-boot-starter-validation** | It is used for Java Bean Validation with Hibernate Validator. |
| **spring-boot-starter-security** | It is used for Spring Security. |
| **spring-boot-starter-data-jpa** | It is used for Spring Data JPA with Hibernate. |

4) As the name implies, it is used to automatically configure the required configuration for the application. It means **Spring Boot** looks for auto-configuration beans on its classpath and automatically applies them. For example, if H2 dependency is added and you have not manually configured any database connection beans, then Spring will auto-configure H2 as an in-memory database.
   To disable the auto-configuration property, you have to use exclude attribute of @EnableAutoConfiguration. For example, Data Source autoconfiguration can be disabled as:
   **@EnableAutoConfiguration(exclude={DataSourceAutoConfiguration.class})**
   The property ***spring.autoconfigure.exclude*** property can be used in application.properties or application .yaml file to mention the exclude list of auto-configuration classes.

5) Spring Boot provides two runner interfaces named as ApplicationRunner and CommandLineRunner. These interfaces enable you to execute a piece of code just after the Spring Boot application is started.
   Both interfaces are Functional Interface. If any piece of code needs to be executed when Spring Boot Application starts, we can implement either of these functional interfaces and override the single method orun.
   **Check the details of ApplicationRunner and CommandLineRunner in section 9.2.4**

## ☛Check Your Progress 2

1) Spring Boot DevTools was released with Spring Boot 1.3. DevToolsstands for developer tool. Aim of DevTools module is to enhance the application development experience by improving the development time of the Spring Boot Application. During a web application development, a developer changes the code many times and then restarts the application to verify the code changes. DevTools reduces the developer effort. It detects the code changes and restarts the application. DevTools can be integrated into a Spring Boot application just by adding the following dependency into pom.xml.

```
<dependency>

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

        <artifactId>spring-boot-devtools</artifactId>

        <scope>runtime</scope>

        <optional>true</optional>

</dependency>
```

Spring Boot DevTools provides the following important features –

- Automatic Restart
- Live Reload
- Property defaults

2) Spring Boot Actuator is a sub-projectof the Spring Boot framework. The Actuator provides production-ready features such as application monitoring, Network traffic, State of database and many more. Without any implementation, Actuator provides production-grade tools. In Spring Boot application, Actuator is primarily used to expose operational information about the running application such as info, health, dump, metrics, env etc. It provides **HTTP** and **JMX** endpoints to manage and monitor the Spring Boot application. There are three main features of the Spring Boot Actuator-
   - Endpoints
   - Metrics
   - Audit

   By default, only two endpoints **/health and /info** are available. Other required endpoints can be enabled by adding ***management.endpoints.web.exposure.include*** property into application.properties. **By default, all Actuator endpoints are now placed under the *actuator* path.** Some of the important endpoints are listed below –

   - **/health** provides the health status of application
   - **/info** provides general information about the application. It might be build information or the latest commit information.
   - **/metrics** provides metrics of application. Returned metrics include generic metrics as well as custom metric.
   - **/env** provides the current environment properties.

3) To create a simple Hello World rest application using Spring Boot, we need to perform the below steps:

   - Create a Spring Boot Project using Spring Initializr. Visit at https://start.spring.io/ and generate the Spring Boot project with **Spring Web** dependency.
   - Import the project in Eclipse IDE as a maven **project**.
   - Create a controller package and add a **HomeController** class annotated with **@RestController**.

```
@RestController

publicclassHomeController {

       @GetMapping(value="/")

       public String index() {

              return"Hello World!!";

       }

}
```

- From the root of the project, execute command **mvnspring-boot:run** in order to run the application.
- Access the application with **http://localhost:8080**

## ✒Check Your Progress 3

1) Application Properties enable us to work in different environments such as Prod, Dev, Test. Spring Boot application properties can be set into properties files, YAML files, command-line arguments and environment variables.
   Spring Boot supports keeping properties file at any location. We can externalize the properties file so that if any property is changed, a new build is not required. Just application restart will take effect on the changed property. While executing the application jar file, the following command line argument is used to specify the location of the property file.

   ```
   -Dspring.config.location = C:\application.properties
   ```

2) Yes, we can change the port of the embedded tomcat server by using the application properties file. In application.properties file, you must add a property of "**server.port**" and assign it to any port you wish to. For example, if you want to assign it to 8081, then you have to mention **server.port=8081**. Once you mention the port number, the application properties file will be automatically loaded by Spring Boot, and the required configurations will be applied to the application.

3) An application is executed in multiple environments such as test, development, production. To modify application.properties file based on the environment is not an ideal approach. There must be multiple properties files corresponding to each environment. At run time Spring Boot must be able to select the desired environment properties file.
   Spring Boot supports different properties based on the Spring active profile. By default, Spring Boot uses application.properties file. But Spring Boot allows to set the active profile and corresponding properties file that is used by Spring Boot. Following command shows to set **dev** as active profile while starting the application from command line, and the execution will read the application-dev.properties file since active profile is set as **dev**.
   **java –jar demo.jar --spring.profiles.active=dev**

4) Check section 9.5.1
5) Check section 9.5.3

## 9.8 REFERENCES/FURTHER READING

● Craig Walls, "Spring Boot in action" Manning Publications, 2016. https://doc.lagout.org/programmation/Spring%20Boot%20in%20Action.pdf)

● Christian Bauer, Gavin King, and Gary Gregory, "**J**ava Persistence with Hibernate",Manning Publications, 2015.

● Ethan Marcotte, "Responsive Web Design", Jeffrey Zeldman Publication, 2011(http://nadin.miem.edu.ru/images_2015/responsive-web-design-2nd-edition.pdf)

● Tomcy John, "Hands-On Spring Security 5 for Reactive Applications",Packt Publishing,2018

… https://spring.io/guides/gs/spring-boot/

… https://dzone.com/articles/introducing-spring-boot

… https://www.baeldung.com/spring-boot

… https://docs.spring.io/spring-boot/docs/1.5.4.RELEASE/reference/pdf/spring-boot-reference.pdf

… https://www.baeldung.com/spring-boot-starters

… https://doc.lagout.org/programmation/Spring%20Boot%20in%20Action.pdf

… https://www.baeldung.com/spring-boot-devtools

… https://docs.spring.io/spring-boot/docs/1.5.16.RELEASE/reference/html/using-boot-devtools.html

… https://howtodoinjava.com/spring-boot/developer-tools-module-tutorial/

… https://www.baeldung.com/spring-boot-run-maven-vs-executable-jar

# UNIT 10 CONFIGURATION OF HIBERNATE(ORM)

**Structure** **Page No.**

10.0 Introduction
10.1 Objectives
10.2 Hibernate Overview
    10.3 Hibernate Configuration with Annotation
    10.3.1 A Complete Hibernate Example
10.4 Hibernate CRUD (Create, Read, Update, and Delete) Features
    10.4.1 Hibernate Get Entity – get vs load
    10.4.2 Hibernate Insert Entity – persist, save and saveOrUpdate
    10.4.3 Hibernate Query Language (HQL)
10.5 Spring Data JPA Overview
10.6 Summary
10.7 Solutions/ Answer to Check Your Progress
10.8 References/Further Reading

## 10.0 INTRODUCTION

In computer science, object-relational mapping (ORM, O/RM, and O/R mapping tool) is a programming technique for converting data between incompatible type systems using object-oriented programming languages. ORM makes it possible to perform ***CRUD (Create, Read, Update and Delete)*** operations without considering how those objects relate to their data source. It manages the mapping details between a set of objects and the underlying database. It hides and encapsulates the changes in the data source itself. Thus, when data sources or their APIs change, only ORM change is required rather than the application that uses ORM.

Hibernate is a pure Java object-relational mapping (ORM) and persistence framework which maps the POJOs to relational database tables. The usage of Hibernate as a persistence framework enables the developers to concentrate more on the business logic instead of making efforts on SQLs and writing boilerplate code. There are several advantages of Hibernate, such as:

- Open Source
- High Performance
- Light Weight
- Database independent
- Caching
- Scalability
- Lazy loading
- Hibernate Query Language (HQL)

Java Persistence API (JPA) is a specification that defines APIs for object-relational mappings and management of persistent objects. JPA is a set of interfaces that can be used to implement the persistence layer. JPA itself doesn't provide any implementation classes. In order to use the JPA, a provider is required which implements the specifications. Hibernate and EclipseLink are popular JPA providers.

**Spring Data JPA** is one of the many sub-projects of **Spring Data**that simplifies the data access to relational data stores. Spring Data JPA is not a JPA provider; instead, it wraps the JPA provider and adds its own features like a **no-code implementation of the repository pattern**. Spring Data JPA uses Hibernate as the **default JPA provider**. JPA provider is configurable, and other providers can also be used with Spring Data JPA. Spring Data JPA provides a complete abstraction over the DAO layer into the project.

This unit explains only Hibernate in detail with its architecture and sample examples. This unit also covers the relationship between Spring Data JPA, JPA and Hibernate. There are many similar Crud operations available in Hibernate, such as **save, persist, saveOrUpdate, get and load**. Hibernate entity state defines the behavior of the operations. Thus, one should study the Hibernate entity states as **transient, persistent, detached and removed** very carefully in order to have in-depth insights for Hibernate ***CRUD*** operations.

This unit also covers the overview of **Spring Data JPA** with its advantages and explains how this reduces the effort of a developer to implement the data access layer. The next unit covers Spring Data JPA in detail.

## 10.1 OBJECTIVES

After going through this unit, you will be able to:

- describe Hibernate as ORM tool,
- explain Hibernate architecture,
- use Annotations in Hibernate and Java-based Hibernate configuration,
- use Hibernate CRUD operations such as save, persist, saveOrUpdate,
- differenciate between same task performing operations such asget Vs load etc.,
- establish the relationship between Spring Data JPA, JPA and Hibernate, and
- use Spring Data JPA.

## 10.2 HIBERNATE OVERVIEW

Hibernate is an open-source Java persistence framework created by Gavin King in 2011. It simplifies the development of Java applications to interact with databases. Hibernate is a lightweight, powerful and high-performance ORM (Object Relational Mapping) tool that simplifies data creation, data manipulation and data access.

**The Java Persistence API (JPA) is a specification that defines how to persist data in Java applications.** Hibernate is a popular ORM which is a standard implementation of the JPA specification with a few additional features specific to Hibernate. Hibernate lies between Java Objects and database servers to handle all persistence and retrieval of those objects using appropriate mechanisms and patterns. A schematic diagram is shown in Figure 10.1

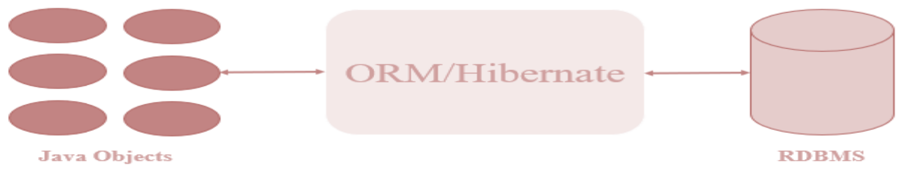


Figure10.1 Hibernate ORM

### 10.2.1 Hibernate Architecture

Hibernate has four-layered architecture. The Hibernate includes many objects such as persistent objects, session factory, session, connection factory, transaction factory, transaction etc. High-level architecture diagram is shown in Figure 10.2

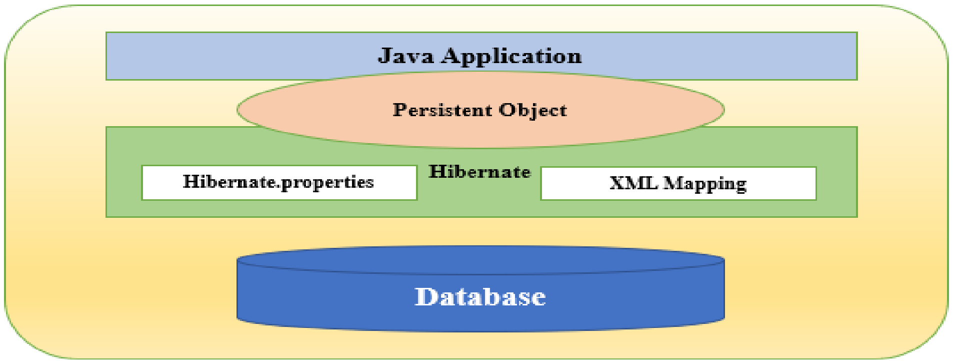


Figure 10.2 Hibernate Architecture

A detailed view of the Hibernate Application Architecture with some of the core classes is shown in Figure 10.3

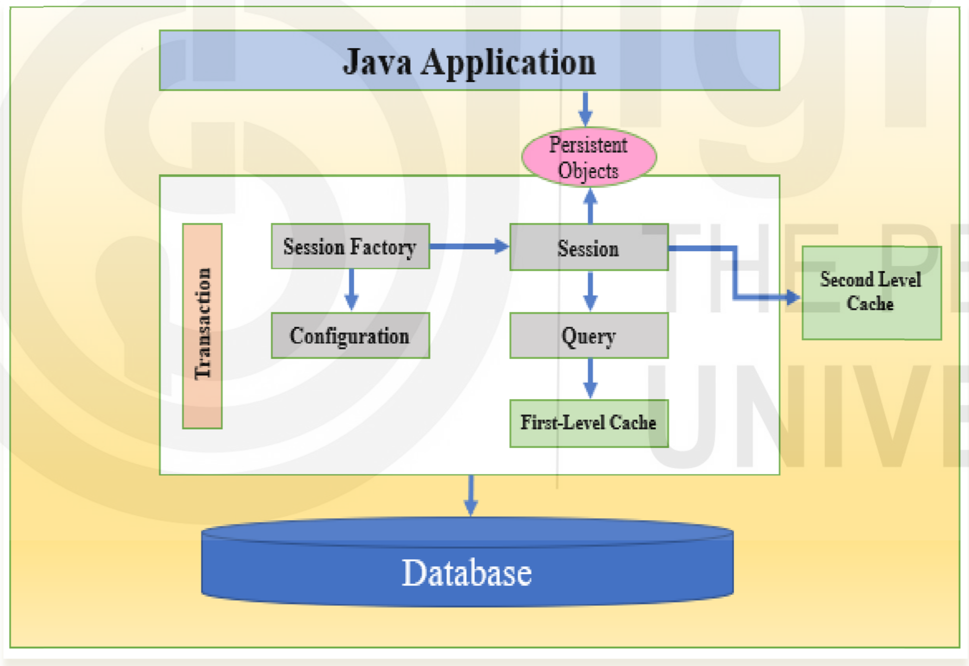


Figure10.3 Hibernate Detailed Architectural View

Hibernate uses Java APIs like Java Transaction API (JTA), Java Database Connectivity (JDBC) and Java Naming and Directory Interface (JNDI). The knowledge of Hibernate architecture elements helps you to understand the internal working of Hibernate.

### Configuration

An instance of Configuration allows the application to specify properties and mapping documents to be used when creating a SessionFactory. The Configuration object is the first object which is being created in the Hibernate application. The Configuration is only an initialization-time object. Usually, a Hibernate application creates a single Configuration, builds a single immutable SessionFactory and then instantiate Sessions in threads servicing client requests. Configuration represents an entire set of mappings of an application's Java types to an SQL database. The Configuration can be configured either programmatically or using configuration file. The configuration file can be either an XML file such as hibernate.cfg.xml or a properties file such as hibernate.properties.

Xml based hibernate configuration for Mysql database is shown below-

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
"-//Hibernate/Hibernate Configuration DTD//EN"
"http://hibernate.sourceforge.net/hibernate-configuration-5.0.dtd">

<hibernate-configuration>
<session-factory>
<property
name="hibernate.connection.driver_class">com.mysql.jdbc.Driver</property>
<property
name="hibernate.connection.url">jdbc:mysql://localhost:3306/test</property>
<property name="hibernate.connection.username">root</property>
<property name="hibernate.connection.password">root</property>
<property name="hibernate.connection.pool_size">10</property>
<property name="show_sql">true</property>
<property name="dialect">org.hibernate.dialect.MySQLDialect</property>
<property name="hibernate.current_session_context_class">thread</property>
</session-factory>
</hibernate-configuration>
```

Properties file base hibernate configuration for Mysql database is shown below-

```
hibernate.dialect= org.hibernate.dialect.MySQLDialect
hibernate.connection.driver_class= com.mysql.jdbc.Driver
hibernate.connection.url= jdbc:mysql://localhost:3306/test
hibernate.connection.username= root
hibernate.connection.password=root
hibernate.show_sql=true
hibernate.hbm2ddl=update
```

**SessionFactory**

The Hibernate creates an immutable SessionFactory object using the Configuration object. SessionFactory is used to instantiate the session object in a thread to service client requests. SessionFactory object is thread safe and used by all the threads of an application.

SessionFactory object is per database using a separate configuration file. Thus, for multiple databases, we will require to create multiple SessionFactory objects. The SessionFactory is a heavyweight object, and it is created at the time of application startup.

**Session**

The session object provides an interface to interact with the database from an application. It is lightweight, which instantiates each time an interaction is required with the database. Session objects are used to retrieve and save persistent objects. It is a short-lived object which wraps the JDBC connection. It also provides a first-level cache of data.

**Transaction**

A Transaction represents a unit of work with the database, and most of the RDBMS supports transaction functionality. Transaction object provides a method for transaction management. It enables data consistency and rollback in case something wrong.

**Query**

Query objects use SQL or Hibernate Query Language (HQL) string to retrieve data from the database and create objects. A Query instance is used to bind query parameters, limit the number of results returned by the query, and finally to execute the query.

**Persistent Objects**

These are plain old java objects (POJOs), which get persisted into the database by hibernate. Persistent objects can be configured in configurations files (hibernate.cfg.xml or hibernate.properties) or annotated with @Entity annotation.

**First-level Cache**

Cache is a mechanism that improves the performance of any application. Hibernate provides a caching facility also. First-level cache is enabled by default, and it can’t be disabled. Hibernate ORM reduces the number of queries made to a database in a single transaction using First-level cache. First-level cache is associated with **Session object** and it is available till the session object is alive. First-level cache associated with session objects is not accessible to any other session object in other parts of the application. Important facts about First-level cache is as followings-

- First-level cache scope is session. Once the session object is closed, the first-level cache is also destroyed.
- First-level cache can’t be disabled.
- First time query for an entity into a session is retrieved from the database and stored in the First-level cache associated with the Hibernate session.
- Query for an object again with the same session object will be loaded from the cache.
- Session evict() method is used to remove the loaded entity from Session.
- Session clear() method is used to remove all the entities stored in the cache.

**Second-level Cache**

Second-level cache is used globally in **Session Factory** scope. **Once the session factory is closed, all cache associated with it dies**, and the cache manager is also closed down. Working of Second-level cache is as followings-

- At first, the hibernate session tries to load an entity, then First-level cache is looked up for cached copy of entity.
- If an entity is available in First-level cache, it is returned as a result of the load method.
- If an entity is not available in First-level cache, Second-level cache is looked up for cached copy of the entity.
- If an entity is available into Second-level cache, it is returned as a result of the load method. But before returning the result, the First-level cache is being updated so that the next invocation for the same entity can be returned from the First-level cache itself.
- If an entity is not found in any of cache, the database query is executed, and both the cache is being updated.
- Second level cache validates itself for modified entities, if the modification has been done through hibernate session APIs.

**☛Check Your Progress 1**

1) What is Hibernate Framework? List down its advantages.

…………………………………………………………………………
…………………………………………………………………………
…………………………………………………………………………
…………………………………………………………………………

2) Explain some important interfaces of Hibernate framework.

…………………………………………………………………………
…………………………………………………………………………
…………………………………………………………………………
…………………………………………………………………………

3) What is Hibernate caching? Explain Hibernate first level cache.

…………………………………………………………………………
…………………………………………………………………………
…………………………………………………………………………
…………………………………………………………………………

4) Explain the working of second level cache.

…………………………………………………………………………
…………………………………………………………………………
…………………………………………………………………………
…………………………………………………………………………

## 10.3 HIBERNATE CONFIGURATION WITH

## ANNOTATION

This section explains various annotations available in Hibernate. Later a “Hello World” hibernate example will be created using explained annotation.

### @Entity

In JPA, POJOs are entities that represent the data that can be persisted into a database. An entity represents the **table** in a database. Every instance of an entity represents a **row** in the table. Let’s say there is a POJO named Student, which represents the data of **Student**. In order to store the student records into a database, **Student** POJO must be defined as an entity so that JPA is aware of it. This can be done by annotating POJO class with **@Entity** annotation. The annotation must be defined at **class level**. The Entity must have no-arg constructor and a primary key. The entity name defaults to the name of the class. Its name can be changed using the name element into @Entity.

```
@Entity(name = “student”)

public class Student

 {

}
```

Entity classes must not be declared final since JPA implementations try to subclass the entity in order to provide their functionality.

### @Id

The primary key uniquely identifies the instances of the entity. The primary key is a must for each JPA entity. The **@Id** annotation is used to define the primary key in a JPA entity. Identifiers are generated using **@GeneratedValue** annotation. There are four strategies to generate id. The ***strategy*** element in **@GeneratedValue** can have value from any of ***AUTO, TABLE, SEQUENCE,* or *IDENTITY.***

```
@Entity
public class Student

{

@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
@Column(name = "id")
privateintid;

}
```

### @Table

Bydefault, the name of the table in the database will be the same as the entity name. **@Table** annotation is used to define the name of the table in the database.

```
@Entity
```

```
@Table(name="STUDENT")

publicclass Student

{

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

@Column(name = "id")

privateintid;

}
```

### @Column

Similar to @Table annotation, @Column annotation is used to provide the details of columns into the database. Check the other elements defined into @Column, such as ***length, nullable, unique***.

```
@Entity

@Table(name="STUDENT")

publicclass Student

{

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

@Column(name = "id")

privateintid;


@Column(name = "firstname", length=50, nullable=false, unique=false)

private String firstName;

}
```

### @Transient

**@Transient** annotation is used to make a field into an entity as non-persistent. For example, the age of a student can be calculated from the date of birth. Thus, age field can be made as non-persistent into Student class as shown below-

```
@Entity

@Table(name=”STUDENT”)

publicclass Student

{
```

```
@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

@Column(name = “id”)

privateintid;

@Column(name = “firstname”, length=50, nullable=false, unique=false)

private String firstName;

@Column(name = “lastname”, length=50, nullable=true, unique=false)

private String lastName;

@Transient

privateintage;

}
```

## @Temporal

The types in the java.sql package such as java.sql.Date, java.sql.Time, java.sql.Timestamp are in line with SQL data types. Thus, its mapping is straightforward, and either the **@basic** or **@column** annotation can be used. The type **java.util.Date** contains both date and time information. This is not directly related to any SQL data type. For this reason, another annotation is required to specify the desired SQL type. **@Temporal** annotation is used to specify the desired SQL data type. It has a single element *TemporalType*. TemporalType can be **DATE, TIME or TIMESTAMP**. The type of java.util.Calendar also requires **@Temporal** annotation to map with the corresponding SQL data type.

```
@Entity
@Table(name="STUDENT")
publicclass Student
{
@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
@Column(name = "id")
privateintid;
@Column(name = "firstname", length=50, nullable=false, unique=false)
private String firstName;
@Column(name = "lastname", length=50, nullable=true, unique=false)
private String lastName;
@Transient
privateintage;
@Temporal(TemporalType.DATE)
@Column(name = "dateofbirth", nullable = false)
private Date dob;
}
```

The types from the *java.time* package in **Java 8**is directly mapped to corresponding SQL types. So there's no need to explicitly specify *@Temporal* annotation:

… *LocalDate* is mapped to *DATE*

- *Instant*, *LocalDateTime*, *OffsetDateTime* and *ZonedDateTime* are mapped to *TIMESTAMP*
- *LocalTime* and *OffsetTime* are mapped to *TIME*

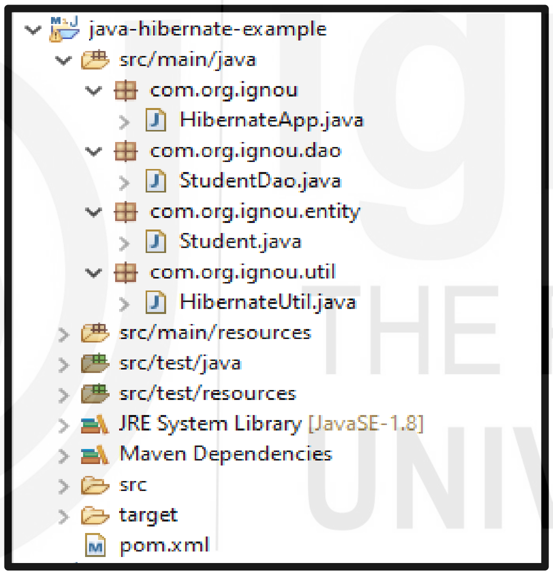
### 10.3.1 A Complete Hibernate Example

This section explains all steps required to create Hibernate Application with Mysql using Java configuration without using hibernate.cfg.xml. In the application, the Data Access Object (DAO) layer persists the Student entity and retrieves the persisted entity from the database. Required tools and technologies are as follows:

- IDE – Eclipse
- Hibernate 5.3.7.Final
- Maven
- MySQL 8
- JavaSE 1.8

Perform the following steps to create and run the Hibernate Application.

**Step 1:** Maven project is created into Eclipse IDE. The directory structure of the project with various layers is shown below-



```
java-hibernate-example
  src/main/java
    com.org.ignou
      HibernateApp.java
    com.org.ignou.dao
      StudentDao.java
    com.org.ignou.entity
      Student.java
    com.org.ignou.util
      HibernateUtil.java
  src/main/resources
  src/test/java
  src/test/resources
  JRE System Library [JavaSE-1.8]
  Maven Dependencies
  src
  target
  pom.xml
```

**Step 2:** Create a maven project with the following dependencies in pom.xml

```
<dependencies>
                <!-- https://mvnrepository.com/artifact/mysql/mysql-connector-java -->
                <dependency>
                        <groupId>mysql</groupId>
                        <artifactId>mysql-connector-java</artifactId>
                        <version>8.0.13</version>
                </dependency>
                <!-- https://mvnrepository.com/artifact/org.hibernate/hibernate-core -
->
                <dependency>
                        <groupId>org.hibernate</groupId>
                        <artifactId>hibernate-core</artifactId>
                        <version>5.3.7.Final</version>
                </dependency>
                <dependency>
                        <groupId>jakarta.xml.bind</groupId>
                        <artifactId>jakarta.xml.bind-api</artifactId>
                        <version>2.3.2</version>
                </dependency>
</dependencies>
```

**Step 3:** Create Hibernate configuration file named as **HibernateUtil** with Java configuration. This contains the information about the database and JPA entity mapping.

```
publicclassHibernateUtil
{
    privatestaticSessionFactorysessionFactory;
    publicstaticSessionFactorygetSessionFactory()
    {
        if (sessionFactory == null)
        {
        try
            {
                Configuration configuration = newConfiguration();
                // Hibernate settings equivalent to hibernate.cfg.xml's properties
                Properties settings = newProperties();
                settings.put(Environment.DRIVER, "com.mysql.cj.jdbc.Driver");
                settings.put(Environment.URL,
"jdbc:mysql://localhost:3306/test?useSSL=false");
                settings.put(Environment.USER, "root");
                settings.put(Environment.PASS, "root");
                settings.put(Environment.DIALECT,
"org.hibernate.dialect.MySQL5Dialect");
                settings.put(Environment.SHOW_SQL, "true");
                settings.put(Environment.CURRENT_SESSION_CONTEXT_CLASS, "thread");
                settings.put(Environment.HBM2DDL_AUTO, "create-drop");
                configuration.setProperties(settings);
                configuration.addAnnotatedClass(Student.class);
                ServiceRegistryserviceRegistry = newStandardServiceRegistryBuilder()
                                .applySettings(configuration.getProperties()).build();
                sessionFactory = configuration.buildSessionFactory(serviceRegistry);
            }
        catch (Exception e)
{
        e.printStackTrace();
        }
    }
    returnsessionFactory;
    }
}
```

In above configuration, list of possible options for **Environment.HBM2DDL_AUTO** are as followings-

- *validate*: validate the schema, makes no changes to the database.
- *update*: update the schema.

- *create*: creates the schema, destroying previous data.
- *create-drop*: drop the schema when the SessionFactory is closed explicitly, typically when the application is stopped.
- *none*: does nothing with the schema, makes no changes to the database

**Step 4:** Create a JPA Entity/Persistent class. This example creates the Student persistent class, which is mapped to a ***student*** database table. A Persistent class should follow some rules:

- **Prefer non-final class:** Hibernate uses proxies to apply some performance optimization. JPA entity or Hibernate Persistent class as final limits the ability of hibernate to use proxies. Without proxies, the application loses lazy loading which will cost performance.
- **A no-arg constructor:** Hibernate creates an instance of a persistent class using newInstance() method. Thus, a persistent class should have a default constructor at least package visibility.
- **Identifier Property:** Each JPA entity should have a Primary Key. Thus an attribute should be annotated with @Id annotation.
- **Getter and Setter methods:** Hibernate recognizes the method by getter and setter method names by default.

```
/**
 * @authorRahulSingh
 */
@Entity
@Table(name="student")
publicclass Student
{
@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
@Column(name = "id")
privateintid;
@Column(name = "firstname", length=50, nullable=false, unique=false)
private String firstName;
@Column(name = "lastname", length=50, nullable=true, unique=false)
private String lastName;

@Transient
privateintage;
@Temporal(TemporalType.DATE)
@Column(name = "dateofbirth", nullable = false)
private Date dob;
private String email;
publicStudent()
  {
   }
publicStudent(String firstName, String lastName, String email, Date dob)
  {
this.firstName = firstName;
this.lastName = lastName;
this.email = email;
this.dob = dob;
   }
publicintgetId()
  {
returnid;
   }
publicvoidsetId(intid)
  {
this.id = id;
   }
public String getFirstName()
  {
returnfirstName;
   }
publicvoidsetFirstName(String firstName)
  {
this.firstName = firstName;
   }
public String getLastName()
{
returnlastName;
```

```
    }
publicvoidsetLastName(String lastName)
  {
this.lastName = lastName;
    }
public String getEmail()
  {
returnemail;
    }
publicvoidsetEmail(String email)
  {
this.email = email;
    }
publicintgetAge()
  {
        returnage;
    }
    publicvoidsetAge(intage)
  {
this.age = age;
    }
public Date getDob()
  {
        return dob;
    }
publicvoidsetDob(Date dob)
  {
this.dob = dob;
    }
@Override
    public String toString()
  {
return"Student [id=" + id + ", firstName=" + firstName + ", lastName=" + lastName + ", age=" + age + ", dob="
                                                + dob + ", email=" + email + "]";
    }
}
```

**Step 5:** Create a DAO layer that performs the operations to the database. StudentDao code is given below, which performs two operations.

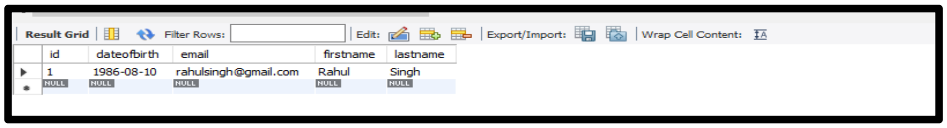
- save(Student student) – saves a given Student into the database.
- getStudentList() – retrieves all students from the database.

```
publicclassStudentDao

{

    publicvoidsave(Student student)

        {

            Transaction txn = null;

            try (Session sess = HibernateUtil.getSessionFactory().openSession())

            {

                txn = sess.beginTransaction(); // start a transaction

                sess.save(student); // save the student object

                txn.commit(); // commit transaction

            }

            catch (Exception e)

            {
                if (txn != null)

                {

                      txn.rollback();

                }

            e.printStackTrace();

            }

      }

            public List<Student>getStudentList()
            {
                  try (Session sess = HibernateUtil.getSessionFactory().openSession())
                  {
                        returnsess.createQuery("from Student", Student.class).list();
                  }
            }

}
```

**Step 6:** Create the main class named as HibernateApp

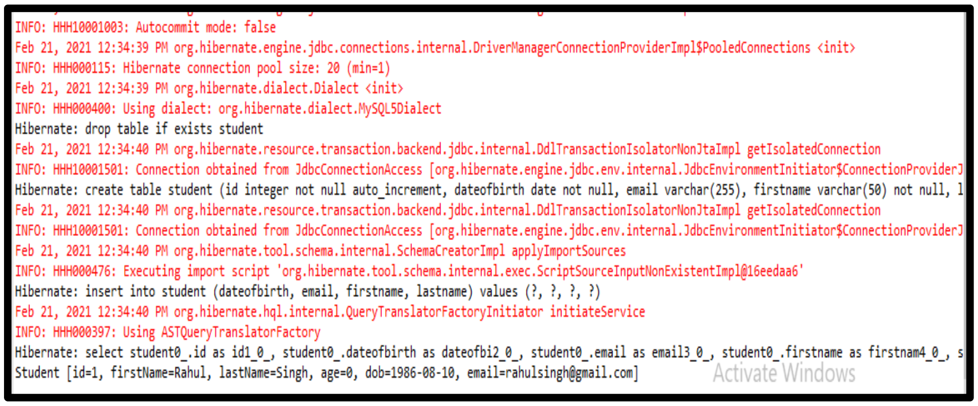
```
publicclassHibernateApp
{
publicstaticvoidmain(String[] args) throws Exception
   {
StudentDaostudentDao = newStudentDao();
SimpleDateFormatdateFormat = newSimpleDateFormat("dd-MM-yyyy");
      Date dob = dateFormat.parse("10-08-1986");
      Student student = newStudent("Rahul", "Singh", "rahulsingh@gmail.com", dob);
studentDao.save(student);
      List < Student >students = studentDao.getStudentList();
students.forEach(s ->System.out.println(s));
   }
}
```

**Step 7:** Run the application from eclipse IDE or command line. Check the logs in the console and record into the database. The application is executed with eclipse IDE. It saves student records into a database. Console log with insert and select query generated by Hibernate as shown below:



| | id | dateofbirth | email | firstname | lastname |
|---|---|---|---|---|---|
| ▶ | 1 | 1986-08-10 | rahulsingh@gmail.com | Rahul | Singh |
| * | NULL | NULL | NULL | NULL | NULL |

Output



```
INFO: HHH10001003: Autocommit mode: false
Feb 21, 2021 12:34:39 PM org.hibernate.engine.jdbc.connections.internal.DriverManagerConnectionProviderImpl$PooledConnections <init>
INFO: HHH000115: Hibernate connection pool size: 20 (min=1)
Feb 21, 2021 12:34:39 PM org.hibernate.dialect.Dialect <init>
INFO: HHH000400: Using dialect: org.hibernate.dialect.MySQL5Dialect
Hibernate: drop table if exists student
Feb 21, 2021 12:34:40 PM org.hibernate.resource.transaction.backend.jdbc.internal.DdlTransactionIsolatorNonJtaImpl getIsolatedConnection
INFO: HHH10001501: Connection obtained from JdbcConnectionAccess [org.hibernate.engine.jdbc.env.internal.JdbcEnvironmentInitiator$ConnectionProviderJ
Hibernate: create table student (id integer not null auto_increment, dateofbirth date not null, email varchar(255), firstname varchar(50) not null, l
Feb 21, 2021 12:34:40 PM org.hibernate.resource.transaction.backend.jdbc.internal.DdlTransactionIsolatorNonJtaImpl getIsolatedConnection
INFO: HHH10001501: Connection obtained from JdbcConnectionAccess [org.hibernate.engine.jdbc.env.internal.JdbcEnvironmentInitiator$ConnectionProviderJ
Feb 21, 2021 12:34:40 PM org.hibernate.tool.schema.internal.SchemaCreatorImpl applyImportSources
INFO: HHH000476: Executing import script 'org.hibernate.tool.schema.internal.exec.ScriptSourceInputNonExistentImpl@16eedaa6'
Hibernate: insert into student (dateofbirth, email, firstname, lastname) values (?, ?, ?, ?)
Feb 21, 2021 12:34:40 PM org.hibernate.hql.internal.QueryTranslatorFactoryInitiator initiateService
INFO: HHH000397: Using ASTQueryTranslatorFactory
Hibernate: select student0_.id as id1_0_, student0_.dateofbirth as dateofbi2_0_, student0_.email as email3_0_, student0_.firstname as firstnam4_0_, s
Student [id=1, firstName=Rahul, lastName=Singh, age=0, dob=1986-08-10, email=rahulsingh@gmail.com]
```

## 10.4 HIBERNATE CRUD (CREATE, READ, UPDATE AND DELETE) FEATURES

This section describes the persistence context. It also explains how hibernate uses the concept of persistence context to solve the problem of managing the entities at runtime.

The persistence context sits between the application and database store. Persistence context keeps monitoring the managed entity and marks the entity as dirty if some modification has been done during a transaction. Persistence context can be considered as a staging area or first-level cache where all the loaded and saved objects to the database reside during a session.

The instance of ***org.hibernate.Session***represents the persistence context in Hibernate. Similarly, the instance of ***javax.persistence.EntityManager***represents the persistence context in JPA. When Hibernate is used as a JPA provider, EntityManager interface is used to perform database related operations. In this case, basically ***java.persistence.EntityManager*** wraps the session object. Hibernate Session has more capability than JPA EntityManager. Thus, sometimes it is useful to work directly with Session.

### Hibernate Entity Lifecycle States

Hibernate works with POJO. Without any Hibernate specific annotation and mapping, Hibernate does not recognize these POJO’s. Once properly annotated with required annotation, hibernate identifies them and keeps track of them to perform database operations such as create, read, update and delete. These POJOs are considered as mapped with Hibernate. An instance of a class mapped with Hibernate can be any of the four persistence states which are known as hibernate entity lifecycle states and depicted in Figure 10.4.

1. Transient
2. Persistent
3. Detached
4. Removed

**Transient:** An object of a POJO class is in a transient state when created using a new

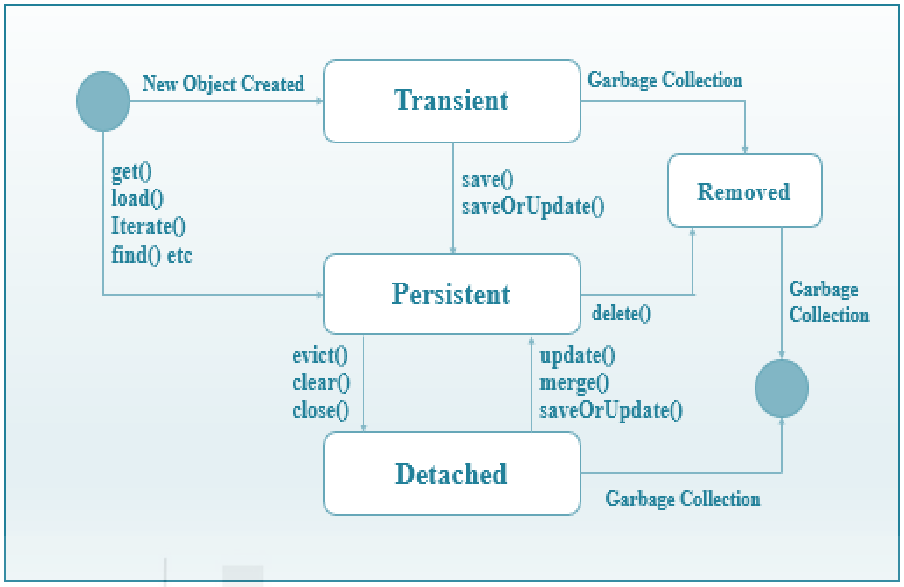


**Figure 10.4: Hibernate Entity Lifecycle States**

operator in Java. In the transient state, the object is not related to any database table. An object in a transient state is not managed by session or Entity Manager. The transient state object is not tracked for any modification from the persistence context. Thus, any modification of a transient state object does not impact the database.

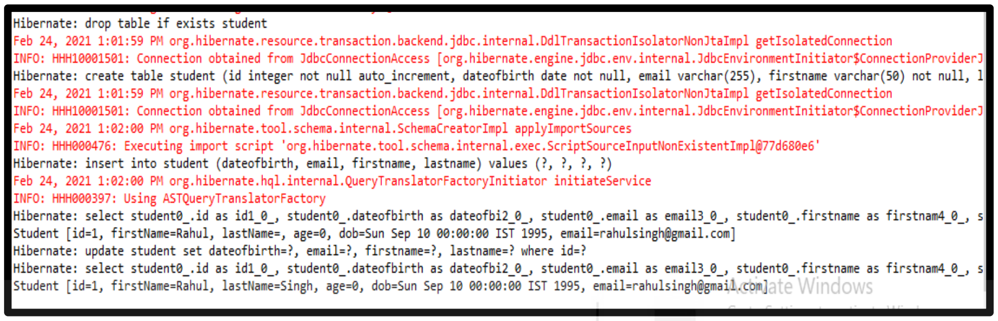
**Persistent:** When a Transient entity is saved, it is associated with a unique session object and it enters into a Persistent state. A Persistent state entity is a representation of a row into a database, although the row might not exist in the database yet; upon flushing the session, the entity is guaranteed to have a corresponding row into the database. Session manages the Persistent state objects and keeps track of every modification done. Session propagates all the modifications into the database automatically. The Hibernate session's following methods make an entity into Persistent state.

- session.save()
- session.update()
- session.saveOrUpdate()
- session.lock()
- session.merge()

```
publicclassPersistentStateExample
{
        publicstaticvoidmain(String[] args)
        {
                Transaction txn = null;
                try (Session sess = HibernateUtil.getSessionFactory().openSession())
                {
                        txn = sess.beginTransaction(); // start a transaction
                        SimpleDateFormatdateFormat = newSimpleDateFormat("dd-MM-yyyy");
                        Date dob = dateFormat.parse("10-09-1995");
                        Student student = newStudent("Rahul", "", "rahulsingh@gmail.com", dob); // 1
                        sess.saveOrUpdate(student); // 2
                        List<Student>students = sess.createQuery("from Student", Student.class).list();
                        students.forEach(s ->System.out.println(s)); // 3
                        student.setLastName("Singh"); //4
                        students = sess.createQuery("from Student",Student.class).list();
                        students.forEach(s ->System.out.println(s)); //5
                        txn.commit(); // 6
                }
                catch (Exception e)
```

```
                        {
                                if (txn != null)
                                {
                                        txn.rollback();
                                }
                                e.printStackTrace();
                        }
                }
}
```

Output:



```
Hibernate: drop table if exists student
Feb 24, 2021 1:01:59 PM org.hibernate.resource.transaction.backend.jdbc.internal.DdlTransactionIsolatorNonJtaImpl getIsolatedConnection
INFO: HHH10001501: Connection obtained from JdbcConnectionAccess [org.hibernate.engine.jdbc.env.internal.JdbcEnvironmentInitiator$ConnectionProviderJ
Hibernate: create table student (id integer not null auto_increment, dateofbirth date not null, email varchar(255), firstname varchar(50) not null, l
Feb 24, 2021 1:01:59 PM org.hibernate.resource.transaction.backend.jdbc.internal.DdlTransactionIsolatorNonJtaImpl getIsolatedConnection
INFO: HHH10001501: Connection obtained from JdbcConnectionAccess [org.hibernate.engine.jdbc.env.internal.JdbcEnvironmentInitiator$ConnectionProviderJ
Feb 24, 2021 1:02:00 PM org.hibernate.tool.schema.internal.SchemaCreatorImpl applyImportSources
INFO: HHH000476: Executing import script 'org.hibernate.tool.schema.internal.exec.ScriptSourceInputNonExistentImpl@77d680e6'
Hibernate: insert into student (dateofbirth, email, firstname, lastname) values (?, ?, ?, ?)
Feb 24, 2021 1:02:00 PM org.hibernate.hql.internal.QueryTranslatorFactoryInitiator initiateService
INFO: HHH000397: Using ASTQueryTranslatorFactory
Hibernate: select student0_.id as id1_0_, student0_.dateofbirth as dateofbi2_0_, student0_.email as email3_0_, student0_.firstname as firstnam4_0_, s
Student [id=1, firstName=Rahul, lastName=, age=0, dob=Sun Sep 10 00:00:00 IST 1995, email=rahulsingh@gmail.com]
Hibernate: update student set dateofbirth=?, email=?, firstname=?, lastname=? where id=?
Hibernate: select student0_.id as id1_0_, student0_.dateofbirth as dateofbi2_0_, student0_.email as email3_0_, student0_.firstname as firstnam4_0_, s
Student [id=1, firstName=Rahul, lastName=Singh, age=0, dob=Sun Sep 10 00:00:00 IST 1995, email=rahulsingh@gmail.com]
```

Check the following code into the above example code.

1. The ***student*** object created using the ***new*** operator is in a transient state.
2. session.saveOrUpdate() method makes the **student** object into persistent state. Once the object is in persistent state, Session keeps track of any modification and automatically flushes the changes into the database.
3. All the records of the student table are being rendered. Notice that at this point lastName as empty for id 1.
4. The student's last name is updated. The student entity is in the persistent state, and Session keeps track of this change and updates the database automatically. The update statement is issued and can be found in the log.
5. All the records of the student table are being rendered. Check at this point lastNameis updated for id 1.
6. The transaction is committed, and all required changes are synced with the database.

**Detached:** On call of any session method such as clear(), evict() or close() make the persistent state objects into the detached state. Session does not keep track of any modification to detached state objects. Detached entity has a corresponding row in the database but changes to the entity will not be reflected in the database.

In order to persist the changes of entity, detached entities must be re-attached to the valid Hibernate Session. The detached state entity can be re-attached to the Hibernate Session with the call of following methods-

... session.update()
... session.merge()
... session.saveOrUpdate()

```
publicclassDetachedStateExample
{
        publicstaticvoidmain(String[] args)
        {
                try
                {
                        Session sess = HibernateUtil.getSessionFactory().openSession();
                        SimpleDateFormatdateFormat = newSimpleDateFormat("dd-MM-yyyy");
                        Date dob = dateFormat.parse("10-09-1995");
                        Student student = newStudent("Rahul", "", "rahulsingh@gmail.com", dob);
                        sess.saveOrUpdate(student); //student entity is in persistent state
                        List<Student>students = sess.createQuery("from Student",
Student.class).list();
                        sess.close(); // session close() makes student entity in detached state.
                        students.forEach(s ->System.out.println(s));
                        student.setLastName("Singh"); //This change is not tracked by Hibernate
session.
                        sess = HibernateUtil.getSessionFactory().openSession();
                        sess.update(student); // session update() makes the detached object into
persistent state.
                        students = sess.createQuery("from Student",Student.class).list();
                        students.forEach(s ->System.out.println(s));
                        sess.close();
                }
                catch (Exception e)
                {
                        e.printStackTrace();
                }
        }
}
```

**Removed:** Once a persistent object passes through session's delete() method, it enters into Removed state. When an application commits the session the entries in the database that correspond to remove entities are deleted. At this state, java instance exists but any change made to the instance will not be saved into the database. If the removed entities are not referenced anywhere, then it is eligible for garbage collection.

The following sections explain the various Hibernate operations and differences between the same types of operations.

## 10.4.1 Hibernate Get Entity – get vs load

The Hibernate provides session.load() and session.get() methods to fetch the entity by id. This section explains both the methods with examples and differences between them.

**Session load() method:** There are several overloaded methods of load() into the Hibernate Session interface. Each load() method requires the object's primary key as an identifier to load the object from the database. Along with the identifier, hibernate also requires to know which class or entity name to use to find the object. Some important overloaded methods are as follows:

- public Object load(Class<T>theClass, Serializable id) throws HibernateException
- public Object load(String entityName, Serializable id) throws HibernateException
- public void load(Object object, Serializable id) throws HibernateException

As we see that return type of load() method is either Object or void. Thus, we need to typecast the returned object with a suitable type of class. There are other overloaded methods of load(), which requires lock mode as an argument too. Hibernate picks the correct lock mode for us. Thus, very rarely an overloaded load() method with lock mode argument is used.

```
publicclassEntityLoadExample

{

    publicstaticvoidmain(String[] args)
    {
        try
        {
                Session sess = HibernateUtil.getSessionFactory().openSession();
                SimpleDateFormatdateFormat = newSimpleDateFormat("dd-MM-yyyy");
                Date dob = dateFormat.parse("10-09-1995");
                Student student = newStudent("Rahul", "Singh", "rahulsingh@gmail.com", dob);
                sess.save(student); //student entity is in persistent state
                sess.close();

                intstudentId = student.getId(); //Id has been generated and set on call of save method.

                // First Overloaded method example
                Session sess1 = HibernateUtil.getSessionFactory().openSession();
                Student student1 = (Student)sess1.load(Student.class, studentId);
                System.out.println(student1);
                sess1.close();

                // Second Overloaded method example
                Session sess2 = HibernateUtil.getSessionFactory().openSession();
                Student student2 = (Student)sess2.load("com.org.ignou.entity.Student", studentId);
                System.out.println(student2);
                sess2.close();

                // Third Overloaded method example
                Session sess3 = HibernateUtil.getSessionFactory().openSession();
                Student student3 = newStudent();
                sess3.load(student3, studentId);
                System.out.println(student3);
                sess3.close();

            }
            catch (Exception e)
            {
                e.printStackTrace();
            }
        }
}
```

**Session get() method:** Theget() method is similar to the load() method to fetch the entity from the database. Like load() method, get() methods also take an identifier and either an entity name of a class. Some important overloaded methods are as follows:

- public Object get(Class<T>clazz, Serializable id) throws HibernateException
- public Object get(String entityName, Serializable id) throws HibernateException

**Difference between load() and get() methods**

Although load() and get() methods perform the same task, still the differences exist in their return values in case the identifier does not exist in the database.

- The get() method returns NULL if the identifier does not exist.
- The load() method throws a runtime exception if the identifier does not exist.

## 10.4.2 Hibernate Insert Entity – persist, save and saveOrUpdate

The Session interface into Hibernate provides a couple of methods to transit an object from transient state or detached state to persistent state e.g. persist(), save(), and saveOrUpdate(). These methods are used to store an object in a database. There are significant differences among these methods.

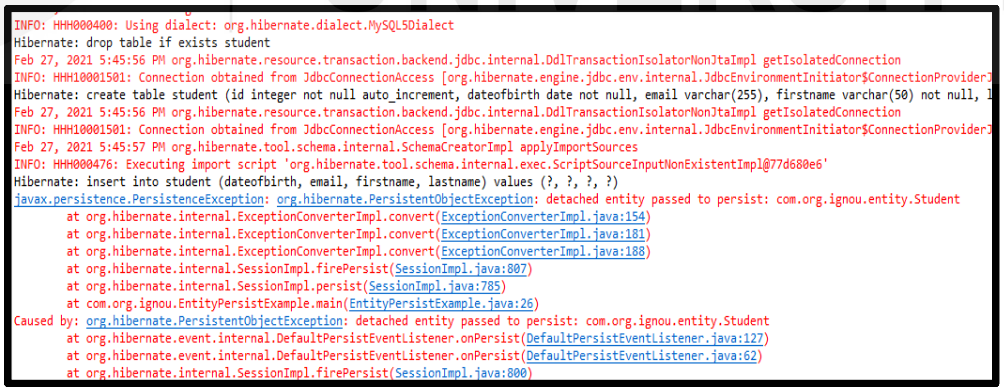
**Session persist() method:** The ***persist()*** method is used to add a record into the database. This method adds a new entity instance to persistent context. On call of ***persist()*** method, the transit state object moves into a persistent state object but not yet saved to the database. The insert statement generates only upon committing the transaction, flushing or closing the session. This method has a return type as **void**. The semantics of ***persist()*** method is as follows:

- A transient state object becomes a persistent object, and the operation cascades all of its relation with cascade=PERSIST or cascade=ALL. The spec does not say that persist() method will generate the **Id** right away but on commit or flush,**non-null Id** is guaranteed. The persist() method does not promise the Id as non-null just after calling this method. Thus one should not rely upon it.
- The persist() method has no impact on persistent objects, but the operation still cascaded all of its relation with cascade=PERSIST or cascade=ALL.
- The persist() method on detached objects throws an exception, either upon calling this method or upon committing or flushing the session.

A sample example explains the above points with output. For better understanding, have hands-on with the following code.

```
publicclassEntityPersistExample
{
      publicstaticvoidmain(String[] args)
      {
            try
            {
                  Session sess = HibernateUtil.getSessionFactory().openSession();
                  Transaction txn = sess.beginTransaction();
                  SimpleDateFormatdateFormat = newSimpleDateFormat("dd-MM-yyyy");
                  Date dob = dateFormat.parse("10-09-1995");
                  Student student = newStudent("Abhishek", "Singh", "abhisheksingh@gmail.com", dob);
                  sess.persist(student); //student entity is transitioned from transit -> persistent
                  sess.persist(student); // No impact
                  sess.evict(student); // student entity is transitioned from persistent -> detached

                  sess.persist(student); // Exception. Since persistent state object is being persisted.
                  txn.commit();
                  sess.close();
            }
               catch (Exception e)
            {
                  e.printStackTrace();
            }
      }
}
```

Output:



```
INFO: HHH000400: Using dialect: org.hibernate.dialect.MySQL5Dialect
Hibernate: drop table if exists student
Feb 27, 2021 5:45:56 PM org.hibernate.resource.transaction.backend.jdbc.internal.DdlTransactionIsolatorNonJtaImpl getIsolatedConnection
INFO: HHH10001501: Connection obtained from JdbcConnectionAccess [org.hibernate.engine.jdbc.env.internal.JdbcEnvironmentInitiator$ConnectionProviderJ
Hibernate: create table student (id integer not null auto_increment, dateofbirth date not null, email varchar(255), firstname varchar(50) not null, l
Feb 27, 2021 5:45:56 PM org.hibernate.resource.transaction.backend.jdbc.internal.DdlTransactionIsolatorNonJtaImpl getIsolatedConnection
INFO: HHH10001501: Connection obtained from JdbcConnectionAccess [org.hibernate.engine.jdbc.env.internal.JdbcEnvironmentInitiator$ConnectionProviderJ
Feb 27, 2021 5:45:57 PM org.hibernate.tool.schema.internal.SchemaCreatorImpl applyImportSources
INFO: HHH000476: Executing import script 'org.hibernate.tool.schema.internal.exec.ScriptSourceInputNonExistentImpl@77d680e6'
Hibernate: insert into student (dateofbirth, email, firstname, lastname) values (?, ?, ?, ?)
javax.persistence.PersistenceException: org.hibernate.PersistentObjectException: detached entity passed to persist: com.org.ignou.entity.Student
	at org.hibernate.internal.ExceptionConverterImpl.convert(ExceptionConverterImpl.java:154)
	at org.hibernate.internal.ExceptionConverterImpl.convert(ExceptionConverterImpl.java:181)
	at org.hibernate.internal.ExceptionConverterImpl.convert(ExceptionConverterImpl.java:188)
	at org.hibernate.internal.SessionImpl.firePersist(SessionImpl.java:807)
	at org.hibernate.internal.SessionImpl.persist(SessionImpl.java:785)
	at com.org.ignou.EntityPersistExample.main(EntityPersistExample.java:26)
Caused by: org.hibernate.PersistentObjectException: detached entity passed to persist: com.org.ignou.entity.Student
	at org.hibernate.event.internal.DefaultPersistEventListener.onPersist(DefaultPersistEventListener.java:127)
	at org.hibernate.event.internal.DefaultPersistEventListener.onPersist(DefaultPersistEventListener.java:62)
	at org.hibernate.internal.SessionImpl.firePersist(SessionImpl.java:800)
```

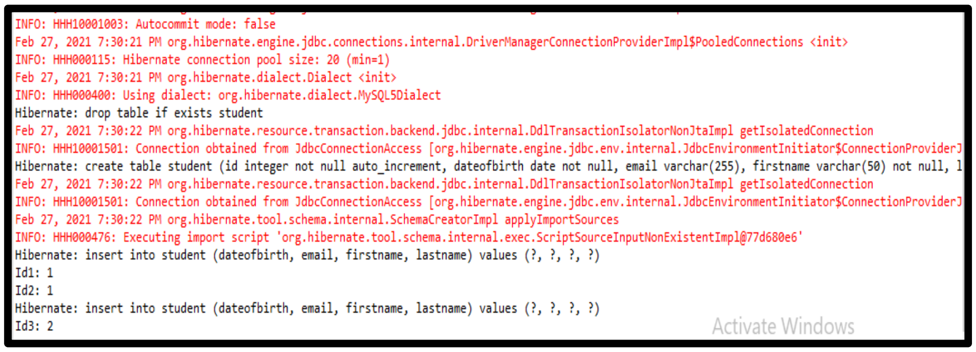
**Session save() method:** The ***save()*** method is similar to the persist***()*** method which stores the record into the database. This is the original Hibernate Session method which does not conform to JPA specification. The ***save()*** method differs from the persist***()*** method in terms of its implementation. The documentation of this method states that at first, it assigns the generated id to the identifier and then persists the entity. The ***save()*** method returns the **Serializable** value of this identifier. The semantics of ***save()*** method is as follows:

- A transient state object becomes a persistent object and the operation cascades all of its relation with cascade=PERSIST or cascade=ALL. At first it assigns the generated id to the identifier and then persists the entity.
- The ***save()*** method has no impact on persistent objects, but the operation still cascaded all of its relations with cascade=PERSIST or cascade=ALL.
- The ***save()*** method on detached instances creates a new persistent instance. It assigns the new generated Id to the identifier, which results in a duplicate record in the database upon committing or flushing.

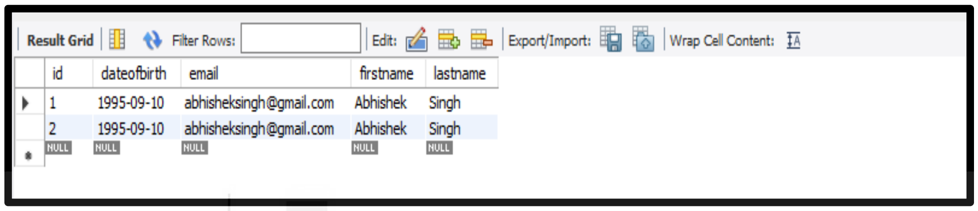
A sample example explains the above points with output as shown below:

```
publicclassEntitySaveExample

{

    publicstaticvoidmain(String[] args)

    {

        try

        {

            Session sess = HibernateUtil.getSessionFactory().openSession();

            Transaction txn = sess.beginTransaction();

            SimpleDateFormatdateFormat = newSimpleDateFormat("dd-MM-yyyy");

            Date dob = dateFormat.parse("10-09-1995");

            Student student = newStudent("Abhishek", "Singh", "abhisheksingh@gmail.com", dob);

            Integer id1 = (Integer) sess.save(student); //student entity is transitioned from transit ->
persistent

            System.out.println("Id1: "+id1);

            Integer id2 = (Integer) sess.save(student); // No impact

            System.out.println("Id2: "+id2);

            sess.evict(student); // student entity is transitioned from persistent -> detached

            Integer id3 = (Integer) sess.save(student); // New Id generated and assigns it to identifier

            System.out.println("Id3: "+id3);

            txn.commit();

            sess.close();

        }

        catch (Exception e)

        {

          e.printStackTrace();

        }

    }
}
```

Execution Result:



```
INFO: HHH10001003: Autocommit mode: false
Feb 27, 2021 7:30:21 PM org.hibernate.engine.jdbc.connections.internal.DriverManagerConnectionProviderImpl$PooledConnections <init>
INFO: HHH000115: Hibernate connection pool size: 20 (min=1)
Feb 27, 2021 7:30:21 PM org.hibernate.dialect.Dialect <init>
INFO: HHH000400: Using dialect: org.hibernate.dialect.MySQL5Dialect
Hibernate: drop table if exists student
Feb 27, 2021 7:30:22 PM org.hibernate.resource.transaction.backend.jdbc.internal.DdlTransactionIsolatorNonJtaImpl getIsolatedConnection
INFO: HHH10001501: Connection obtained from JdbcConnectionAccess [org.hibernate.engine.jdbc.env.internal.JdbcEnvironmentInitiator$ConnectionProviderJ
Hibernate: create table student (id integer not null auto_increment, dateofbirth date not null, email varchar(255), firstname varchar(50) not null, l
Feb 27, 2021 7:30:22 PM org.hibernate.resource.transaction.backend.jdbc.internal.DdlTransactionIsolatorNonJtaImpl getIsolatedConnection
INFO: HHH10001501: Connection obtained from JdbcConnectionAccess [org.hibernate.engine.jdbc.env.internal.JdbcEnvironmentInitiator$ConnectionProviderJ
Feb 27, 2021 7:30:22 PM org.hibernate.tool.schema.internal.SchemaCreatorImpl applyImportSources
INFO: HHH000476: Executing import script 'org.hibernate.tool.schema.internal.exec.ScriptSourceInputNonExistentImpl@77d680e6'
Hibernate: insert into student (dateofbirth, email, firstname, lastname) values (?, ?, ?, ?)
Id1: 1
Id2: 1
Hibernate: insert into student (dateofbirth, email, firstname, lastname) values (?, ?, ?, ?)
Id3: 2
```



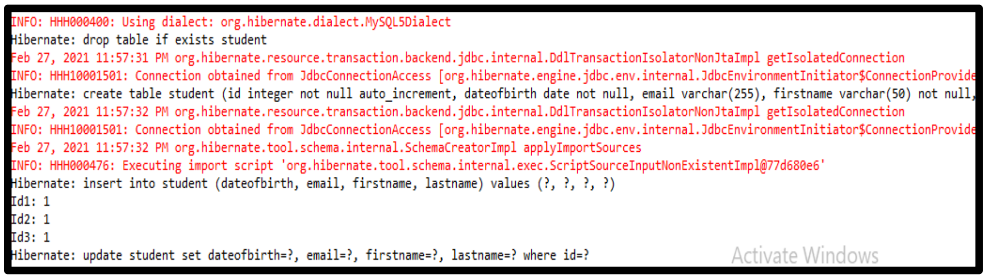
| id | dateofbirth | email | firstname | lastname |
|---|---|---|---|---|
| 1 | 1995-09-10 | abhisheksingh@gmail.com | Abhishek | Singh |
| 2 | 1995-09-10 | abhisheksingh@gmail.com | Abhishek | Singh |
| NULL | NULL | NULL | NULL | NULL |

It can be observed in the above log that the detached instance is saved into the database with a newly generated id and which results in a duplicate record into the database as shown in the database screenshot.

**Session saveOrUpdate() method:** This method either performs **save()** or **update()** on the basis of identifier existence. E.g. If an identifier exists for the instance, update() method will be called otherwise save() will be performed. The saveOrUpdate() method handles the cases where we need to save a detached instance. Unlike the save() operation on detached instances, the saveOrUpdate() method does not result in a duplicate record. Similar to update(), this method is used to reattach an instance to the session. This method can be considered as a universal tool to make an object persistent regardless of its state, whether it is transient or detached.

```
publicclassEntitySaveOrUpdateExample
{
    publicstaticvoidmain(String[] args)
    {
        try
        {
            Session sess = HibernateUtil.getSessionFactory().openSession();
            Transaction txn = sess.beginTransaction();
            SimpleDateFormatdateFormat = newSimpleDateFormat("dd-MM-yyyy");
            Date dob = dateFormat.parse("10-09-1995");
            Student student = newStudent("Abhishek", "Singh", "abhisheksingh@gmail.com", dob);
            sess.saveOrUpdate(student);   //student entity is transitioned from transit -> persistent
            System.out.println("Id1: "+student.getId());
            sess.saveOrUpdate(student);  // No impact
            System.out.println("Id2: "+student.getId());

            sess.evict(student);   // student entity is transitioned from persistent -> detached

            sess.saveOrUpdate(student); // Since Id exist, thus only update operation is performed on
detached entity
            System.out.println("Id3: "+student.getId());
            txn.commit();
            sess.close();
        }
        catch (Exception e)
        {
          e.printStackTrace();
        }
    }
}
```

Program Output:



```
INFO: HHH000400: Using dialect: org.hibernate.dialect.MySQL5Dialect
Hibernate: drop table if exists student
Feb 27, 2021 11:57:31 PM org.hibernate.resource.transaction.backend.jdbc.internal.DdlTransactionIsolatorNonJtaImpl getIsolatedConnection
INFO: HHH10001501: Connection obtained from JdbcConnectionAccess [org.hibernate.engine.jdbc.env.internal.JdbcEnvironmentInitiator$ConnectionProvide
Hibernate: create table student (id integer not null auto_increment, dateofbirth date not null, email varchar(255), firstname varchar(50) not null,
Feb 27, 2021 11:57:32 PM org.hibernate.resource.transaction.backend.jdbc.internal.DdlTransactionIsolatorNonJtaImpl getIsolatedConnection
INFO: HHH10001501: Connection obtained from JdbcConnectionAccess [org.hibernate.engine.jdbc.env.internal.JdbcEnvironmentInitiator$ConnectionProvide
Feb 27, 2021 11:57:32 PM org.hibernate.tool.schema.internal.SchemaCreatorImpl applyImportSources
INFO: HHH000476: Executing import script 'org.hibernate.tool.schema.internal.exec.ScriptSourceInputNonExistentImpl@77d680e6'
Hibernate: insert into student (dateofbirth, email, firstname, lastname) values (?, ?, ?, ?)
Id1: 1
Id2: 1
Id3: 1
Hibernate: update student set dateofbirth=?, email=?, firstname=?, lastname=? where id=?
```

As in output, you can see that saveOrUpdate() operation does not result in duplicate records even for the detached entity . Thus, this operation can be used as a universal tool to make an object persistent regardless of its state.

## 10.4.3 Hibernate Query Language (HQL)

Hibernate Query Language (HQL) is an object-oriented query language. HQL is similar to the database SQL language. The main difference between HQL and SQL is HQL uses class name instead of table name, and property names instead of column name. HQL queries for Select, Update, Delete and Insert is explained below.

**HQL Select** query to retrieve the student data of id 1.

```
Query query = session.createQuery("from Student where id = :id ");
query.setParameter("id", "1");
List list = query.list();
```

**HQL Update** query to update the student's last name as 'Singh' whose id 1.

```
Query query = session.createQuery("Update Student set lastName = :lastName where id = :id
");
query.setParameter("lastName", "Singh");
query.setParameter("id", "1");
int result = query.executeUpdate();
```

**HQL Delete** query to delete the student where id is 1.

```
Query query = session.createQuery("Delete Student where id = :id ");
query.setParameter("id", "1");
int result = query.executeUpdate();
```

**HQL Insert**

In HQL, only the INSERT INTO … SELECT … is supported; there is no INSERT INTO … VALUES. HQL only support insert from another table. For example, insert student records from another student_backup table. This is also called bulk-insert statement.

```
Query query = session.createQuery("Insert into Student(firstName, lastName, emailId)"+
                                    "Select    firstName,    lastName,    emailed    from
student_backup");
int result = query.executeUpdate();
```

**☛Check Your Progress 2**

1) Describe @Temporal annotation with its usage.

………………………………………………………………………………
………………………………………………………………………………
………………………………………………………………………………
………………………………………………………………………………

2) List down the prescribed guidelines for a persistent class.

………………………………………………………………………………
………………………………………………………………………………
………………………………………………………………………………
………………………………………………………………………………

3) Explain Hibernate entity lifecycle state.

………………………………………………………………………………
………………………………………………………………………………
………………………………………………………………………………
………………………………………………………………………………

4) Explain HQL with examples.

………………………………………………………………………………
………………………………………………………………………………
………………………………………………………………………………
………………………………………………………………………………

## 10.5 SPRING DATA JPA OVERVIEW

A well-structured codebase of a web project consists of various layers such as Controller, Service and DAO (data access object) layer. Each layer has its own responsibilities. While developing a web application, business logic should be focused instead of technical complexity and boilerplate code. The DAO layer usually consists of lots of boilerplate code, and that can be simplified. The simplification reduces the number of artifacts that need to be defined and maintained. It also makes the data access pattern and configuration consistent.

Different storage technologies require different API's to access data, different configurations and different methods. JPA handles the object-relational mappings and technical complexities in JDBC based database interaction. The JPA provides standardization across SQL data stores. The Java Persistence API (JPA) specification and Spring Data JPA are very popular to simplify the DAO layer. Before the explanation of Spring Data JPA, the following section explains the relationship between Spring Data JPA, JPA and Hibernate.

Java Persistence API (JPA) is just a specification that defines APIs for object-relational mappings and management of persistent objects. It is a set of interfaces that can be used to implement the persistence layer. It itself doesn't provide any implementation classes. In order to use the JPA, a provider is required which implements the specifications. Hibernate and EclipseLink are popular JPA providers.

**Spring Data JPA** is one of the many sub-projects of **Spring Data**that simplifies the data access for relational data stores. Spring Data JPA is not a JPA provider; instead it wraps the JPA provider and adds its own features like a no-code implementation of the repository pattern. Spring Data JPA uses Hibernate as the default JPA provider. JPA provider is configurable, and other providers can also be used with Spring Data JPA. Spring Data JPA provides a complete abstraction over the DAO layer into the project.

Spring Data JPA introduces the concept of **JPA Repositories**. JPA Repositories are a set of Interfaces that defines query methods. It does not require writing native queries anymore. Sometimes custom queries may be required in order to fetch the data from the database. These queries are JPQL (Java Persistence Query Language), not native queries. The advantages of using Spring Data JPA are as follows.

### DAO Abstraction with No-Code Repositories

Spring Data JPA defines many Repository Interfaces such as ***CrudRepository, PagingAndSortingRepository, JpaRepository***having methods to store, retrieve, sorted retrieval, paginated result and many more. The interfaces only define query methods and spring provides proxy implementation at runtime. With Spring Data JPA, we don't have to write SQL statement, instead we just need to extend the interface defined by Spring Data JPA for one of the entities. Based on JPA specification, the underlying JPA implementation enables the *Entity* objects and their metadata mapping. It also enables the entity manager who is responsible for persisting and retrieving entities from the database.

### Query Methods

Another robust and comfortable feature of Spring Data JPA is the Query Methods. Based on the name of methods declared in the repository interfaces are converted to low-level SQL queries at runtime. If the custom query isn't complex, only a method is required to be defined in the repository interface with a name starting with ***find…By***. Here is an example of a Student Repository to find a student by id and list of students based on firstName and lastName.

```
public interface StudentRepository extends CrudRepository<Student, Long>
{
   Optional<Student>findById(Long studentId);
   List<Student>findByFirstNameAndLastName(String firstName, String lastName);
}
```

1. The first method corresponds to **select * from student where id = ?**
2. The second method corresponds to **select * from student where firstName= ?andlastName = ?**

### Seamless Integration

Spring Data JPA seamlessly integrates ***JPA*** into the Spring stack, and its repositories reduce the boilerplate code required to the JPA specification. **Spring Data JPA** also helps your DAO layer integrating and interacting with other Spring based components in your application, like accessing property configurations or auto-wiring the repositories into the application service layer. It also works perfectly with Spring Boot auto-configuration. We only need to provide the data source details and the rest of the things like configuring and using *EntityManager*.

## ☛Check Your Progress 3

1) What is the difference between load() and get() method of Hibernate Session?

…………………………………………………………………………

…………………………………………………………………………

…………………………………………………………………………

…………………………………………………………………………

2) What is the difference between persist() and save() method of Hibernate Session?

…………………………………………………………………………

…………………………………………………………………………

…………………………………………………………………………

…………………………………………………………………………

3) Why is Hibernate Session's saveOrUpdate() method considered a universal tool to make an object persistent?

…………………………………………………………………………

…………………………………………………………………………

…………………………………………………………………………

…………………………………………………………………………

4) Explain Spring Data JPA with its advantages.

…………………………………………………………………………

…………………………………………………………………………

…………………………………………………………………………

…………………………………………………………………………

## 10.6 SUMMARY

This unit has explained the concepts of ORM, Hibernate as ORM tool and Spring Data JPA, which uses Hibernate as the default JPA provider. Below are the important things which have been explained.

- Many different data storage technologies are available in this world, and each one comes with its own data access API and driver.
- ORM stands for Object Relational Mapping, where a Java object is mapped to a database table and with APIs.We need to work with objects and not with native queries.
- The Java Persistence API (JPA) is a specification that defines how to persist data in Java applications
- The Hibernate architecture elements includ such as SessionFactory, Session, Query, First Level Cache and Second Level Cache. The first level cache can't be disabled.
- In order to use optimized performance of Hibernate, a POJO should follow some rules such as-
    - Prefer non-final class
    - A no-arg constructor
    - Identifier property
    - Getter and Setter method
- The Hibernate entity lifecycle states as transient, persistent, detached and removed.

- The Hibernate Session persist(), and the save() methods perform the same job still; their implementation differs in terms of generated Id assignment.
- The **saveOrUpdate()** method can be used as a universal tool for persisting an entity irrespective of entity state.
- Spring Data JPA takes one step forward and enables developers to use data access layers without any implementation.

## 10.7 SOLUTIONS/ANSWERS TO CHECK YOUR PROGRESS

### Check Your Progress 1

1) Hibernate is an open-source Java persistence framework created by Gavin King in 2011. It simplifies the development of Java applications to interact with databases. Hibernate is a lightweight, powerful, and high-performance ORM (Object Relational Mapping) tool that simplifies data creation, manipulation, and access.
   **The Java Persistence API (JPA) is a specification that defines how to persist data in Java applications.** Hibernate is a popular ORM which is a standard implementation of the JPA specification with a few additional features specific to Hibernate. Hibernate lies between Java Objects and database servers to handle all persistence and retrieval of those objects using appropriate mechanisms and patterns. There are several advantages of Hibernate, such as:
   - Open Source
   - High Performance
   - Light Weight
   - Database independent
   - Caching
   - Scalability
   - Lazy loading
   - Hibernate Query Language (HQL)

2) Hibernate has four-layered architecture. The Hibernate includes many objects such as persistent objects, sessionfactory, session, connection factory, transaction factory, transaction etc. A few important interfaces have been explained.

   **SessionFactory**
   The Hibernate creates an immutable SessionFactory object using the Configuration object. SessionFactory is used to instantiate the session object in thread to service client requests. SessionFactory object is thread-safe and used by all the threads of an application.
   SessionFactory object is per database using a separate configuration file. Thus, for multiple databases, we will require to create multiple SessionFactory objects. The SessionFactory is a heavyweight object, and it is created at the time of application startup.

   **Session**
   The session object provides an interface to interact with the database from an application. A Session object is lightweight, which instantiates each time an interaction is required with the database. Session objects are used to retrieve and save persistent objects. It is a short-lived object which wraps the JDBC connection. It also provides a first-level cache of data.

   **Transaction**

A Transaction represents a unit of work with the database, and most of the RDBMS supports transaction functionality. Transaction object provides methods for transaction management. Transaction objects enable data consistency, and rollback in case something goes wrong.

3) The cache is a mechanism that improves the performance of any application. Hibernate provides a caching facility also. First-level cache is enabled by default, and it can't be disabled. Hibernate ORM reduces the number of queries made to a database in a single transaction using First-level cache. First-level cache is associated with **Session object,** and it is available till session object is alive. First-level cache associated with session objects is not accessible to any other session object in other parts of the application. Important facts about First-level cache is as follows:
   - First-level cache scope is session. Once the session object is closed, the first-level cache is also destroyed.
   - First-level cache can't be disabled.
   - First time query for an entity into a session is retrieved from the database and stored in the First-level cache associated with the hibernate session.
   - Query for an object again with the same session object will be loaded from cache.
   - Session evict() method is used to remove the loaded entity from Session.

   Session's clear() method is used to remove all the entities stored into cache.

**4)** Second-level cache is used globally in **Session Factory** scope. **Once session factory is closed, all cache associated with it die**, and the cache manager also closed down. **Check section 9.2.1 for the working of Second-level cache.**

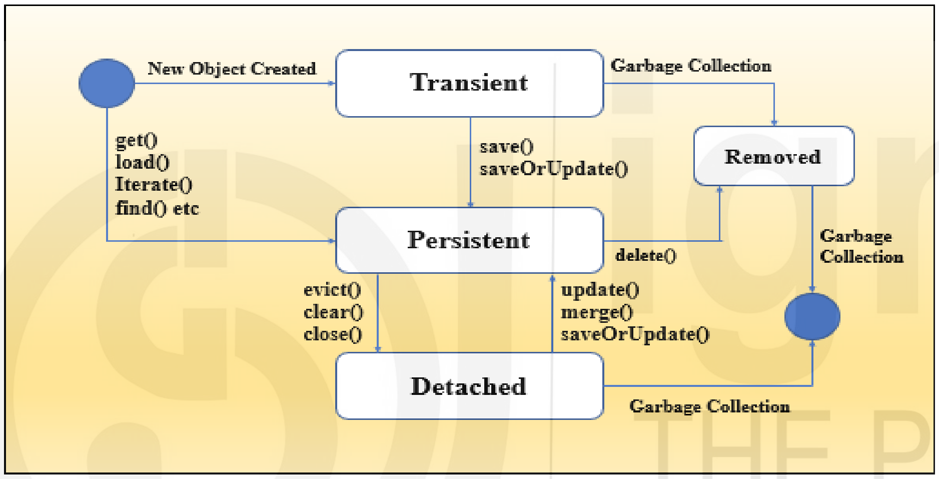
**Check Your Progress 2**

1) The types in the java.sql package such as java.sql.Date, java.sql.Time, java.sql.Timestamp are in line with SQL data types. Thus, its mapping is straightforward and either the **@basic** or **@column** annotation can be used. The type java.util.Date contains both date and time information. This is not directly related to any SQL data type. For this reason, another annotation is required to specify the desired SQL type. **@Temporal** annotation is used to specify the desired SQL data type. It has a single element *TemporalType.* TemporalType can be DATE, TIME or TIMESTAMP. The type of java.util.Calendar also requires **@Temporal** annotation to map with the corresponding SQL data type.
2) A Persistent class should follow the following guidelines:
   - **Prefer non-final class:** Hibernate uses proxies to apply some performance optimization. JPA entity or Hibernate Persistent class as final limits the ability of hibernate to use proxies. Without proxies, application loses lazy loading, which will cost performance.
   - **A no-arg constructor:** Hibernate creates an instance of a persistent class using newInstance() method. Thus, a persistent class should have default constructor at least package visibility.
   - **Identifier Property:** Each JPA entity should have a primary Key. Thus, an attribute should be annotated with @Id annotation.
   - **Getter and Setter methods:** Hibernate recognizes the method by getter and setter method names by default.

3) Hibernate works with POJO. Without any Hibernate specific annotation and mapping, Hibernate does not recognize these POJOs. Once properly annotated with required annotation, hibernate identifies them and keeps track of them to perform database operations such as create, read, update and delete. These POJOs are considered as mapped with Hibernate. An instance of a class mapped with Hibernate, can be any of the four persistence states which are known as hibernate entity lifecycle states.
**Same as Figure 10.5: Hibernate Entity Lifecycle States**

1. Transient
2. Persistent
3. Detached
4. Removed

4) Check section 10.4.3



## Check Your Progress 3

1) **Difference between load() and get() methods**
Although load() and get() methods perform the same task, still the differences exist in their return values in case the identifier does not exist in the database.

... The get() method returns NULL if the identifier does not exist.

... The load() method throws a runtime exception if the identifier does not exist.

2) Differences between save() and persist() methods are listedin the below table.

| S.NO | Key | Save() | Persist() |
|---|---|---|---|
| 1 | Basic | It stores object in database | It also stores object in database |
| 2 | Transaction Boundaries | It can save object within boundaries and outside boundaries | It can only save object within the transaction boundaries |
| 3 | Return Type | It returns generated id and return type is serializable | It does not return anything. Its return type is void. |

| | | | |
|---|---|---|---|
| 4 | Detached Object | It will create a new row in the table for detached object | It will throw persistence exception for detached object |
| 5 | Supported by | It is only supported by Hibernate | It is also supported by JPA |

3) This saveOrUpdate() method either performs**save()** or **update()** based on identifier existence. E.g. If an identifierexists for the instance, update() method will be called, otherwise save() will be performed. The saveOrUpdate() method handles the cases where we need to save a detached instance. Unlike the save() operation on detached instances, the saveOrUpdate() method does not result in a duplicate record. Similar to update(), this method is used to reattach an instance to the session. This method can be considered as a universal tool to make an object persistent regardless of its state,whether it is transient or detached.

4) Spring Data JPA is one of the many sub-projects of Spring Data which simplifies the data access for relational data stores. Spring Data JPA is not a JPA provider; instead it wraps the JPA provider and adds its own features like a no-code implementation of the repository pattern. Spring Data JPA uses Hibernate as the default JPA provider. JPA provider is configurable, and other providers can also be used with Spring Data JPA. Spring Data JPA provides complete abstraction over the DAO layer into the project. The advantages of using Spring Data JPA are as follows:

   **DAO Abstraction with No-Code Repositories**

   Spring Data JPA defines many Repository Interfaces such as CrudRepository, PagingAndSortingRepository, JpaRepository having methods to store, retrieve, sorted retrieval, paginated result and many more.With Spring Data JPA, we don't have to write SQL statements; instead, we just need to extend the interface defined by Spring Data JPA for one of the entities.

   **Query Methods**
   Another robust and comfortable feature of Spring Data JPA is the Query Methods. Based on the name of methods declared in the repository interfaces are converted to low level SQL queries at runtime.

   **Seamless Integration**
   Spring Data JPA seamlessly integrates ***JPA*** into the Spring stack, and its repositories reduce the boilerplate code required to the JPA specification. Spring Data JPA also helps the DAO layer integration and interaction with other Spring based components in your application, like accessing property configurations or auto-wiring the repositories into the application service layer. It also works perfectly with Spring Boot auto-configuration.

## 10.8 REFERENCES/FURTHER READING

- Craig Walls, "Spring Boot in action" Manning Publications, 2016. (https://doc.lagout.org/programmation/Spring%20Boot%20in%20Action.pdf)

- Christian Bauer, Gavin King, and Gary Gregory, "**J**ava Persistence with Hibernate",Manning Publications, 2015.
- Ethan Marcotte, "Responsive Web Design", Jeffrey Zeldman Publication, 2011(http://nadin.miem.edu.ru/images_2015/responsive-web-design-2nd-edition.pdf)
- Tomcy John, "Hands-On Spring Security 5 for Reactive Applications",Packt Publishing,2018
- https://docs.jboss.org/hibernate/orm/3.5/reference/en/html/tutorial.html
- https://www.w3spoint.com/get-vs-load-hibernate
- https://www.baeldung.com/learn-jpa-hibernate
- https://www.javatpoint.com/hibernate-with-annotation
- https://www.tutorialspoint.com/jpa/index.htm
- https://howtodoinjava.com/hibernate/hibernate-jpa-2-persistence-annotations-tutorial/
- https://docs.jboss.org/hibernate/orm/3.6/quickstart/en-US/html/hibernate-gsg-tutorial-jpa.html
- https://www.baeldung.com/hibernate-save-persist-update-merge-saveorupdate
- https://www.journaldev.com/3472/hibernate-session-get-vs-load-difference-with-examples
- https://www.baeldung.com/the-persistence-layer-with-spring-data-jpa
- https://dzone.com/articles/spring-data-jpa

# UNIT 11 CRUD APPLICATION USING SPRING BOOT AND HIBERNATE

**Structure** **Page No.**

11.0 Introduction
11.1 Objectives
11.2 Spring Data Repository
  11.2.1 CrudRepository
  11.2.2 PagingAndSortingRepository
  11.2.3 JpaRepository
11.3 Hibernate Association Mappings
  11.3.1 One-to-Many / Many-to-One
  11.3.1.1 Owning Side and Bi-directionality
    11.3.1.2 Eager vs Lazy loading
    11.3.1.3 Cascading Operations
  11.3.2 One-to-One
    11.3.2.1 Implementation with Foreign key
    11.3.2.2 Implementation with shared Primary key
  11.3.3 Many-to-Many
11.4 Create records using Spring Boot and Hibernate
1.5 Read Records Using Spring Boot and Hibernate
11.6 Update records using Spring Boot and Hibernate
11.7 Delete records using Spring Boot and Hibernate
11.8 Summary
11.9 Solutions/ Answer to Check Your Progress
11.10 References/Further Reading

## 11.0 INTRODUCTION

Nowadays, there are varieties of data stores such as relational databases, NoSQL databases like MongoDB, Casandraetc, Big Data solutions such as Hadoop etc. Different storage technologies have different configurations, different methods and different API's to fetch the data.

Spring Data simplifies and makes homogeneous data access technologies for relational and non-relational databases, cloud-based data services and map-reduce frameworks. Spring Data provides an abstraction that allows us to connect in the same way to relational databases and NoSQL databases. Thus, switching can be done effortlessly between data stores.

**Spring Data JPA** is one of the many sub-projects of **Spring Data**that focuses on simplification of the data access for relational data stores. It is not a JPA provider; instead, it wraps the JPA provider and adds its own features like a **no-code implementation** of the repository pattern. Hibernate is the default JPA provider into Spring Data JPA. Spring Data JPA is very flexible, and other providers can also be configured as JPA providers. Spring Data JPA provides a complete abstraction over the DAO layer into the project.

The actual strength of Spring Data JPA lies in the ***repository abstraction***. The ***repository abstraction*** enables us to write the business logic code on a higher abstraction level. Developers do not need to worry about DAO layer implementations;instead they just need to learn Spring Data repository interfaces.

Spring Data provides an implementation for these repository interfaces out-of-the-box.

The Hibernate mappings are one of the key features of Hibernate. Association in Hibernate tells the relationship between the objects of POJO classes, i.e. how the entities are related to each other. The established relationship can be either unidirectional or bidirectional. The supported relationships are similar to the database relationships such as **One-to-One, Many-to-Many, Many-to-One/One-to-Many**.

## 11.1 OBJECTIVES

After going through this unit, you will be able to:

- describe Spring Data Repository such as CrudRepository, PagingAndSortingRepository and JpaRepository,
- elaborate Hibernate association mappings,
- describe Hibernate Performance optimization via Eager and Lazy loading,
- demonstrate Hibernate Cascading and its impact,
- create records using Spring Data JPA,
- get records using Spring Data JPA,
- update record using Spring Data JPA, and
- delete record using Spring Data JPA

## 11.2 SPRING DATA REPOSITORY

The previous unit gave us the overview of Spring Data JPA. This section explains the Spring Data Repository to create the persistence layer into the Spring Boot application.

The ***Spring Data repository*** aims to reduce the boilerplate code required to implement the DAO layer into a project for various persistence stores. Spring Data repository has different interfaces, each having different functionality. Thefollowing interfaces in the Spring Data repository have been explained.

- CrudRepository
- PagingAndSortingRepository
- JpaRepository

JpaRepository extends PagingAndSortingRepository interface, which extends CrudRepository. Thus, **JpaRepository contains all API of CrudRepositoty and PagingAndSortingRepository.** The persistent layer can extend any of the above interfaces based on required APIs. The implementation of persistent layers is very easy and is shown asan example.

```
@Entity
publicclass Book
{
	@Id
	@GeneratedValue(strategy = GenerationType.IDENTITY)
	privateintid;

	private String title;
// getter setter
}
```

Following Repository provides a simple operation – find a Book based on its title.

```
@Repository
publicinterfaceBookRepositoryextendsCrudRepository<Book, Long>
{
publicBook findByTitle(String bookTitle);
}
```

The Spring Data Repository will auto-generate the implementation based on the method name. Supported keywords inside method name can be found at - https://docs.spring.io/spring-data/data-jpa/docs/current/reference/html/#jpa.query-methods.query-creation

## 11.2.1 CrudRepository

This interface provides all basic CRUD operation related APIs. The methods provided in CrudRepository are shown in the code.

```
publicinterfaceCrudRepository<T, ID>extends Repository<T, ID>
{

        <S extends T> S save(S entity);

        <S extends T>Iterable<S>saveAll(Iterable<S>entities);

        Optional<T>findById(ID id);

        booleanexistsById(IDid);

        Iterable<T>findAll();

        Iterable<T>findAllById(Iterable<ID>ids);

        long count();

        voiddeleteById(ID id);

        void delete(T entity);

        voiddeleteAll(Iterable<? extends T>entities);

        voiddeleteAll();
}
```

- save(…) - Saves a given entity.
- saveAll(…) - Saves all given entities.
- findById(…) - Retrieves an entity by its id.
- existsById(…) - Returns whether an entity with the given id exists.
- findAll(…) - Returns all instances of the type.
- findAllById(…) - Returns all instances of the type with the given IDs
- count(…) - Returns the number of entities available.
- deleteById(…) - Deletes the entity with the given id.
- delete(…) - Deletes a given entity.
- deleteAll(…) - Deletes the given entities.
- deleteAll() - Deletes all entities managed by the repository.

### 11.2.2 PagingAndSortingRepository

PagingAndSortingRepository interface extends CrudRepository interface. It provides the paging and sorting feature apart from CRUD feature. The methods in the interface are shown in the code.

```
publicinterfacePagingAndSortingRepository<T, ID>extendsCrudRepository<T, ID>
{
        Iterable<T>findAll(Sort sort);

        Page<T>findAll(Pageablepageable);
}
```

- findAll(Sort sort) - Returns all entities sorted by the given options.
- findAll(Pageablepageable) - Returns a page of entities meeting the paging restriction provided in the pageable object.

### 11.2.3 JpaRepository

JpaRepository extends PagingAndSortingRepository interface, which extends CrudRepository. Thus, **JpaRepository contains all API of CrudRepositoty and PagingAndSortingRepository.** The methods available in JpaRepository are shown in the code.

```
publicinterfaceJpaRepository<T, ID>extendsPagingAndSortingRepository<T, ID>,
QueryByExampleExecutor<T>
{

        @Override
        List<T>findAll();

        @Override
        List<T>findAll(Sort sort);

        @Override
        List<T>findAllById(Iterable<ID>ids);

        @Override
        <S extends T> List<S>saveAll(Iterable<S>entities);

        void flush();

        <S extends T> S saveAndFlush(S entity);

        voiddeleteInBatch(Iterable<T>entities);

        voiddeleteAllInBatch();

        T getOne(ID id);

        @Override
        <S extends T> List<S>findAll(Example<S>example);

        @Override
        <S extends T> List<S>findAll(Example<S>example, Sort sort);
}
```

**✒Check Your Progress 1:**

1) What is the relationship between JPA, Hibernate and Spring data JPA?

........................................................................................................................

........................................................................................................................

........................................................................................................................

........................................................................................................................

2) What is Spring Data?

........................................................................................................................

........................................................................................................................

........................................................................................................................

........................................................................................................................

3) What is the difference between CrudRepository and JpaRepository? Which canextend and when?

........................................................................................................................

........................................................................................................................

........................................................................................................................

........................................................................................................................

## 11.3 HIBERNATE ASSOCIATION MAPPINGS

The previous unit has explained that hibernate can identify POJO classes as persistent only when they are annotated with certain annotations. While making the POJO classes as persistent entities using JPA annotations, we may face situations where two entities can be related and must be referenced from each other, either uni-directional or bi-directional. Association mappings are one of the key features of JPA and Hibernate. That establishes the relationship between two database tables as attributes in your model and allows you to easily navigate the association in your model and JPQL or criteria queries.

When only one pair of entities contains a reference to the other, the association is **unidirectional**. If the pair of entities contains a reference to each other, then it is referred to as **bi-directional**. Entities can contain references to other entities, either directly as an embedded property or field or indirectly via a collection of some sort (arrays, sets, lists, etc.). There is no impact of unidirectional or bidirectional association on database mapping. **Foreign Key Relationships** are used to represent the associations in the underlying tables.

JPA and Hibernate have the same association as you are aware of relational databases. The followings are the association that will be described in subsequent sections.

- one-to-many/many-to-one
- one-to-one
- many-to-many

### 11.3.1 One-to-Many / Many-to-One

One-to-Many and Many-to-One relationships are the opposite sides of the same coin and closely related. **One-to-Many** mapping is described as one row in a table and is mapped to multiple rows in another table.

For the illustration of the One-to-Many relationship, the **Teacher** and their **Courses** will be taken. A teacher can give multiple courses but a course which is given by only one teacher is an example of one-to-many relationship. While another perspective, many courses are given by a teacher, is an example of the many-to-one relationship. Before diving into details of how to map this relationship, let's create the entities.

```
@Entity
publicclass Teacher
{
        @Id
        @GeneratedValue(strategy = GenerationType.IDENTITY)
        private Long id;
        private String name;

}


@Entity
publicclass Course
{
        @Id
        @GeneratedValue(strategy = GenerationType.IDENTITY)
        privateintid;
        private String title;

}
```

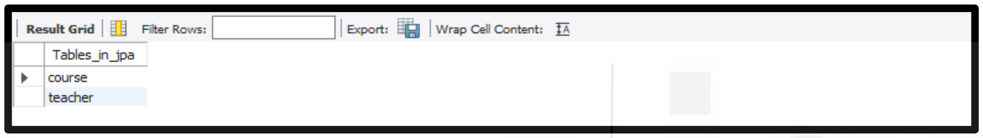
Now, the **Teacher** class should include a list of courses, and mapping will be required to map this relationship into the database. This will be annotated with a @OnetoMany annotation.

```
@Entity
publicclass Teacher
{

        @Id
        @GeneratedValue(strategy = GenerationType.IDENTITY)
        private Long id;

        private String name;

        @OneToMany(cascade = CascadeType.ALL)
        private List<Course>courses;
}
```

With the above configuration, Hibernate uses an association table to map the relationship. With the above configuration, three tables named ***teacher***, ***course*** and ***teacher_courses*** will be created. But, definitely, we are not looking for such mapping. In order to avoid the third table teacher_courses, **@JoinColumn** annotation is used to specify the foreign key column.

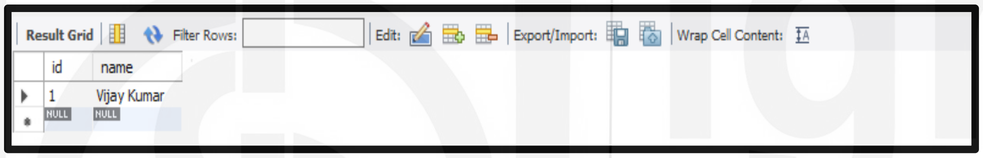
```
@Entity
publicclass Teacher
{

            @Id
            @GeneratedValue(strategy = GenerationType.IDENTITY)
            private Long id;

            private String name;

            @OneToMany(cascade = CascadeType.ALL)
@JoinColumn(name = "teacher_id", referencedColumnName = "id")
            private List<Course>courses;
}
```

The tables created into the database and save() operation on teacher object will result in the following into the database (as shown in Figure 11.1, 11.2 and 11.3):



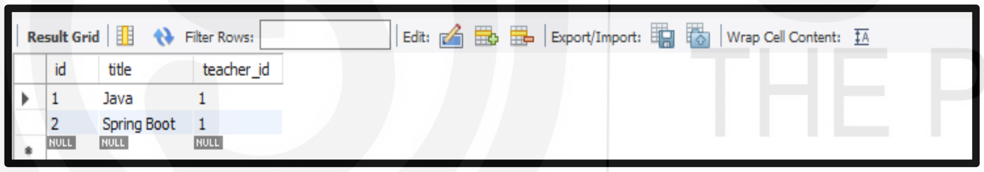
| Tables_in_jpa |
|---|
| course |
| teacher |

Figure11.1: Hibernate Created Tables



| id | name |
|---|---|
| 1 | Vijay Kumar |
| NULL | NULL |

Figure11.2: Hibernate Created Records Into teacher Table



| id | title | teacher_id |
|---|---|---|
| 1 | Java | 1 |
| 2 | Spring Boot | 1 |
| NULL | NULL | NULL |

Figure11.3: Hibernate Created Records Into course Table

### 11.3.1.1 Owning Side and Bi-directionality

The previous example has defined the Teacher class as the owning side of the One-to-Many relationship. This is because the Teacher class defines the join column between the two tables. The **Course** is called the referencing side in the relationship.

Course class can be made as to the owning side of the relationship by mapping the Teacher field with @ManytoOne in the course class instead.

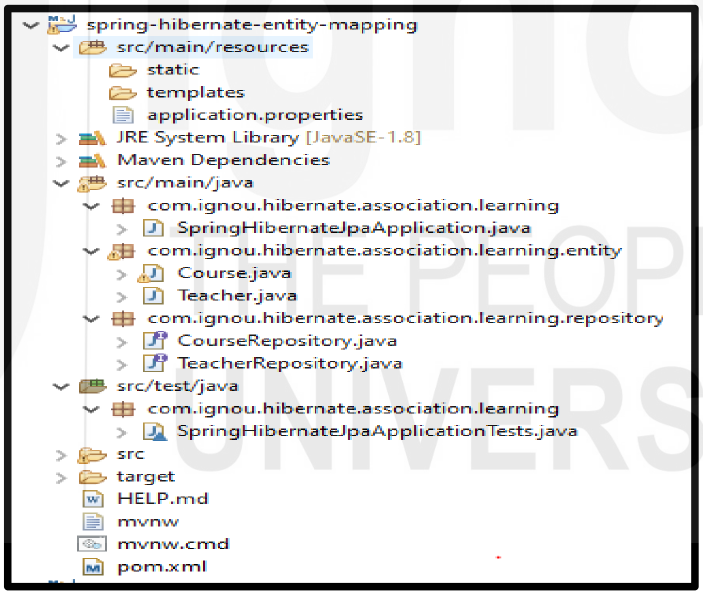
```
@Entity
publicclass Teacher
{
            @Id
            @GeneratedValue(strategy = GenerationType.IDENTITY)
            private Long id;
            private String name;

}
@Entity
publicclassCourse
{
            @Id
            @GeneratedValue(strategy = GenerationType.IDENTITY)
            privateintid;
            private String title;

            @ManyToOne(cascade = CascadeType.ALL)
            @JoinColumn(name = "teacher_id", referencedColumnName = "id")
            private Teacher teacher;

}
```

This time **@ManytoOne** annotation has been used instead of **@OnetoMany** annotation. In the above example, it can be observed that the Teacher class does not have a list of courses. Uni-directional relationship has been used here; thus, only one of the entities has reference to another entity. **@ManytoOne** will result similar to **@OnetoMany**.

***Note:*** *It's a good practice to put the owning side of a relationship in the class/table where the foreign key will be held.*

According to best practice, the second version of code using **@ManytoOne**is better. If you look at the second version of the code, the Teacher class does not offer to access the list of courses. This can be achieved by the bidirectional relationship.

The following section explains the bi-directional one-to-many mapping between teacher and courses with a complete example. The Spring Hibernate project structure is shown in Figure 11.4 and the Source Code structure follows;



```
spring-hibernate-entity-mapping
  src/main/resources
    static
    templates
    application.properties
  JRE System Library [JavaSE-1.8]
  Maven Dependencies
  src/main/java
    com.ignou.hibernate.association.learning
      SpringHibernateJpaApplication.java
    com.ignou.hibernate.association.learning.entity
      Course.java
      Teacher.java
    com.ignou.hibernate.association.learning.repository
      CourseRepository.java
      TeacherRepository.java
  src/test/java
    com.ignou.hibernate.association.learning
      SpringHibernateJpaApplicationTests.java
  src
  target
  HELP.md
  mvnw
  mvnw.cmd
  pom.xml
```

**Figure11.4: Spring Hibernate Project Structure**

Teacher.java

```
packagecom.ignou.hibernate.association.learning.entity;

@Entity
publicclass Teacher
{
            @Id
            @GeneratedValue(strategy = GenerationType.IDENTITY)
            private Long id;

            private String name;

            @OneToMany(mappedBy = "teacher", cascade= CascadeType.ALL, fetch =
FetchType.EAGER)
        private List<Course>courses;

        public Teacher() {}

        public Teacher(String name)
        {
                super();
                this.name = name;
        }

        public Long getId()
        {
                returnid;
        }

        publicvoidsetId(Long id)
        {
                this.id = id;
        }

        public String getName()
        {
                returnname;
        }

        publicvoidsetName(String name)
        {
                this.name = name;
        }

        public List<Course>getCourses()
        {
                returncourses;
        }

        publicvoidsetCourses(List<Course>courses)
        {
                this.courses = courses;
        }

        @Override
        public String toString()
        {
        return"Teacher [id=" + id + ", name=" + name + ", courses=" + courses + "]";
        }

}
```

The Teacher class has reference courses. **Teacher** and **Course** has a relationship as one-to-many, and thus, it is annotated with **@OnetoMany. @OnetoMany**annotation has an attribute **mappedBy.** Without mappedBy attribute, we would not have a two-way relationship. The ***mappedBy*** attributes indicatesthe JPA that the field is already mapped by another entity. It's mapped by the teacher field of the Course entity. Other attributes such as cascade and fetch will be explained in the next section.

**Course.java**

```
packagecom.ignou.hibernate.association.learning.entity;

@Entity
publicclass Course
{
        @Id
        @GeneratedValue(strategy = GenerationType.IDENTITY)
        privateintid;
        private String title;

        @ManyToOne
        @JoinColumn(name = "teacher_id", referencedColumnName = "id")
        private Teacher teacher;

        public Course()
        {

        }
        public Course(String title)
        {
                this.title = title;
        }
        publicintgetId()
        {
                returnid;
        }
        publicvoidsetId(intid)
        {
                this.id = id;
        }
        public String getTitle()
        {
                returntitle;
        }
        publicvoidsetTitle(String title)
        {
                this.title = title;
        }
        public Teacher getTeacher()
        {
                returnteacher;
        }
        publicvoidsetTeacher(Teacher teacher)
        {
                this.teacher = teacher;
        }
        @Override
        public String toString()
        {
                return"Course [id=" + id + ", title=" + title + "]";
        }
}
```

Course class is having the reference of Teacher. **@ManytoOne** annotation is used since the Course and Teacher are having many to one relationship. Course class is the owning side since @JoinColumn is defined in this class.

**TeacherRepository.java**

```
packagecom.ignou.hibernate.association.learning.repository;

publicinterfaceTeacherRepositoryextendsCrudRepository<Teacher, Long>
{
        // No code repository. Spring data JPA will provide the implementation.
}
```

**CourseRepository.java**

```
packagecom.ignou.hibernate.association.learning.repository;

publicinterfaceCourseRepositoryextendsCrudRepository<Course, Long>
{
        // No code repository. Spring data JPA will provide the implementation.
}
```

**SpringHibernateJpaApplication.java**

```
packagecom.ignou.hibernate.association.learning;

@SpringBootApplication
publicclassSpringHibernateJpaApplicationimplementsCommandLineRunner
{
@Autowired
privateTeacherRepositoryteacherRepo;

publicstaticvoid main(String[] args)
{
SpringApplication.run(SpringHibernateJpaApplication.class, args);
   }

@Override
publicvoidrun(String... args) throws Exception
{
    Teacher t1 = new Teacher("Vijay Kumar");

    Course c1 = new Course("Java");
    Course c2 = new Course("Spring Boot");

    List<Course>courseList = newArrayList<Course>();
courseList.add(c1);
courseList.add(c2);

    t1.setCourses(courseList);
    c1.setTeacher(t1);
    c2.setTeacher(t1);

    t1 = teacherRepo.save(t1);
   Teacher t = teacherRepo.findById(t1.getId()).get();
System.out.println(t);

}
}
```

### 11.3.1.2 Eager vs Lazy loading

It is worth noting that a mapped relationship should not impact the software's memory by putting too many unnecessary entities. For example, consider that the Course is a heavy object, i.e. having too many attributes. For some business logic, all **Teacher** objects are loaded from the database. Business logic does not need to retrieve or use courses in it but **Course** objects are still being loaded alongside the **Teacher** objects.

Such loading will degrade the performance of applications. Technically, this can be solved by using the Data Transfer Object Design Pattern and retrieving Teacher information *without* the courses.

JPA has considered all such problems ahead and made **One-to-Many relationships load *lazily* by default**. Lazy loading means relationships will be loaded when it is actually needed. The execution of **@OnetoMany** relationship with the default fetch type will issue two queries. First is for the Teacher object and the second for the Course objects when it's needed. The log of the query issued by Hibernate is as follows.

```
2021-03-06 17:52:47.095 DEBUG 8124 --- [  restartedMain] org.hibernate.SQL                        : select
teacher0_.id as id1_1_0_, teacher0_.name as name2_1_0_ from teacher teacher0_ where
teacher0_.id=?
2021-03-06 17:52:47.096 TRACE 8124 --- [  restartedMain] o.h.type.descriptor.sql.BasicBinder      :
binding parameter [1] as [BIGINT] - [1]
2021-03-06 17:52:47.104 TRACE 8124 --- [  restartedMain] o.h.type.descriptor.sql.BasicExtractor   :
extracted value ([name2_1_0_] : [VARCHAR]) - [Vijay Kumar]
2021-03-06 17:52:47.111 TRACE 8124 --- [  restartedMain] org.hibernate.type.CollectionType        :
Created collection wrapper: [com.ignou.hibernate.association.learning.entity.Teacher.courses#1]
2021-03-06 17:52:47.117 DEBUG 8124 --- [  restartedMain] org.hibernate.SQL                        : select
courses0_.teacher_id as teacher_3_0_0_, courses0_.id as id1_0_0_, courses0_.id as id1_0_1_,
courses0_.teacher_id as teacher_3_0_1_, courses0_.title as title2_0_1_ from course courses0_ where
courses0_.teacher_id=?
```

While the execution of **@OnetoMany** relationship with fetch type as ***Eager*** will issue only one query, Log of the query issued by Hibernate is as follows.

```
2021-03-06 18:28:11.107 DEBUG 15728 --- [  restartedMain] org.hibernate.SQL                       : select
teacher0_.id as id1_1_0_, teacher0_.name as name2_1_0_, courses1_.teacher_id as teacher_3_0_1_,
courses1_.id as id1_0_1_, courses1_.id as id1_0_2_, courses1_.teacher_id as teacher_3_0_2_,
courses1_.title as title2_0_2_ from teacher teacher0_ left outer join course courses1_ on
teacher0_.id=courses1_.teacher_id where teacher0_.id=?
2021-03-06 18:28:11.107 TRACE 15728 --- [  restartedMain] o.h.type.descriptor.sql.BasicBinder     :
binding parameter [1] as [BIGINT] - [1]
```

***Many-to-One*** **relationships are *eager* by default**.A eager relationship means all related entities are loaded at the same time. The default behavior of fetch type can be changed with fetch attribute as shown below-

```
@OneToMany(mappedBy = "teacher", cascade= CascadeType.ALL, fetch =
FetchType.EAGER)
private List<Course>courses;

@ManyToOne(fetch = FetchType.LAZY)
private Teacher teacher;
```

### 11.3.1.3 Cascading Operations

**JPA** operations affect the only entity on which it has been performed. These operations will not affect the other entities that are related to it. For example, In case of Person-Address relationship, the Address entity doesn't have any meaning of its own without a Person entity. Thus, on delete of Person entity, Address entity should also get deleted.
Cascading is about JPA actions involving one entity propagating to other entities via an association. JPA provides **javax.persistence.CascadeType** enumerated types that define the cascade operations. These cascading operations can be defined with any type of mapping i.e. One-to-One, One-to-Many, Many-to-One, Many-to-Many. JPA provides the following cascade type to perform cascading operations.

| Cascade Type | Description |
|---|---|
| ALL | CascadeType.ALL propagates **all operations** including hibernate specific from a parent to a child entity. |
| PERSIST | CascadeType.PERSIST propagates the **save()** or **persist()** operation to the related entities. |
| MERGE | CascadeType.MERGE propagates only the **merge()** operation to the related entities. |
| REFRESH | CascadeType.REFRESH propagates only the **refresh()** operation to the related entities. |
| REMOVE | CascadeType.REMOVE removes all related entities association when the owning entity is deleted. |
| DETACH | CascadeType.DETACH detaches all related entities if owning entity is detached. |

**Hibernate Cascade Type**

Hibernate provides three additional Cascade Types along with provided by JPA. Hibernate provides *org.hibernate.annotations.CascadeType*enumerated types that define the cascade operations.

... CascadeType.REPLICATE
... CascadeType.SAVE_UPDATE
... CascadeType.Lock

Clear understanding of **One-to-Many/Many-to-One** association, as explained above, will help you to understand other associations. The following section will explain the other association as bidirectional with example.

## 11.3.2 One-to-One

**One-to-One** mapping is described as one row in a table mapped to only one row in another table.

For the illustration of One-to-One relationship, The **Student** and the **Address** have been taken.

### 11.3.2.1 Implementation with Foreign key

ER diagram of foreign key based one-to-one mapping is shown in Figure 11.5. The address_id column in Student is the foreign key to the address.

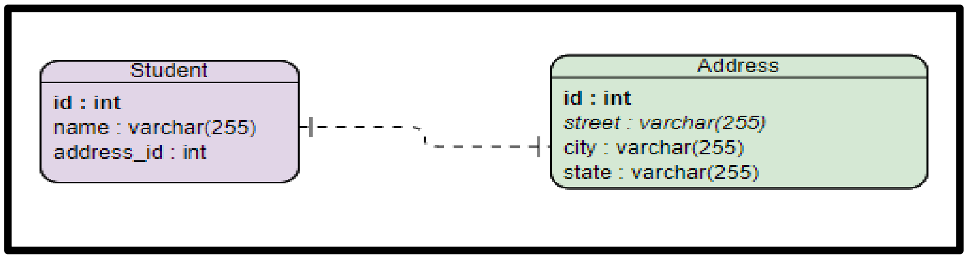


**Figure 11.5: ER Diagram For One-to-One Mapping With Foreign Key**

```
@Entity
publicclass Student
{

        @Id
        @GeneratedValue(strategy = GenerationType.IDENTITY)
```

```
            private Long id;

            private String name;
            @OneToOne(cascade = CascadeType.ALL)
            @JoinColumn(name="address_id", referencedColumnName = "id")
            private Address address;

              // getter setter
}
```

The Student and the Address has a one-to-one relationship. Entity Student comprises another entity Address, and it is annotated with **@OnetoOne** annotation. Check the cascade attribute into the annotation. Student entity is the owner, and thus, it has **@JoinColumn** annotation on Address entity. @JoinColumn is to configure the name of the column in the Student table that maps to the primary key in the address table. If the name attribute is not provided in @JoinColumn, Hibernate will follow some rules to select the default name of the column.

```
@Entity
publicclass Address
{

            @Id
            @GeneratedValue(strategy = GenerationType.IDENTITY)
            privateintid;
            private String street;
            private String state;
            private String pincode;
            private String country;
            @OneToOne(mappedBy = "address")
            private Student student;

               // getter setter

}
```

The Address entity is having @OneToOne annotation on another entity Student with attribute mappedBy. The mappedBy attribute is used since the Address is non-owning. Note that on both entities @OneToOne annotations are used since its bidirectional.

### 11.3.2.2 Implementation with shared Primary key

In this strategy, instead of creating a new column *address_id*, **we'll mark the primary key column (*student_id*) of the *address* table as the foreign key to the *student* table**. ER diagram is shown in Figure 11.6. This strategy optimizes the storage space.

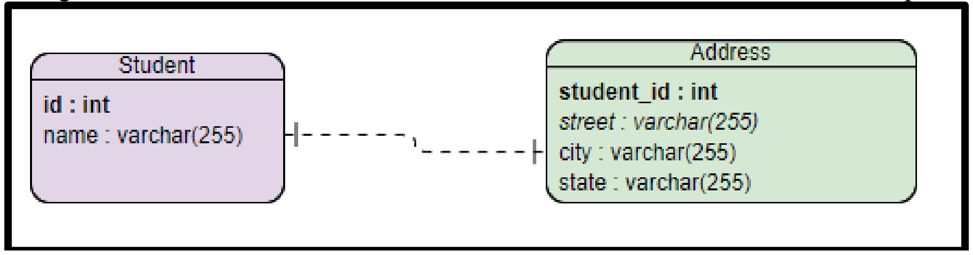


**Figure 11.6: ER Diagram For One-to-One Mapping With Shared Primary Key**

```
@Entity
publicclass Student
{

        @Id
        @GeneratedValue(strategy = GenerationType.IDENTITY)
        private Long id;

        private String name;

        @OneToOne(mappedBy = "student",  cascade = CascadeType.ALL)
        @PrimaryKeyJoinColumn
        private Address address;

          // getter setter
}
```

The **@PrimaryKeyJoinColumn**annotation indicates that the primary key of Student entity is used as the foreign key value for the associated Address entity. The mappedBy attribute is used here because the foreign key is now present in the address table.

```
@Entity

publicclass Address

{


        @Id

        @Column(name ="student_id")

        privateintid;

        private String street;

        private String state;

        private String pincode;

        private String country;


        @OneToOne

          @MapsId

          @JoinColumn(name="student_id")

        private Student student;


          // getter setter

}
```

The identifier of the Address entity is still having @Id annotation but it no longer uses @GeneratedValue annotation. Instead it references the student_id column. The **@MapsId**indicates that the primary key values will be copied from the Student entity.

## 11.3.3 Many-to-Many

**Many-to-Many** mapping is described as many rows in a table mapped to many rows in another table.

For the illustration of Many-to-Many relationship, **Book** and the **Author** have been taken. A book may be written by multiple writers and a writer will write multiple books. Thus, the Book and Author relationship is many-to-many and is shown as an ER diagram in Figure 11.7.

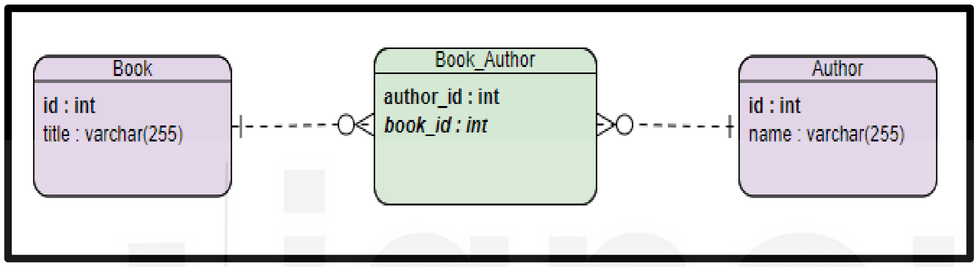


**Figure11.7: ER Diagram For Many-to-Many Relationship**

The JPA implementation for many-to-many mapping for Book as owning side is shown below:

```
@Entity
@Table(name = "books")
publicclass Book
{

          @Id
          @GeneratedValue(strategy = GenerationType.IDENTITY)
          private Long id;

//Other fields

          @ManyToMany(cascade = CascadeType.ALL)
          @JoinTable(name = "book_author" , joinColumns = @JoinColumn(name =
"book_id" ), inverseJoinColumns = @JoinColumn(name = "author_id"))
          private List<Author>authors;

//getter setter

}
```

**@JoinTable and @JoinColumn**aren't mandatory annotations. These are used to customize the name of the table and column name. **@ManyToMany**annotation has been used for many-to-many mapping. **@JoinTable**annotationhas a name attribute which defines the name of the joining table. It also requires the foreign keys with the *@JoinColumn* annotations**. The *joinColumn* attribute will connect to the owner side of the relationship, and the *inverseJoinColumn* to the other side.**

```
@Entity
@Table(name = "authors")
publicclass Author
```

```
{
        @Id
        @GeneratedValue(strategy = GenerationType.IDENTITY)
        private Long id;

        @Column(name = "author_name")
        @Size(max = 100)
        @NotNull
        private String name;


        private Integer rating;

        @ManyToMany(mappedBy = "authors", cascade = CascadeType.ALL)
        private List<Book>books;
}
```

**☛Check Your Progress 2:**

1) Explain One-to-Many association in Hibernate. Write an example for uni-directional one-to-many mapping.

..............................................................................................................................

..............................................................................................................................

..............................................................................................................................

..............................................................................................................................

2) Explain Eager and Lazy loading in Hibernate. How does it improve the performance of hibernate applications?

..............................................................................................................................

..............................................................................................................................

..............................................................................................................................

..............................................................................................................................

3) What are the different Cascade types supported by JPA?

..............................................................................................................................

..............................................................................................................................

..............................................................................................................................

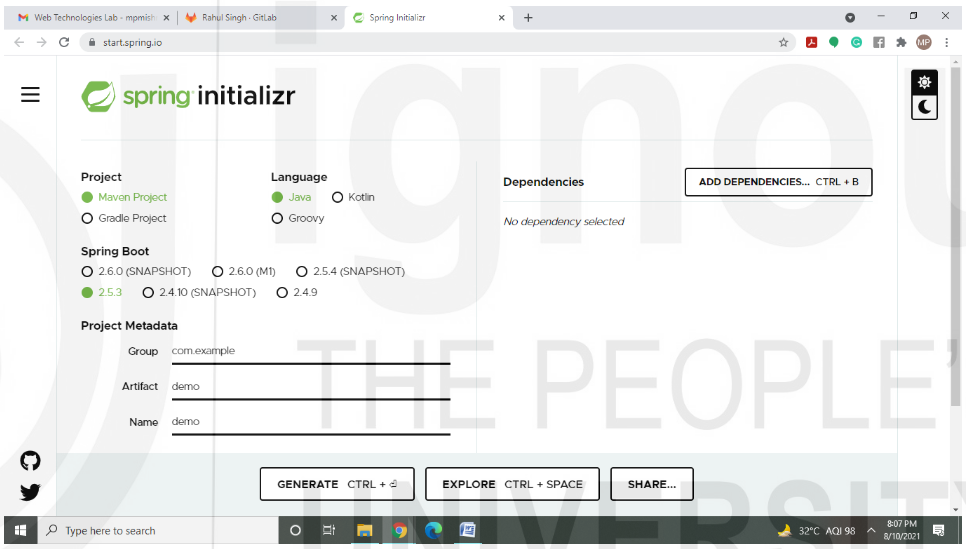
..............................................................................................................................

## 11.4 CREATE RECORDS USING SPRING BOOT AND HIBERNATE

The previous unit explained Spring Data JPA and hibernate as the default provider for Spring Data JPA. This unit has elaborated on Spring Data Repository and hibernate association mapping. This section provides the complete Rest Application to create records using Spring Boot and Hibernate by using all the explained concepts. The example considers the many-to-many relationship between **Book** and **Author.** Many books can be written by an Author and many authors can write a book. The same example will be considered for CRUD operation.The required tools and software are as follows:

- Eclipse IDE
- Maven
- Spring Boot
- Hibernate
- Mysql
- Postman

Visit Spring Initializr (https://start.spring.io) in order to generate the spring project (shown in Figure 11.8). Add the required dependencies and export the project into eclipse as maven project. Modify the data source properties into application.properties file and do some hands-on.



Project: [✓ Maven Project] Gradle Project

Language: [✓ Java] Kotlin Groovy

Spring Boot: 2.6.0 (SNAPSHOT) 2.6.0 (M1) 2.5.4 (SNAPSHOT) [✓ 2.5.3] 2.4.10 (SNAPSHOT) 2.4.9

Project Metadata

Group com.example

Artifact demo

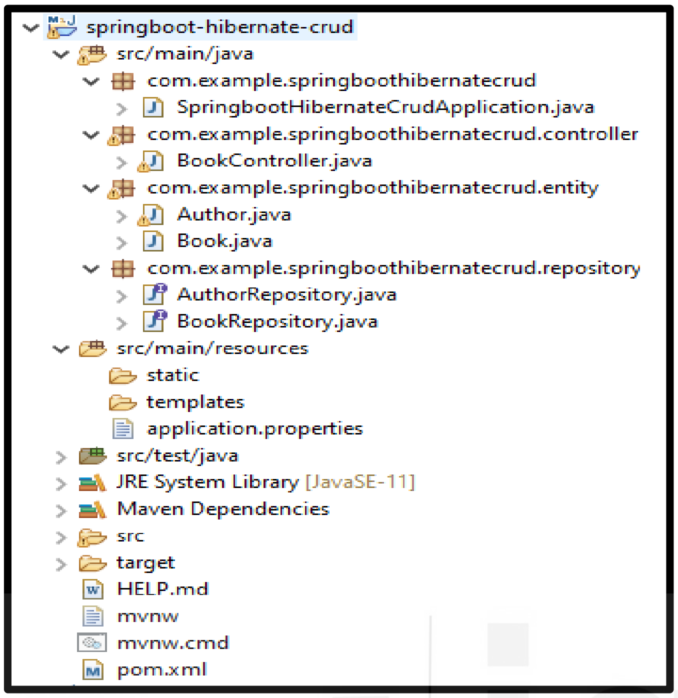
Name demo

Dependencies

*No dependency selected*

**Figure11.8: SpringBoot Project Setup Using Spring Initializr**

The source code structure is shown below.



```
springboot-hibernate-crud
  src/main/java
    com.example.springboothibernatecrud
      SpringbootHibernateCrudApplication.java
    com.example.springboothibernatecrud.controller
      BookController.java
    com.example.springboothibernatecrud.entity
      Author.java
      Book.java
    com.example.springboothibernatecrud.repository
      AuthorRepository.java
      BookRepository.java
  src/main/resources
    static
    templates
    application.properties
  src/test/java
  JRE System Library [JavaSE-11]
  Maven Dependencies
  src
  target
  HELP.md
  mvnw
  mvnw.cmd
  pom.xml
```

**Figure 11.9: SpingBootAnd Hibernate CRUD Applicatication Folder Structure**

**SpringbootHibernateCrudApplication.java** – This is the main class of Spring Boot application. To create the records at the time of application start, it implements CommandLineRunner interface and creates records using book repository.

```
@SpringBootApplication
@EnableJpaRepositories
publicclassSpringbootHibernateCrudApplicationimplementsCommandLineRunner
{
	@Autowired
	privateBookRepositorybookRepository;

	publicstaticvoid main(String[] args)
	{
		SpringApplication.run(SpringbootHibernateCrudApplication.class, args);
	}

	@Override
	publicvoid run(String... args) throws Exception
	{
		Book b1 = new Book("Java In Action", "ISBN-001");
		List<Book>bookList = newArrayList<Book>();
		bookList.add(b1);

		Author au1 = new Author("O. P. Sharma");
		Author au2 = new Author("K L Mishra");

		List<Author>authorList = newArrayList<Author>();
		authorList.add(au1);
		authorList.add(au2);

		b1.setAuthors(authorList);
		au1.setBooks(bookList);
		au2.setBooks(bookList);

		bookRepository.save(b1);
```

```
                Book b2 = new Book("Java 8 In Action", "ISBN-002");
                List<Book>bookList2 = newArrayList<Book>();
                bookList2.add(b2);

                Author au3 = new Author("R K Singh");
                Author au4 = new Author("A K Sharma");

                List<Author>authorList2 = newArrayList<Author>();
                authorList2.add(au3);
                authorList2.add(au4);

                b2.setAuthors(authorList2);
                au3.setBooks(bookList2);
                au4.setBooks(bookList2);

                bookRepository.save(b2);

        }
}
```

**BookController.java –** The ***BookController*** is responsible to handle the request from client for the URL pattern **/books/***. The controller is having a method ***createBookRecord*** which is annotated with ***@PostMapping*** which means it will accept the post request for the URL pattern **/books** with the request body having a valid Book entity json.

```
@RestController
@RequestMapping("/books")
publicclassBookController
{
        @Autowired
        privateBookRepositorybookRepository;
        @Autowired
        privateAuthorRepositoryauthorRepository;
        @PostMapping
        public Book createBookRecord(@Valid@RequestBody Book b)
        {
                returnbookRepository.save(b);
        }
}
```

In the example, there are two entities Book and Author, having a many-to-many relationship. Entities with bidirectional many-to-many associations are shown in the code.

**Book.java**

```
@Entity
@Table(name = "books")
publicclass Book
{
        @Id
        @GeneratedValue(strategy = GenerationType.IDENTITY)
        private Long id;

        @NotNull
        @Size(max=75)
        private String title;

        @NotNull
        @Size(max = 100)
        private String isbn;

        @ManyToMany(cascade = CascadeType.ALL)
        @JoinTable(joinColumns = @JoinColumn(name = "book_id" ), inverseJoinColumns
        = @JoinColumn(name = "author_id"))
        private List<Author>authors;

        public Book() {}

        public Book(@NotNull@Size(max = 75) String title, @NotNull@Size(max = 100)
        String isbn)
        {
                this.title = title;
                this.isbn = isbn;
        }

        // getter setter

        @Override
        public String toString()
        {
        return"Book [id=" + id + ", title=" + title + ", isbn=" + isbn + ", authors=" + authors
        + "]";
        }
}
```

**Author.java**

```
@Entity
@Table(name = "authors")
publicclass Author
{

        @Id
        @GeneratedValue(strategy = GenerationType.IDENTITY)
        private Long id;

        @Column(name = "author_name")
        @Size(max = 100)
        @NotNull
        private String name;

        private Integer rating;

        @ManyToMany(mappedBy = "authors", cascade = CascadeType.ALL)
        @JsonIgnore
        private List<Book>books;

        public Author() {}

        public Author(@Size(max = 100) @NotNull String name)
        {
```

```
                    super();
                    this.name = name;
          }

          // getter setter

          @Override
          public String toString()
          {
                    return"Author [id=" + id + ", name=" + name + ", rating=" + rating + "]";
          }
}
```

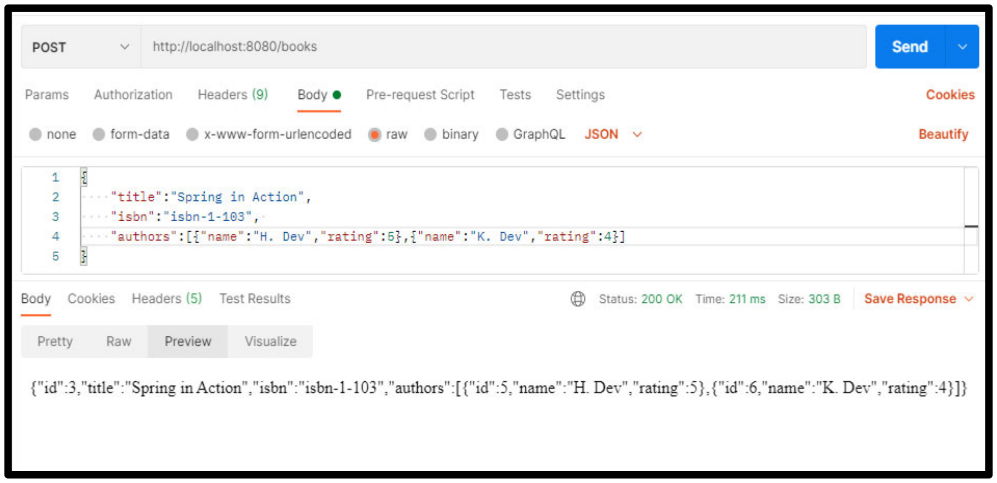
**BookRepository.java** – BookRepository provides the methods to interact with databases to perform CRUD operation. BookRepository extends CrudRepository interface. To create the records, the default save() method of CrudRepository will be used.

```
@Repository
publicinterfaceBookRepositoryextendsCrudRepository<Book, Long>
{
   // No implementation is required.
}
```

**application.properties**

```
# DATASOURCE (DataSourceAutoConfiguration&DataSourceProperties)
spring.datasource.url=jdbc:mysql://localhost:3306/test?useSSL=false&serverTimezone=UTC
&useLegacyDatetimeCode=false
spring.datasource.username=root
spring.datasource.password=root

# Hibernate
# The SQL dialect makes Hibernate generate better SQL for the chosen database
spring.jpa.properties.hibernate.dialect = org.hibernate.dialect.MySQL5InnoDBDialect

# Hibernateddl auto (create, create-drop, validate, update)
spring.jpa.hibernate.ddl-auto = create

logging.level.org.hibernate.SQL=DEBUG
logging.level.org.hibernate.type=TRACE

spring.jpa.properties.hibernate.enable_lazy_load_no_trans=true
```

Start the rest application by executing **SpringbootHibernateCrudApplication.java.** Use the postman to send the post request for book record creation. Sample post request with postman for book record creation is shown in Figure 11.10. Check the response status as **200 OK** and response body with generated id for new record.



```
http://localhost:8080/books
```

```
{
    "title":"Spring in Action",
    "isbn":"isbn-1-103",
    "authors":[{"name":"H. Dev","rating":5},{"name":"K. Dev","rating":4}]
}
```

Status: 200 OK Time: 211 ms Size: 303 B

```
{"id":3,"title":"Spring in Action","isbn":"isbn-1-103","authors":[{"id":5,"name":"H. Dev","rating":5},{"id":6,"name":"K. Dev","rating":4}]}
```

**Figure 11.10: Create Record Using SpringBoot and Hibernate**

## 11.5 READ RECORDS USING SPRING BOOT AND HIBERNATE

This section explains how to read the records and search the records using Spring Boot and Spring Data JPA with Hibernate. The previous section example will be used to explain the read operation. We will just add the new feature into the existing rest application. Add the new features into BookRepository and BookController as shown below in order to test the read records operation.

**BookRepository.java**

```
@Repository
publicinterfaceBookRepositoryextendsCrudRepository<Book, Long>
{
        public List<Book>findByTitleContaining(String pattern);     // 1

        public List<Book>findByTitleContaininsIgnoreCase(String pattern); //2

        public List<Book>findByTitleLike(String pattern);           // 3

        @Query("Select b from Book b where b.title like %:pattern%")
        public List<Book>findByTitle(String pattern);               // 4

}
```

BookRepository extends the CrudRepository interface, which provides the methods to read all records, specific record by id etc. Here we have added three new methods in order to provide search features. All three methods generate the same ***like queries,*** but there are some differences.

1. Query methods using Containing, Contains, and IsContaining generate the like query. For example, the query method findByTitleContaining(String pattern) generates the following sql query.

   **SELECT * FROM books WHERE title LIKE '%<pattern>%'**

2. The ***findByTitleContainsIgnoreCase(String pattern)*** is similar to ***findByTitleContaining(String pattern)*** but with case insensitive.

3. Spring Data JPA also provides a Like keyword, but it behaves differently. An example to call the **findByTitle(…)** is shown inBookRepository code in the next paragraph.
4. Sometimes we need to create queries that are too complicated for Query Methods or would result in absurdly long method names. In those cases, we can use the @Query annotation to query our database.
   - Named Parameters (**:name**) – Query parameters can be passed as named parameters into the query. Check the example in the aboveBookRepository.
   - Ordinal Parameters (**?index**) **-** JPQL also supports ordinal parameters, whose form is **?index**. An example of ordinal parameters is as **"Select b from Book b where b.title like %?1%"**

**BookController.java**

```
@RestController
@RequestMapping("/books")
publicclassBookController
{
        @Autowired
        privateBookRepositorybookRepository;


        @GetMapping
        publicIterable<Book>getBooks()
        {
                returnbookRepository.findAll();
        }
        @GetMapping("/search-by-method-containing")
        public List<Book>searchByMethod(@RequestParam(value = "q", required = true)
        String pattern)
        {
                returnbookRepository.findByTitleContaining(pattern);
        }
        @GetMapping("/search-by-method-like")
        public List<Book>searchByMethodLike(@RequestParam(value = "q", required =
        true) String pattern)
        {
                returnbookRepository.findByTitleLike("%"+pattern+"%");
        }
        @GetMapping("/search-by-query")
        public List<Book>searchByQuery(@RequestParam(value = "q", required = true)
        String pattern)
        {
                returnbookRepository.findByTitle(pattern);
        }
}
```

The BookController provides the following **GET** endpoints to get all books and search the book with title keywords.

- http://localhost:8080/books - Returns all the books with its authors.
- http://localhost:8080/search-by-method-containing?q=Java – Returns all the books with title containing Java
- http://localhost:8080/search-by-method-like?q=Java – Returns all the books with title containing Java.
- http://localhost:8080/search-by-query?q=Java – Returns all the books with title containing Java

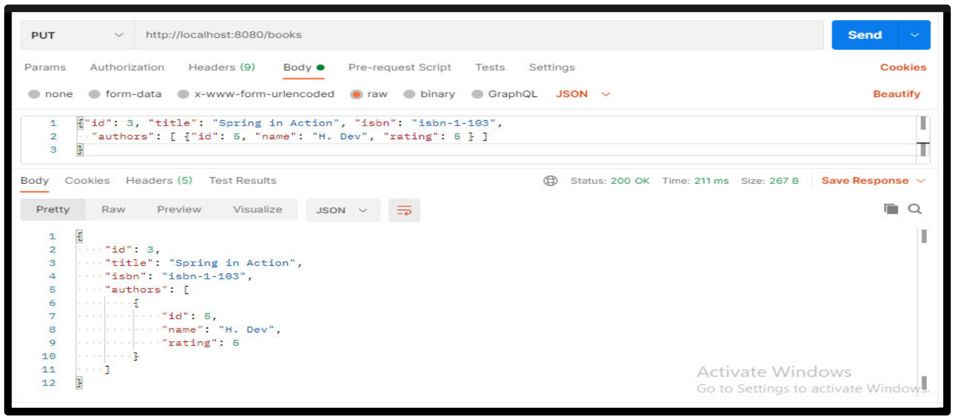
## 11.6 UPDATE RECORDS USING SPRING BOOT AND HIBERNATE

To update the record/records, CrudRepository provides save(…) and saveAll(…) methods. If the entity has an identifier, JPA will perform update operation instead of save operation. Add the following into the BookController.java in order to provide the **update** feature to the rest application.

**BookController.java**

```
@RestController
@RequestMapping("/books")
publicclassBookController
{

        @Autowired
        privateBookRepositorybookRepository;

        @PutMapping
        public Book updateBook(@Valid@RequestBody Book b)
        {
                returnbookRepository.save(b);
        }

}
```

To test the update feature, use the postman and use the PUT method with a valid Book record into the request body. Following request, update the authors of the given book with id 3 (as shown in Figure 11.11).


PUT http://localhost:8080/books

```
{"id": 3, "title": "Spring in Action", "isbn": "isbn-1-103",
  "authors": [ {"id": 5, "name": "H. Dev", "rating": 5 } ]
}
```

Status: 200 OK Time: 211 ms Size: 267 B

```
{
    "id": 3,
    "title": "Spring in Action",
    "isbn": "isbn-1-103",
    "authors": [
        {
            "id": 5,
            "name": "H. Dev",
            "rating": 5
        }
    ]
}
```

Figure 11.11: Update Record Using SpringBoot and Hibernate

## 11.7 DELETE RECORDS USING SPRING BOOT AND HIBERNATE

To delete a record or all records, CrudRepository provides deleteById(…), delete(…) and deleteAll(…) methods. Add the following code into the BookController.java in order to provide the **delete** feature into the rest application.

```
@RestController
@RequestMapping("/books")
publicclassBookController
{

        @Autowired
        privateBookRepositorybookRepository;

        @DeleteMapping("/{bookId}")
        @ResponseStatus(value = HttpStatus.OK)
        publicvoiddeleteBook(@PathVariable(value="bookId", required=true) Long bookId)
        {
                bookRepository.deleteById(bookId);
        }
}
```

**BookController.java**

The delete operation does not return any body response. It just returns the response status as 200 OK on successful deletion. Use the postman to issue the **DELETE** request, as shown into the screenshot (Figure 11.12). Thefollowing is the delete request to delete the book with id as 2. The response status is **200 OK,** and the response body is empty as expected.

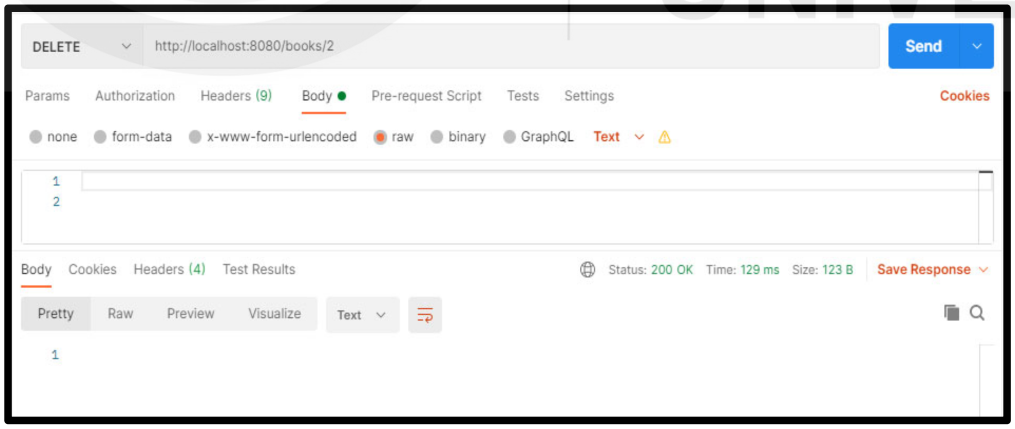


**Figure 11.12: Delete Record Using SpringBoot and Hibernate**

## ✒Check Your Progress 3

1) Explain the annotation **@OneToMany** and its attributes :

```
@OneToMany(cascade=CascadeType.ALL, fetch = FetchType.LAZY)
@JoinColumn(name="EMPLOYEE_ID")
private Set<AccountEntity> accounts;
```

-----------------------------------------------------------------------------------------------

-----------------------------------------------------------------------------------------------

-----------------------------------------------------------------------------------------------

-----------------------------------------------------------------------------------------------

2) How does Spring Data framework derive queries from method names?

-----------------------------------------------------------------------------------------------

-----------------------------------------------------------------------------------------------

-----------------------------------------------------------------------------------------------

-----------------------------------------------------------------------------------------------

3) Explain the following User repository definition.

```
@Repository
public interface UserRepository extends Repository<User, Long> {
List<User>findByEmailAddressAndLastname(String emailAddress, String lastname);
}
```

-----------------------------------------------------------------------------------------------

-----------------------------------------------------------------------------------------------

-----------------------------------------------------------------------------------------------

-----------------------------------------------------------------------------------------------

4) Explain the following repository definition.

```
@Repository
publicinterfaceBookRepositoryextendsCrudRepository<Book, Long>
{
        public List<Book>findByTitleContaining(String pattern);     // 1
        public List<Book>findByTitleContaininsIgnoreCase(String pattern); //2
        public List<Book>findByTitleLike(String pattern);            // 3
                @Query("Select b from Book b where b.title like %:pattern%")
        public List<Book>findByTitle(String pattern);                // 4

}
```

## 11.8 SUMMARY

This unit has explained Spring Data Repository with a complete example using other hibernate association concepts. For better understanding, start coding by using the examples explained in this unit. The important points are as following:

- The goal of Spring Data repository abstraction is to significantly reduce the amount of boilerplate code required to implement data access layers for various persistence stores.
- CrudRepository, PagingAndSortingRepository and JpaRepositoryhave been explained with all available methods.
- Association mappings are one of the key features of JPA and Hibernate. That establishes the relationship between two database tables as attributes in your model and allows you to navigate the association easily in your model and JPQL or criteria queries
- JPA has considered the related entity loading overhead problems ahead and made One-to-Many relationships load lazily by default.
- The fetch type for Many-to-One relationship is eager by default.
- Cascading is about JPA actions involving one entity propagating to other entities via an association. JPA provides javax.persistence.CascadeType enumerated types that define the cascade operations. JPA provides the cascade types as ALL, PERSIST, MERGE, REFRESH, REMOVE, DETACH.
- Query methods using Containing, Contains, and IsContaining generate the like query. For example, query method findByTitleContaining(String pattern) generates the sql query as SELECT * FROM books WHERE title LIKE '%<pattern>%'
- Query parameters as Named Parameters (:name) and Ordinal Parameters (?index)

## 11.9 SOLUTIONS/ ANSWER TO CHECK YOUR PROGRESS

### Check Your Progress 1

1) Java Persistence API (JPA) is a specification that defines APIs for object-relational mappings and management of persistent objects. JPA is a set of interfaces that can be used to implement the persistence layer. JPA itself doesn't provide any implementation classes.
   Hibernate is an implementation of JPA.
   Spring Data JPA is one of the many sub-projects of Spring Data that focuses on simplification of the data access for relational data stores. Spring Data JPA is not a JPA provider; instead it wraps the JPA provider and adds its own

features like a no-code implementation of the repository pattern. Spring Data JPA uses Hibernate as the default JPA provider.

2) There are varieties of data stores such as relational databases, NoSQL databases like MongoDB, Casandraetc, Big Data solutions such as Hadoop etc. Different storage technologies have different configurations, different methods and different API's to fetch the data.
   ***Spring Data*** aims at providing a homogeneous and consistent Spring-based programming model to fetch data along with keeping the special traits of the underlying data stores. Spring Data provides Abstractions (interfaces) that can be used irrespective of the underlying data source.
   **public interface EmployeeRepository extends CrudRepository<Employee, Long> {}**
   No implementation of EmployeeRepository is required in order to perform insert, update, delete and retrieval of EMPLOYEE entities from the database.

3) CrudRepository provides all the typical Save, Update, Delete operations for an entity while JpaRepository extends CrudRepository as well as **Repository**, **QueryByExampleExecutor** and **PagingAndSortingRepository** interfaces.
   So, if only CRUD operations are required to perform, select CRUDRepository. Else, if one or more features out of Pagination, Batch processing and flush operations are required, choose JPARepository.

## Check Your Progress 2

1) One-to-Many and Many-to-One relationships are the opposite sides of the same coin and closely related. **One-to-Many** mapping is described as one row in a table  mapped to multiple rows in another table.
   For the illustration of the One-to-Many relationship, the **Teacher** and their **Courses** will be taken. A teacher can give multiple courses, but a course is given by only one teacher is an example of one-to-many relationship. While another perspective, many courses are given by a teacher is an example of the many-to-one relationship.

```
@Entity
publicclass Teacher
{

            @Id
            @GeneratedValue(strategy = GenerationType.IDENTITY)
            private Long id;

            private String name;

            @OneToMany(cascade = CascadeType.ALL)
   @JoinColumn(name = "teacher_id", referencedColumnName = "id")
            private List<Course>courses;

}


@Entity
publicclass Course
{
            @Id
            @GeneratedValue(strategy = GenerationType.IDENTITY)
            privateintid;
            private String title;

}
```

2) The difference between lazy and eager loading is described in the table.

| S.No. | Key | Lazy | Eager |
|---|---|---|---|
| 1 | Fetching strategy | In Lazy loading, associated data loads only when we explicitly call getter or size method. | In Eager loading, data loading happens at the time of their parent is fetched |
| 2 | Default Strategy in ORM Layers | ManyToMany and OneToMany associations used a lazy loading strategy by default. | ManyToOne and OneToOne associations used a lazy loading strategy by default. |
| 3 | Loading Configuration | It can be enabled by using the annotation parameter :<br><br>fetch = FetchType.LAZY | It can be enabled by using the annotation parameter :<br><br>fetch = FetchType.EAGER |
| 4 | Performance | Initial load time much smaller than Eager loading | Loading too much unnecessary data might impact performance |

Performance improvement can be explained with an example. Consider the teacher and course as a one-to-many relationship. The **Course** is a heavy object i.e. having too many attributes. For some business logic, all **Teacher** objects are loaded from the database. Business logic does not need to retrieve or use courses in it, but **Course** objects are still being loaded alongside the **Teacher** objects.
Such loading will degrade the performance of applications. JPA has considered all such problems ahead and made **One-to-Many relationships load *lazily* by default**. Lazy loading means relationships will be loaded when it is actually needed.

3) One of the main aspects of JPA is that it helps to propagate Parent to Child relationships. This behavior is possible through CascadeTypes. The CascadeTypes supported by JPA are:

| Cascade Type | Description |
|---|---|
| ALL | CascadeType.ALL propagates **all operations,** including hibernate specific from a parent to a child entity. |
| PERSIST | CascadeType.PERSIST propagates the **save()** or **persist()** operation to the related entities. |
| MERGE | CascadeType.MERGE propagates only the **merge()** operation to the related entities. |
| REFRESH | CascadeType.REFRESH propagates only the **refresh()** operation to the related entities. |
| REMOVE | CascadeType.REMOVE removes all related entities associations when the owning entity is deleted. |
| DETACH | CascadeType.DETACH detaches all related entities if the owning entity is detached. |

Cascading is useful only for Parent - Child associations where the parent entity state transitions cascade to the child entity. The cascade configuration option accepts an array of CascadeTypes. The below example shows how to add the refresh and merge operations in the cascade operations for a One-to-Many relationship:

```
@OneToMany(cascade={CascadeType.REFRESH, CascadeType.MERGE},
fetch = FetchType.LAZY)
@JoinColumn(name="EMPLOYEE_ID")
private Set<AccountEntity> accounts;
```

## Check Your Progress 3

1) **One-to-Many** mapping is described as one row in a table is mapped to multiple rows in another table and Hibernate provides annotation @OnetoMany. The attribute "cascade=CascadeType.ALL" means that any change that happens to Employee Entity must cascade to all associated entities (Accounts Entity in this case) also. This means that if you save an employee into the database, then all associated accounts will also be saved into the database. If an Employee is deleted, then all accounts associated with that Employee will also be deleted. However, if we want to cascade only the save operation but not the delete operation, we need to clearly specify it using below code.

   ```
   @OneToMany(cascade=CascadeType.PERSIST, fetch = FetchType.LAZY)
   @JoinColumn(name="EMPLOYEE_ID")
   private Set<AccountEntity> accounts;
   ```

   Now only when the save() or persist() methods are called using an employee instance, the accounts will be persisted. If any other method is called on session, its effect will not affect/cascade to the accounts entity.
2) Spring Data framework has an in-built query builder mechanism that derives queries based on the method name. Method names are defined as 'find...By..', 'count...By...' etc. 'By' acts as the delimiter to identify the start of the criteria. 'And' and 'Or' are used to define multiple criteria. 'OrderBy' is used to identify the order by criteria. Few examples of query methods are as follows:
   - **findByFirstname(String firstName):** It derives the query to get the list based on the first name.
   - **findByFirstnameAndLastname(String firstName, String lastName):** It derives the query to get the list based on first name and last name
   - **findByFirstnameAndLastnameOrderByFirstnameAcs(String firstName, string last):** it derives the query to get the list based on the first name and sorted by the first name in ascending order.

3) UserRepository extends the Repository marker interface. Marker interface does not contain any method. Thus, UserRepository provides a single API to find the list of users based on the email address and last name. The query method:
   **List<User>findByEmailAddressAndLastname(String emailAddress, String lastname)** translates into the following query:
   **select u from User u where u.emailAddress = ?1 and u.lastname = ?2.**
4) BookRepository extends the CrudRepository interface, which provides the methods to read all records, specific record by id etc. All methods generate the same queries, but there are some differences.
   1. Query methods using Containing generates the following sql query.
      **SELECT * FROM books WHERE title LIKE '%<pattern>%'**

2. The ***findByTitleContainsIgnoreCase(String pattern)*** is similar to ***findByTitleContaining(String pattern)*** but with case insensitive.
3. The **findByTitleLike(String pattern)** generates **SELECT * FROM books WHERE title LIKE '<pattern>'**. JPA Like keyword behaves differently. It can be noticed in the generated query that it does not include wild char %. Thus, while calling this method, argument should have wild char included.
4. Sometimes we need to create queries that are too complicated for Query Methods or would result in absurdly long method names. In those cases, we can use the @Query annotation to query our database. The method findByTitile with @Query annotation is similar to the first method.

## 11.10 REFERENCES/FURTHER READING

- Craig Walls, "Spring Boot in action" Manning Publications, 2016. (https://doc.lagout.org/programmation/Spring%20Boot%20in%20Action.pdf)
- Christian Bauer, Gavin King, and Gary Gregory, "Java Persistence with Hibernate",Manning Publications, 2015.
- Ethan Marcotte, "Responsive Web Design", Jeffrey Zeldman Publication, 2011(http://nadin.miem.edu.ru/images_2015/responsive-web-design-2nd-edition.pdf)
- Tomcy John, "Hands-On Spring Security 5 for Reactive Applications",Packt Publishing,2018
- https://spring.io/projects/spring-data
- https://www.baeldung.com/the-persistence-layer-with-spring-data-jpa
- https://github.com/spring-projects/spring-data-examples
- https://dzone.com/articles/magic-of-spring-data
- https://www.journaldev.com/17034/spring-data-jpa
- https://docs.jboss.org/hibernate/orm/3.3/reference/en/html/associations.html
- https://www.baeldung.com/spring-data-rest-relationships