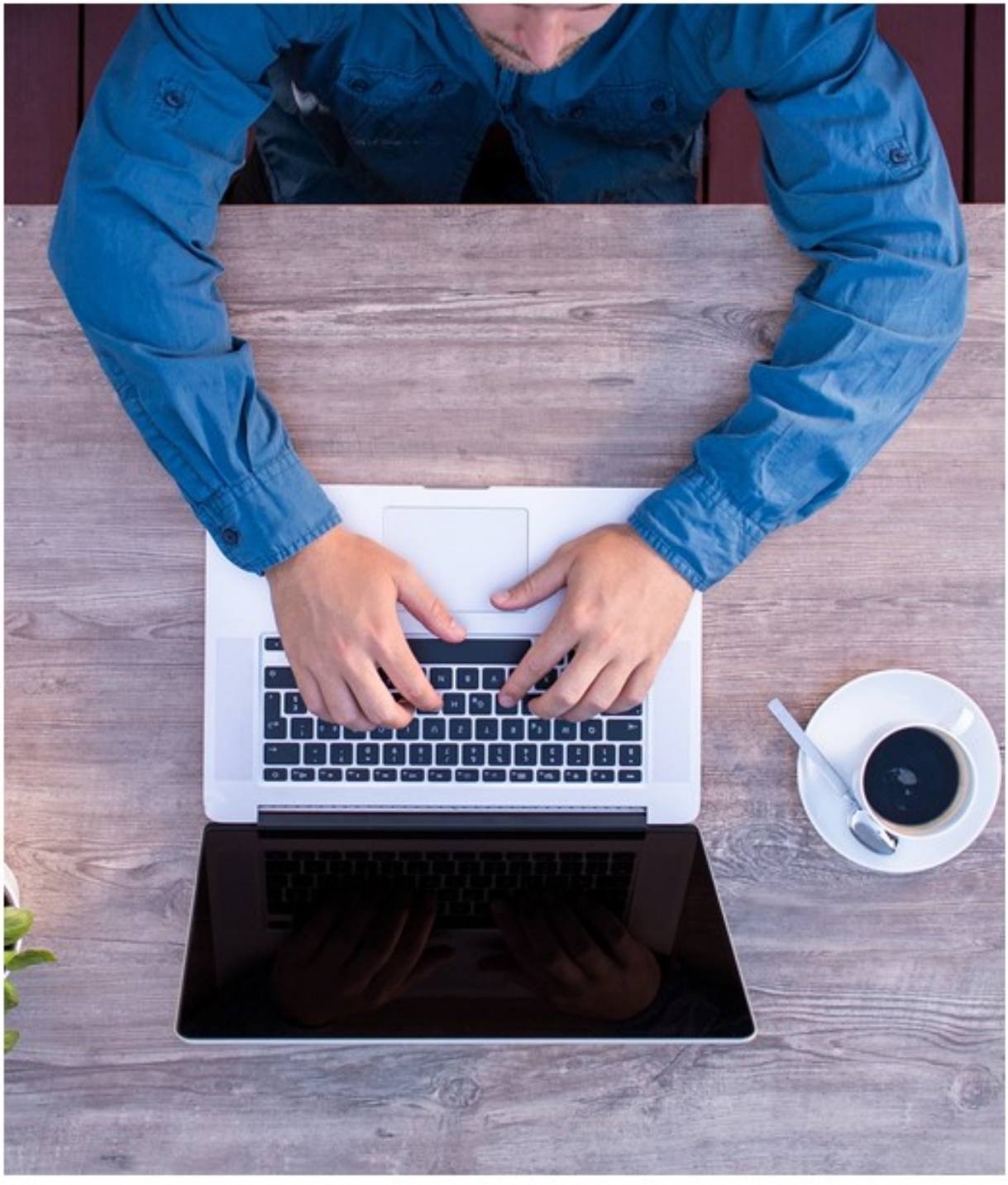
Integrating Java Applications with Spring Framework



Integrating Java Applications with Spring Framework

Learner's Guide

APTECH LIMITED
Contact E-mail: ov-support@onlinevarsity.com

Edition 1 - 2018

Preface

The book, **Integrating Java Applications with Spring Framework**, covers the features of the Spring Framework for Java. The Spring Framework has grown over years from just being an Inversion of Control container and currently includes several modules that provide a range of services such as Aspect-oriented programming, data access, transaction management, Model-View-Controller features, and more. This book describes the core components of Spring Framework and explains how Spring can be used for development of enterprise grade applications.

This book is the result of a concentrated effort of the Design Team, which is continuously striving to bring you the best and the latest in Information Technology. The process of design has been a part of the ISO 9001 certification for Aptech-IT Division, Education Support Services. As part of Aptech's quality drive, this team does intensive research and curriculum enrichment to keep it in line with industry trends.

We will be glad to receive your suggestions.

Design Team

Onlinevarsity

GROUPS

COLLABORATIVE LEARNING

Onlinevarsity

Onlinevarsity App

for Android devices

Download from Google Play Store

Table of Contents

MANY COURSES ONE PLATFORM

Session 1

Introduction to Spring Framework

Objectives

Welcome to the session, Introduction to Spring Framework.

This session describes the Spring Framework and its features and architecture. In addition, this session discusses the advantages of the framework and describes how to create an application using Spring Framework.

In this Session, you will learn to:

- Describe Spring Framework
- Identify the features and advantages of Spring Framework
- Describe the architecture of Spring Framework

1.1 What is Spring Framework?

Spring Framework is an open-source application development framework that is used to develop robust enterprise-level Java applications. Spring Framework provides infrastructural support to Java developers, allowing them to concentrate on the core program logic.

Spring Framework

- Uses a simple approach to develop enterprise-level Java applications and is sometimes referred to as framework of frameworks. For example, a Java code without the Spring Framework support, will include code that is redundant or written multiple times to perform a simple task. This redundant code is called boilerplate code. However, if a Java code is developed using the Spring Framework, the JdbcTemplate class eliminates the requirements of boilerplate code. This enables developers to concentrate on developing actual modules of the Java application.

- Is also defined as a lightweight Inversion-of-Control (IoC) and aspect-oriented container framework, that is used to build enterprise-level Java applications. Inversion-of-Control is a programming design principle in which the user-defined code modules of a program receive the control from a framework. This design principle is termed as Inversion-of-Control because it inverts the control flow of the commonly used procedural programming design principle. In procedural programming, custom or user-defined code calls reusable libraries to execute generic tasks, such as printing results. However, in Inversion-of-Control design principle, the framework calls the custom or task-specific code modules.

1.2 Features of Spring Framework

Spring Framework offers the following features:

➢ **Non-intrusive**: The Spring Framework is developed to be non-intrusive, which implies that the domain's logic code is not dependent on the framework. Therefore, an object of an application developed using the Spring Framework is not dependent on any class or interface defined in the Spring API.

➢ **Lightweight**: The Spring Framework is considered as a lightweight framework as it reduces complexities in the application code. A framework is considered as a

lightweight framework when it has low start-up time and runs in any environment. In addition, Spring Framework's own functioning is transparent and avoids unnecessary complexities.

- **Inversion-of-Control (IoC)**: As already mentioned earlier, Spring Framework uses the IoC framework that improves modularity of the code and makes the code loosely-coupled. IoC implementation identifies and describes the dependency injection needs of the objects. Dependency Injection is the process of instantiating an object externally.

- **Aspect-Oriented Programming**: The Spring Framework uses the AOP methodology to isolate supporting functions from the main program's business logic. This methodology helps to apply middleware services, such as security and transaction management to a Spring-enabled Java application.

- **JDBC abstraction layer**: This layer is used in the Spring Framework to reduce the amount of boilerplate code written for error and exception handling.

- **MVC Framework**: The Spring Framework uses the MVC framework to develop robust Web applications. The MVC framework provides utility classes to handle some of the common tasks in Web application development.

1.3 Advantages of Spring Framework

Spring Framework offers the following advantages:

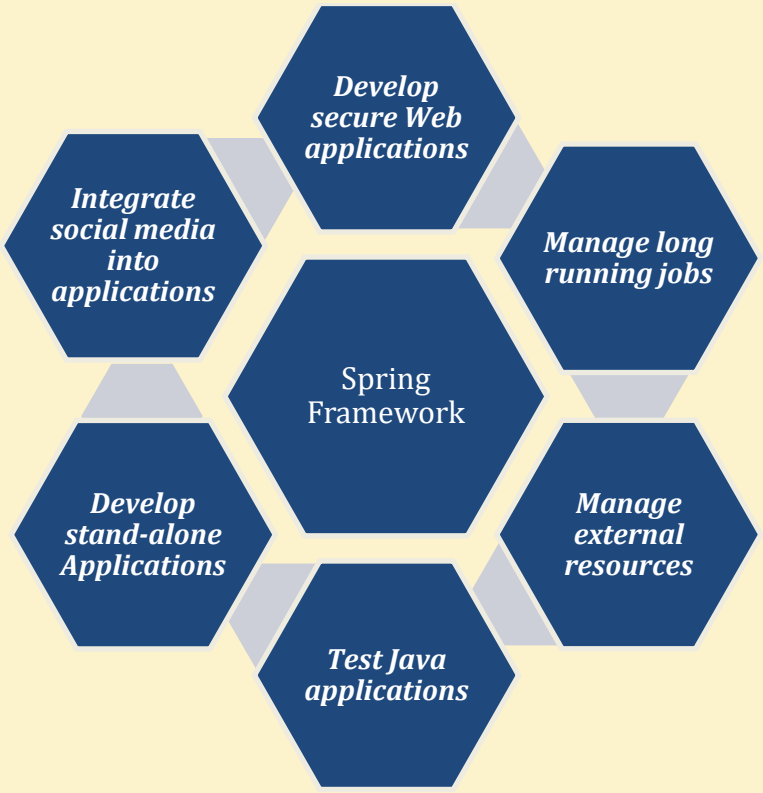
- Spring Framework promotes good programming practices, such as using interfaces instead of classes.
- Spring Framework follows a modular programming approach, in which developers use only those components which they require.
- Spring Framework supports both XML-based and annotation-based configuration methodologies.
- Spring Framework is lightweight and does not require a Web server or an application server software for activation.
- Spring Framework uses IoC and Dependency Injection to help to develop loosely-coupled applications.
- Spring Framework supports JDBC framework that improves error and exception handling in application development.
- Sprint Framework supports MVC framework that facilitates development of flexible and robust Web applications.

1.4 Inversion-of-Control and Spring Framework

IoC principle is aimed at improving the modularity of the entire application code and making the code extensible. A program code is considered extensible if it allows addition of new elements and features in its existing framework to extend its capabilities. IoC

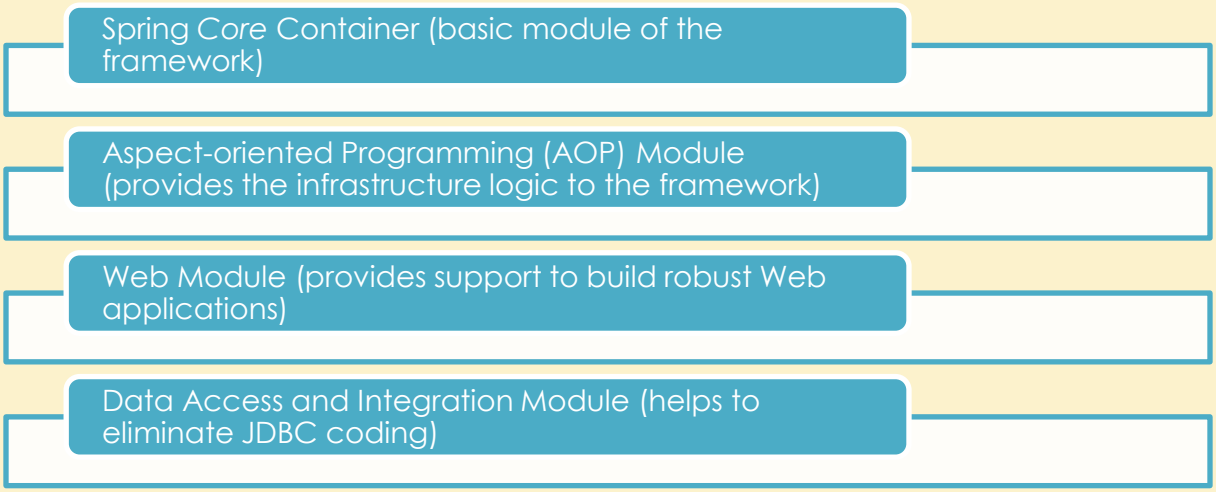
principle also aims to make the program code independent of any environment so that it can be used in different programming environments.
IoC is implemented through Dependency Injection (DI). DI is a pattern in which one object provides the dependencies of another object. A dependency is an object or a service that can be used by another object and injection refers to the process of passing the dependency to the object that needs it. Here, the object that uses the dependency is the dependent object. For example, Dependency Injection is a pattern in which service required by a client is provided from outside instead the client finding or building its service on its own. Dependency Injection pattern enables loosely-coupled programming modules, in which no module code should be updated because the object it depends on, is changed.

With the concept of IoC, DI, and more, Spring Framework is one of the most commonly used application development framework because it enables you to do the following:



1.5 Spring Framework Architecture

The Spring Framework consists of following modules:



Spring Core Container

The Spring Core container is the basic module of the Spring Framework and comprises following sub-modules:

- **Core module** is the base module of the Spring Framework, which includes IoC and Dependency Injection features. Dependency Injection is a design principle that reduces or removes dependency on container APIs. Dependency Injection design principle is used to make applications independent of its objects or to make classes independent of how its objects are created.
- **Beans module** in the Spring Framework Core container provides the BeanFactory factory pattern. This pattern separates the dependencies, such as creation of and access to objects from the program logic.
- **Context module** provides the ApplicationContext container that loads the bean definitions and joins them together. This container is the central point of the Context module. The Context module is the medium to access the objects defined and configured in the Framework.
- **Expression language** or the Spring Expression Language (SpEL) is a powerful expression language that supports various features, such as injecting bean or bean property into another bean or retrieving and invocating objects from IoC by name.

Aspect-oriented Programming (AOP) Module

AOP framework is a modular programming methodology that uses aspect as a module unit. This is complementary to Object-oriented Programming framework, which uses class as a module unit.

In simple words, AOP divides the program logic into units called **concerns**, isolates the concerns of your application, reduces code clutter, and improves maintainability and readability of your code.

The AOP programming methodology includes modularity by using cross-cutting concerns. Cross-cutting concerns are concerns or functionalities that impact multiple objects and types.

Let us understand the concept of AOP with an example.

All applications generate run-time logs, which contain performance information of all application functionalities. To collect this log information, relevant functionality has to be inserted in all modules as well as the main application. Such functionality is spread across all modules in an application and is not central to the application's business logic. In the AOP world, it is called as cross-cutting concern. AOP methodology recommends that such concerns are encapsulated and abstracted into modules, called aspects.

In the Spring Framework, AOP provides the infrastructure logic to it. Using the AOP methodology, developers can introduce new functionalities and capabilities into the existing application code without modifying the design. In addition, developers can use AOP to include cross-cutting aspects in the application code. The Spring AOP framework is configured at runtime and is also used to provide enterprise-level services, such as application security.

To summarize, benefits of AOP include:

- Lesser code clutter
- Lesser code redundancy
- Improved code readability
- Easier maintenance of source code
- Quicker development

Data Access and Integration Module

The Data Access module includes the following sub-modules:

- Spring JDBC is the JDBC abstraction framework that helps to eliminate JDBC related coding. In addition, the JDBC abstraction framework provides the JdbcTemplate class, which is the central class of this framework. The JdbcTemplate class provides the logic to perform data access functions, such as handling connection creation, creating and executing a statement, and releasing a resource.
- Spring Object-Relational Mapping (ORM) module helps to avoid the use of boilerplate code in the application and reduces complexity. The Spring ORM module also provides an abstraction layer for ORM APIs.
- Spring Object XML Mapper (OXM) sub-module that provides support to the Object/XML mapping. This sub-module facilitates data transfer between applications by transforming a JavaBean into an XML format and vice-versa. This sub-module supports integration with Castor, XmlBeans, XStream framework, and JAXB.

- Java Messaging Service sub-module is used to produce and consume messages. This sub-module uses the Java Message Oriented Middleware (MOM) API to send messages between two or multiple clients.

Web Module

The Web module consists of the following sub-modules:

- Web sub-module provides support to build robust Web applications that can be easily maintained. This sub-module also supports the multipart file-upload functionality.
- Servlet sub-module includes the Model-View-Controller (MVC) implementation that is used to develop enterprise-level Web applications.
- Struts sub-module supports the integration of the Struts Web tier with the Java application developed using the Spring Framework.
- Portlet sub-module facilitates easy and quick development of Web applications. Portlet is used to display the contents from the data source and is used in the UI layer.

1.6 Evolution of Spring Framework

Latest version is Spring Framework 5.0. However, the current commonly used version is Spring Framework 4.3.x.

Various versions of Spring Framework are as follows:

Let us discuss the features of Spring versions in detail.

- **Spring Framework 1.0**: This was a comprehensive Java/J2EE application framework. Some of its components are as follows:
  - Spring Context is the medium to access objects that are defined and configured in the Framework.
  - Spring AOP allows Java developers to implement custom-defined aspects.
  - Spring Data Access Object (DAO) enables easy access to database technologies, such as Hibernate, JDBC, JPA, or JDO.
  - Spring JDBC, the JDBC abstraction framework, provides classes to interface with the database and helps to eliminate JDBC related coding.
  - Spring ORM helps to avoid use of boilerplate code from the application and reduces complexity. The Spring ORM module also provides an abstraction layer for ORM APIs.
  - Spring Web is the Spring Model-View-Controller framework that allows the development of loosely-coupled and flexible Web applications.

  The Spring Framework 1.0 was released in March 2004.

- **Spring Framework 2.x**: This version was developed with two main themes, simplicity and power. The Spring Framework 2.0 was released in October of 2006 and version 2.5 was released in November of 2007. This version included the following features:
    - @AspectJ annotation support AOP development
    - Bean configuration dialects
    - XML schema support and custom namespace
    - Annotations, such as @RequestMapping, @requestParam, and @ModelAttributes for MVC controllers

  This version also included improvements to the IoC container and introduced the annotation-driven configuration to auto-detect annotated components using component scanning.

- **Spring Framework 3.0**: This version was developed to enable the entire Spring base code to take advantage of Java 5.0 technology and was released in December of 2009. This version included the following features:
    - REST in Spring MVC
    - New XML namespace that enables easier configuration of Spring MVC
    - New annotations, @CookieValue and @RequestHeader to pull values from cookies and request headers
    - Task scheduling
    - Asynchronous method execution with annotation support

  This version supported the Hibernate version 3.6 final.

- **Spring Framework 3.1**: This version was introduced with a host of new features, such as:
    - Cache abstraction to add caching concept using the @Cacheable annotation to an existing application
    - c:namespace to support constructor injection
    - PropertySource abstraction object
    - @RequestPart annotation that enables access to multipart form-data content on the controller method arguments

  This Spring Framework version was released in December 2011 and supported the Hibernate 4.x version though the Java Persistence API (JPA).

- **Spring Framework 3.2.x**: This version introduced the following features and enhancements:
    - Servlet-3 based asynchronous request processing
    - Spring MVC application testing done without a Servlet container; using DispatcherServlet for server-side REST tests and RestTemplate for client-side REST tests
    - ContentNegotiationStrategy that resolved media types from an incoming request
    - @MatrixVariable annotation to extract matrix variables

- @DateTime Format annotation to remove dependency on Joda-Time library

This Spring Framework version was introduced in November of 2013 and provided support for Java 7 features.

➢ **Spring Framework 4.x**: This version introduced the following features:
- @Description annotation
- @Conditional to filter the beans based on the specified condition
- External bean configuration using Groovy DSL
- Testing features, such as SQL Script execution and bootstrap strategy
- WebSocket-style architectures
- Lightweight messaging

The Spring Framework 4.x version supports Java SE 8 platform and Groovy 2 programming language.

1.7 Creating an Application in Spring

You need to perform the following steps before creating an application in Spring Framework:

1. **Get Spring JAR Files**: The first step is to download the Spring Framework library from the following Web location:
   http://repo.spring.io/release/org/springframework/spring/

   Navigate to the appropriate release version folder to download the required Spring Framework release version. The Spring Framework files for distribution follow the naming convention as `spring-framework-x.x.x.Release-dist.zip`.

   Extract all the files from the zip file and navigate to the **libs** folder. The **libs** folder contains all the JAR files for each Spring Framework module. The Spring Framework distribution zip file contains the following sub-folders:
   - Docs
   - Libs
   - Schema

2. **Download and install SpringSource Tool Suite (STS)**: This is an Eclipse-based development environment to develop Spring applications. The latest version of STS can be downloaded from the following Web page:

   http://spring.io/tools/sts

   Now you are ready to create a Spring application. The steps to create a Spring application are as follows:
   i. Launch STS.
   ii. Select the **File → New → Spring Project** menu option and specify the name as FirstSpringProject.

iii. Select the **Simple Java** template from the **Templates:** section on the **New Spring Project → Spring Project** dialog box.

iv. Click **Finish** to create the Spring project, titled as FirstSpringProject as shown in Figure 1.1.

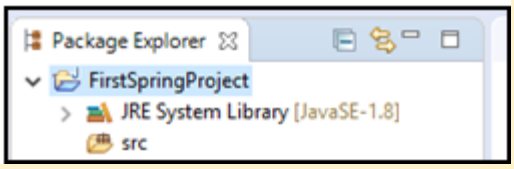


Figure 1.1: Create a Spring Project

v. Now you will add the required libraries, which include the Spring Framework libraries and the common logging API libraries. First, download the common logging library from the following Web page:
http://commons.apache.org/proper/commons-logging/download_logging.cgi

vi. Then, to add the required Spring Framework libraries, right-click the FirstSpringProject and select the **Build Path → Configure Build Path** pop-up menu option.

vii. Click **Add External JARs** from the **Libraries** tab to add the core JAR files from the Spring Framework and common logging installation directories. These files can be:

- spring-aop-4.3.9.RELEASE
- spring-aspects-4.3.9.RELEASE
- spring-beans-4.3.9.RELEASE
- spring-context-4.3.9.RELEASE
- spring-context-support-4.3.9.RELEASE
- spring-core-4.3.9.RELEASE
- spring-expression-4.3.9.RELEASE
- commons-logging-1.2

viii. Create the source files for the FirstSpringProject. To do this, perform the following steps:

- Create two packages titled as **org.springframework.module1.service** and **org.springframework.module1.main**. Let's refer to these packages as **main** and **service** respectively.

- Create a class called **ModuleClass.java** inside the **main** package.

- Create an interface named **HelloMessage.java** and its implementation class titled, **HelloMessageImpl.java** inside the **service** package.

Figure 1.2 displays the source files for FirstSpringProject.

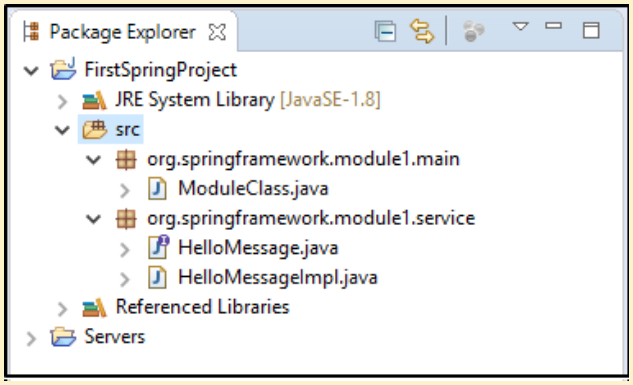


Figure 1.2: Spring Project Source Files

Figure 1.3 displays the various libraries referenced by the project that were added through step vii.

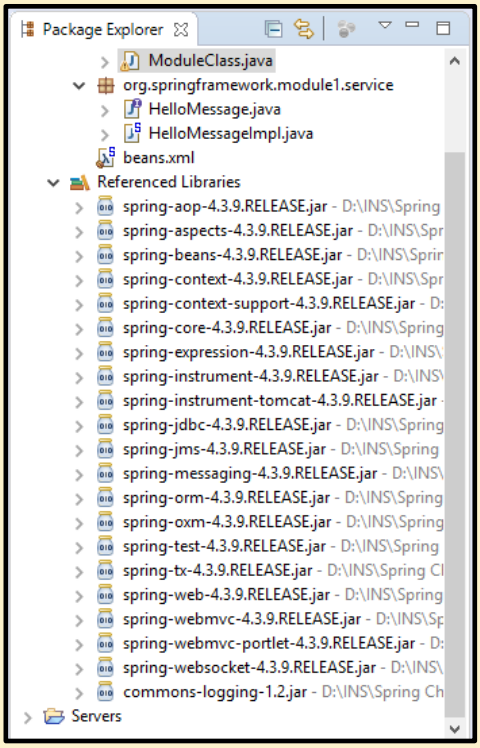


Figure 1.3: Libraries Referenced by FirstSpringProject

- Add the following code to the interface named **HelloMessage.java**:

```
package org.springframework.module1.service;

public interface HelloMessage
{
	public String greetUser();
}
```

- Add the following code to implementation class titled **HelloMessageImpl.java**:

```
package org.springframework.module1.service;

import org.springframework.stereotype.Service;
@Service
public class HelloMessageImpl implements HelloMessage
{
    public String greetUser()
    {
            return "Welcome to the Class on Java Spring Framework";
    }
}
```

In the ModuleClass.java class, the ApplicationContext is created using framework API, as illustrated by the following code:

```
ApplicationContext context = new ClassPathXmlApplicationContext(
"beans.xml");
```

This API loads the beans.xml, which is the Spring beans configuration file. This Spring beans configuration file takes care of creating and initializing all the bean objects. The getBean() method of ApplicationContext, is used to retrieve the required Spring bean from the application context, as shown in the following code:

```
HelloMessage hello = context.getBean(
"HelloMessageImpl", HelloMessage.class);
```

The getBean() method uses bean ID and bean class to return a bean object.

The complete code for ModuleClass.java is as follows:

```
package org.springframework.module1.main;
import org.springframework.context.ApplicationContext;
import
org.springframework.context.support.ClassPathXmlApplicationContext;
import org.springframework.module1.service.HelloMessage;
public class ModuleClass
{
        public static void main(String[] args)
        {
                ApplicationContext context = new
                    ClassPathXmlApplicationContext("beans.xml");
               HelloMessage hello = context.getBean("HelloMessageImpl",
                   HelloMessage.class);
                System.out.println(hello.greetUser());
        }
}
```

ix. Create a Spring bean configuration file in the **src** directory by performing the following steps:
- Right-click the **src** option and select the **New → Spring Bean Configuration File** menu option.
- Enter the name as **beans** in the **File name:** text box on the **New Spring Bean Definition File** dialog box.
- Select the **context** option from the **Select desired XSD namespace declarations:** section on the **New Spring Bean Definition File** dialog box.
- Click **Finish** to close the **New Spring Bean Definition File** dialog box and create the Spring Bean configuration file.

x. Add the following code in the beans.xml file:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:context="http://www.springframework.org/schema/context"
        xsi:schemaLocation="http://www.springframework.org/schema/be
ans http://www.springframework.org/schema/beans/spring-beans.xsd
                http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-
4.3.xsd">
<context:component-scan base-
package="org.springframework.module1.service"/>
 <bean id="HelloMessageImpl"
class="org.springframework.module1.service.HelloMessageImpl" />
</beans>
```

The **beans.xml** file is used by the Spring Framework to create the beans required by the Spring application.

xi. Execute the application by performing the following steps:

- Right-click the **ModuleClass.java** file and select the **Run as → Java Application** menu option.
- After successful execution of the program, the following message is displayed on the STS IDE's console:

  'Welcome to the class on Java Spring Framework'

  Figure 1.4 displays the output of the Spring project.

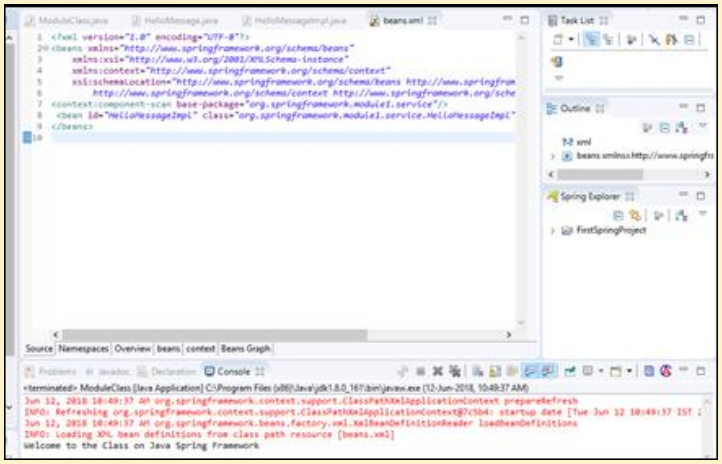


Figure 1.4: Spring Project Output

Summary

- Spring Framework is an open-source modular application development framework that is used to develop robust enterprise-level Java applications.
- Spring Framework is defined as a lightweight Inversion-of-Control (IoC) and aspect-oriented container framework.
- The Spring Core container is the basic module of the Spring Framework and consists of four sub-modules, core, beans, context, and expression language.
- In the Spring Framework, AOP provides the infrastructure logic to it and is configured at runtime.
- Spring JDBC helps eliminate JDBC related coding by using the JdbcTemplate class, which provides the logic to perform data access functions, such as handling connection creation and so on.
- The Web module consists of the Web, Servlet, and Struts sub-modules, and provides support to build robust Web applications.
- Spring Framework was first developed in 2003, and the latest version is Spring 5, though commonly used current version is 4.3.x.
- The Spring Framework is developed to be non-intrusive, which implies that the domain's logic code is not dependent on the framework.
- The Spring Framework uses the MVC framework, which provides utility classes to handle some of the common tasks in Web application development.

Check Your Progress

1. Which of the following Spring Framework versions was developed to take advantage of Java 5.0?

A.	3.0	**B.**	2.x
C.	2.3	**D.**	All of these

2. In AOP methodology, _____ are functionalities that impact multiple objects and types.

A.	Cross-cutting points	**B.**	Concerns
C.	Cross-cutting concerns	**D.**	Cross-cutting aspects

3. Which of the following annotation is used to extract matrix variables?

A.	@Matrix	**B.**	@MatrixVariable
C.	C:MatrixVariable	**D.**	@Variable

4. _____ contains all the JAR files when extracted from the Spring Framework distribution zip file.

A.	libs	**B.**	libraries
C.	applibs	**D.**	lib

5. The getBean() method uses bean ID and _____ to return a bean object.

A.	Bean interface	**B.**	Bean object
C.	Bean library	**D.**	Bean class

Answers

1.	A
2.	C
3.	B
4.	A
5.	D

Session 2

Spring Core

Objectives

Welcome to the session, Spring Core.

This session describes various concepts involved in the Inversion of Control design principle and basics of aspect-oriented programming. It also describes the new features supported by Spring 5.0.

In this Session, you will learn to:

- Explain how to use Inversion of Control in your programs
- Explain the basics of aspect-oriented programming
- Describe new features of Spring 5.0

2.1 Understanding Inversion of Control

Inversion of Control (IoC) is a conceptual design principle in which the control of objects or custom-written portions of a program is transferred to a framework.

In simplest terms, IoC enables one object to control the behavior of other objects, however, they do not have strong dependencies on each other.

In traditional programming, the custom code that defines the purpose of a program makes calls to reusable libraries to perform generic tasks. In simple words, programs followed a given path through the code. The order of the called functions was decided by the programmer.

On the other hand, IoC enables a framework to take control of the flow of a program to make calls into the custom or task-specific code.

To support this, the frameworks are defined through abstractions with additional built-in behavior. To add the required behavior, extend the classes of the framework or plugin your own classes.

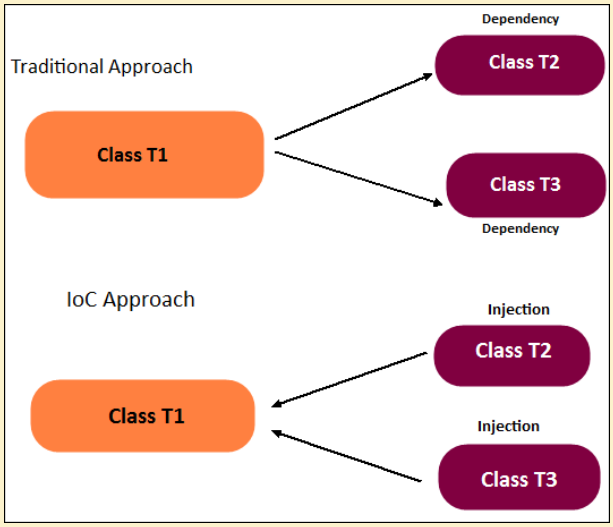


Figure 2.1: Traditional versus IoC Approach

Figure 2.1 illustrates the IoC approach.

IoC provides more freedom, flexibility, and less dependency.

To understand it better, consider that you are working on a desktop computer. In this scenario, you are **controlled** and must sit at a place to continue working on it. If you replace your desktop with a laptop, then you **invert control**. You can easily move around with it. As a result, **you** can control where you are with your device, instead of **the device controlling you**.

Another analogy to understand IoC can be the 'Chain of Command' observed in the military. Upon enlisting in the military, each new member is provided with basic items and instructions required to perform his/her duty based on their rank. They are given commands that they must obey.

Similar logic can be applied to code. Each component is provided with necessary items it needs to operate by the instantiating entity, like the Commanding Officer in our analogy. The instantiating entity then acts on that component as per the instructions or requirements.

IoC is useful when modular systems are being developed because IoC provides greater modularity of a program that makes it easy for to replace components without requiring recompilation.

Additionally, there are following benefits of using IoC:

- The execution of a task is decoupled from its implementation.
- It is easy to switch between different implementations.
- It is easy to test a program by separating a component or removing its dependencies and allowing components to communicate through contracts.

Consider the following example:
There is a **Car** class and a **Vehicle** class object, **vehicle**. The biggest issue with the code is tight coupling between classes. In other words, the Car class depends on the **vehicle** object. So, for any reason, changes in the Vehicle class will lead to the changes in and compilation of the Car class too. So, let's note down the problems with this approach:

- The biggest problem is that the Car class controls the creation of the vehicle object.
- The Vehicle class is directly referenced in the Car class, which leads to tight coupling between the car and vehicle objects.

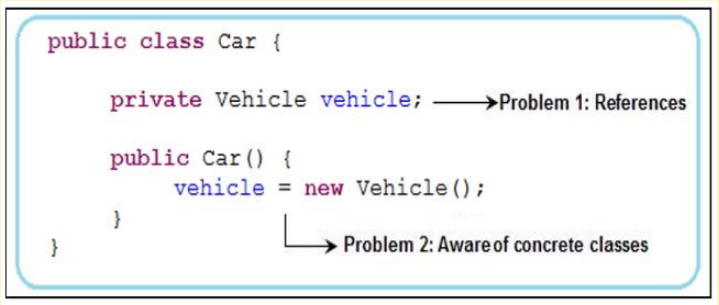


Figure 2.2: Sample Code with Public and Private Class

Figure 2.2 illustrates this concept.

If, for any reason, user has not defined the vehicle object, it will lead to the failure of the Car class in the constructor initialization stage.

The basic principle of IoC is similar to what they say commonly in interviews after they finish your rounds: Don't call us; we'll call you.

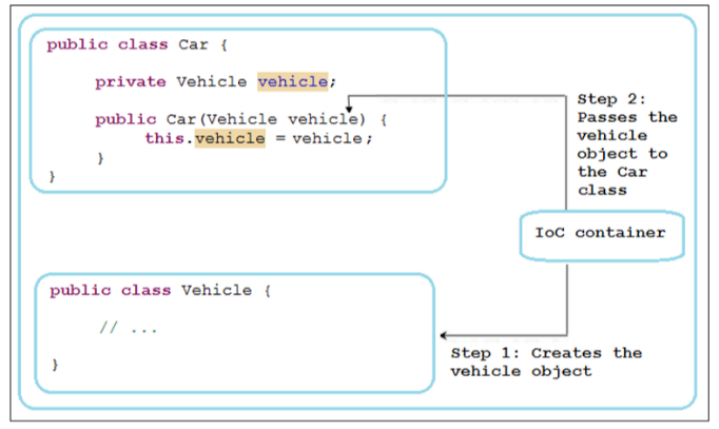


Figure 2.3: Sample Code with IoC Container

In other words, it's as if the Vehicle class saying to the Car class, 'Don't create me, I'll create myself using someone else'.

The IoC framework can be a class, client, or some kind of IoC container. The IoC container creates the vehicle object and passes this reference to the Car class, as shown in Figure 2.3.

Let us now explore some of the IoC terminology.

2.1.1 Containers

Container, in general, is an object that is used to hold or transport some goods. In computer science, a container is an application program or a subsystem in which a component (program building block) runs.

In programming, a container is a class, a data structure, or an Abstract Data Type (ADT), whose instances are collections of other objects. The size of a container is dependent on the number of objects or elements it contains.

Containers can have the following properties:

- **Access**: To access the objects of the container.
- **Storage**: To store the objects of the container.
- **Traversal**: To traverse the objects of the container.

2.1.2 Spring Container

The Spring container is one of the most essential components of Spring Framework that implements IoC; therefore, it is also known as IoC container. The Spring container creates the objects, keeps them connected, configures them, and manages their complete

lifecycle from the time they are created up to the time they are destroyed. The objects that the Spring container contains are called Spring beans.

The Spring container makes use of Dependency Injection (DI) to manage the components of an application.

Input to the Spring container include:

- Plain Old Java Object (POJO) classes that are Spring beans
- Configuration metadata

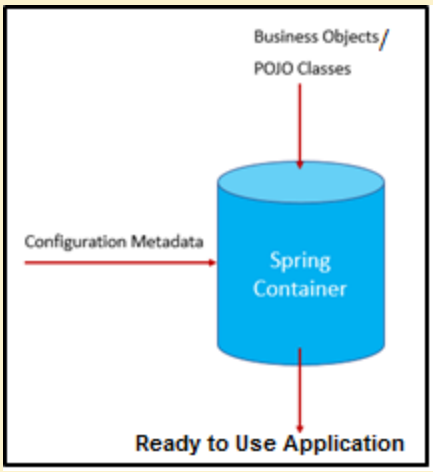


Figure 2.4: Spring Container Framework

Using these inputs, it generates a fully configured and executable system or application. POJO is a normal Java class that is not bound by any special restriction and does not need any class path. The Spring container framework is shown in Figure 2.4.

Spring provides following two types of containers:

BeanFactory

- Provides support for DI and is defined by the org.springframework.beans.factory.BeanFactory interface.
- Is responsible for creating and dispensing beans and managing dependencies between beans.
- Can be used to provide backward compatibility to several third-party frameworks that integrate with Spring.
- The most common implementation of this container is XMLBeanFactory.

ApplicationContext

- Provides enterprise-specific functionality, such as ability to resolve textual messages from a properties file and the ability to publish application events to interested event listeners.
- Is defined by the org.springframework.context.ApplicationContext interface.
- The most common implementation of this container is ClassPathXmlApplicationContext.

2.1.3 Dependency Injection

An application may have a few objects that work together to give users the desired outcome. In a complex Java application, the classes are made independent of other Java classes to remove tight coupling between objects, enhance the reusability of these classes, and test them independently of other classes.

As mentioned earlier, Dependency Injection is a principle that helps in implementing IoC and linking these classes together and keeping them independent at the same time.

To understand it better, let us consider two classes, Class X and Class Y. Class X has a dependency to class Y if Class X uses class Y as a variable. If dependency injection is used, then the class Y is given to class X via any of the following:

- The constructor of the class X - this is called construction injection.
- A setter - this is called setter injection.

Let us look at these two types in detail.

- **Constructor-based DI**: In this type of DI, the container invokes a class constructor with multiple arguments, each representing a dependency on the other class.

  Following code illustrates this:

```
@Configuration
public class AppConfig {

   @Bean
   public Marks marks1() {
      return new MarksImpl1();
   }
    @Bean
   public Result result() {
      return new Result(marks1());
   }
}
```

  The @Configuration annotation denotes that the class is a source of bean definitions and can be used on multiple configuration classes. The @Bean annotation is used on a method to define a bean. If a custom name is not specified, the program uses method name.

- **Setter-based DI**: In this type of DI, the container calls setter methods on the beans after invoking a no-argument constructor or no-argument static factory method to instantiate the bean.

Following code illustrates this:

```
@Bean
public Result result() {
    Result result = new Result();
    result.setMarks(mark1());
    return result;
}
```

Let's consider an example where the EmployeeServiceImpl class has an instance field employeeDao of the EmployeeDao type, a constructor with an argument, and a setEmployeeDao method. The EmployeeServiceImpl.java class contains the following code:

```
public class EmployeeServiceImpl implements EmployeeService {

  private EmployeeDao employeeDao;

  public EmployeeServiceImpl(EmployeeDao employeeDao) {
        this.employeeDao = employeeDao;
  }
  public void setEmployeeDao(EmployeeDao employeeDao) {
        this.employeeDao = employeeDao;
   }
}
```

In the EmployeeDaoImpl.java class, the following code is available:

```
public class EmployeeDaoImpl implements EmployeeDao {
  // ...
}
```

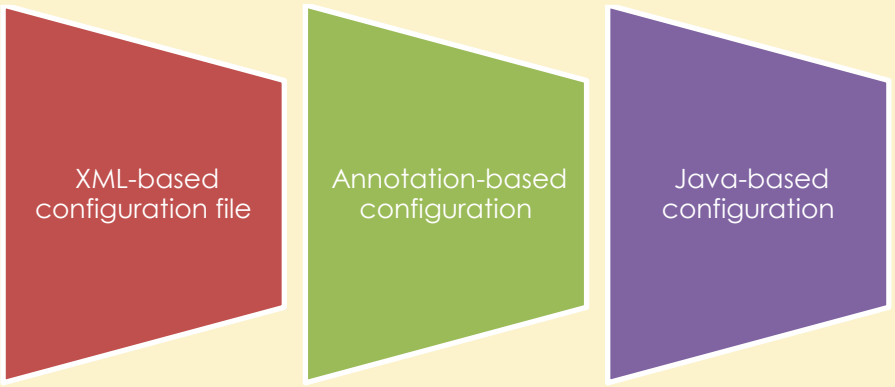
Additional Information

Users can use both the DIs, constructor-based and setter-based in a program. However, the rule of thumb states to use constructor arguments for mandatory dependencies and setters for optional dependencies.

2.1.4 Bean Definition

As stated earlier, beans are the objects that the Spring container contains, instantiates, assembles, and manages. The information about the beans and their dependencies that is provided to the container is called configuration metadata. The configuration metadata helps the container to know how to create a bean and provides the container details of a bean's lifecycle and dependencies of that bean on other classes.

Configuration metadata can be provided to the Spring container by using any of the following methods:



Configuration metadata contains details about properties, as listed in Table 2.1.

Property	Description
Autowiring mode	Used for dependency injection
Class	Bean class to be used to create the bean
Constructor-arg	Used for dependency injection
Destruction method	Method to be called after the container containing the bean is destroyed
Initialization method	Method to be called after the container sets all required bean properties
Lazy-initialization mode	Enables bean creation when it is requested rather than at the startup
Name	Specifies bean identifier uniquely
Properties	Used for dependency injection
Scope	Defines scope of the objects created from a bean definition

Table 2.1: Properties Related to Configuration

Let us now see how to define the bean. Following code demonstrates this:

```
<!-- A simple bean definition -->
  <bean id = "..." class = "...">
    <!-- collaborators and configuration for the bean-->
  </bean>
```

Beans can also be defined by using the initialization and destruction methods. Following are some sample codes:

```
<!-- A bean definition with initialization method -->
  <bean id = "..." class = "..." init-method = "...">
```

```
  <!-- collaborators and configuration for this bean go here -->
  </bean>

<!-- A bean definition with destruction method -->
  <bean id = "..." class = "..." destroy-method = "...">
    <!-- collaborators and configuration for this bean go here -->
  </bean>
```

2.1.5 Bean Definition Inheritance

Inheritance works the same in Spring as in other languages. Spring container defines any number of beans in a container and extends them, whenever required. Users can define a parent bean definition as a template and other child bean definitions can inherit the required configuration data. The configuration data may include values, properties, and methods from the parent bean definition. If a child bean definition has same method as in the parent bean definition, the child bean method overrides the parent bean method. In case of classes, the child bean can use classes defined in the parent bean. For the child bean to have class name, same properties must be defined in the parent bean.

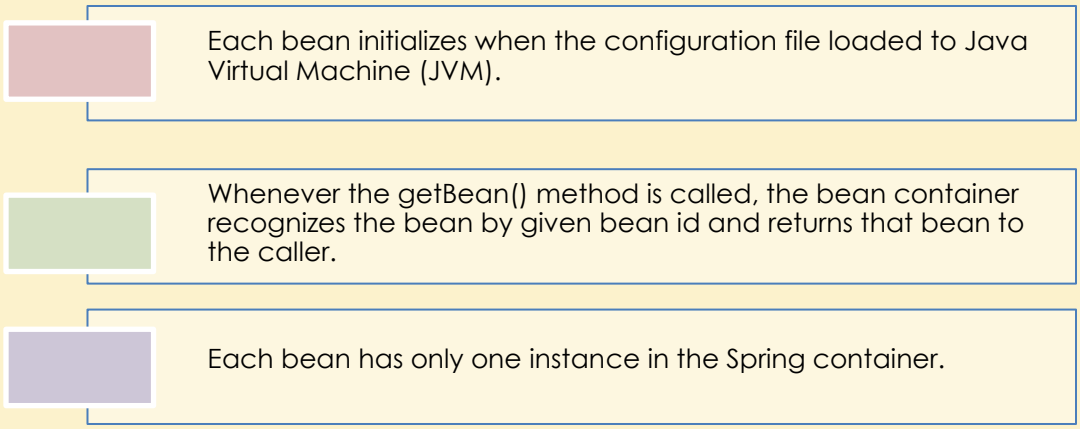
While defining a parent bean definition template, do not specify the **class** attribute, but the abstract attribute with a value of true. For example, the following code creates a parent bean definition:

<bean id = "beanTemplate" abstract = "true">

2.1.6 Bean Scope

A bean defined in the Spring container has some default configuration and behavior.

Following processing takes place according to the default behavior:



The bean scope helps to define and decide the type of bean instance to be returned from Spring container back to the caller.

Spring supports following five types of bean scopes:

- **Singleton (default)**: Returns a single bean instance per Spring IoC container.
- **Prototype**: Returns a new bean instance for any number of object instances.
- **Request:** Returns a single instance for each HTTP request. This scope type is applicable only in the context of a Web-aware Spring ApplicationContext.
- **Session:** Returns a single instance for each HTTP session. This scope type is applicable only in the context of a Web-aware Spring ApplicationContext.
- **Global Session:** Returns a single instance for a global HTTP session. This scope type is applicable only in the context of a Web-aware Spring ApplicationContext.

Following sample code shows how to define the scope property to singleton in the bean configuration file:

```
<bean class="com.javasession.sdnext.beanscope.Point" id="zeroPoint"
scope="singleton">
  <property name="x" value="0"></property>
  <property name="y" value="0"></property>
</bean>
```

Similarly, other scope types can be used in other programs.

2.1.7 Spring Bean Lifecycle

Spring bean factory manages the lifecycle of beans created through Spring container. When a bean is instantiated, the code must perform some initialization to get it into a usable state and must perform clean-up when the bean is no longer required.

The lifecycle of a bean includes a list of activities between the time of bean instantiation and its destruction.

However, the two important stages comprise bean initialization and its destruction.

The complete lifecycle of a bean is shown in Figure 2.5.

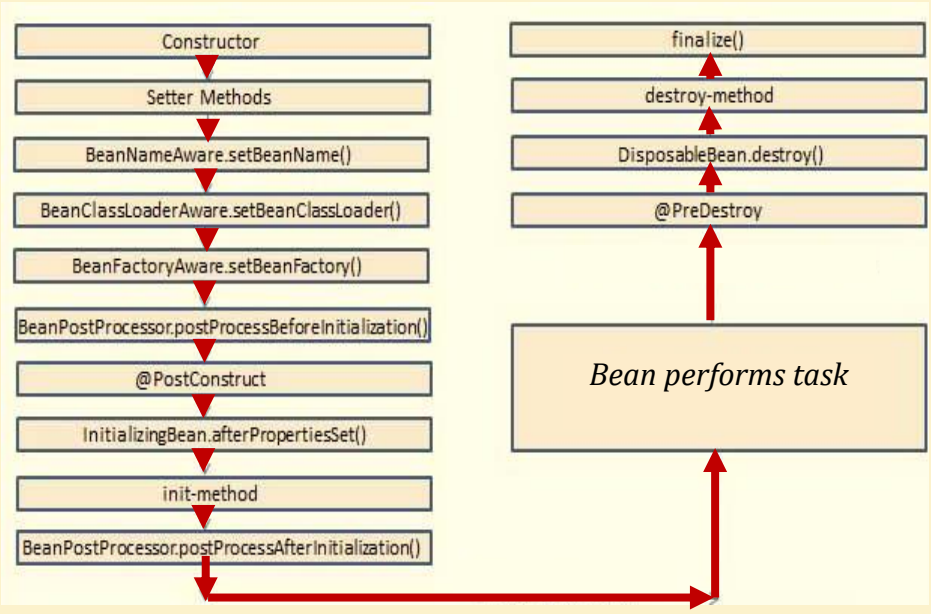


Figure 2.5: Spring Bean Lifecycle

It includes the following steps:

i. Within IoC container, create a Spring bean using class constructor.
ii. Perform the dependency injection using setter method.
iii. Once the dependency injection is completed, call the BeanNameAware.setBeanName() to set the name of bean in the bean factory that created this bean.
iv. Call the <code>BeanClassLoaderAware.setBeanClassLoader(), which supplies the bean class loader to a bean instance.
v. Call the <code>BeanFactoryAware.setBeanFactory(), which provides the owning factory to a bean instance.
IoC container calls BeanPostProcessor.postProcessBeforeInitialization on the bean to apply a wrapper on original bean.
vi. Call the method annotated with @PostConstruct.
vii. After @PostConstruct, call the method InitializingBean.afterPropertiesSet().
viii. Call the method specified by init-method attribute of bean in XML configuration.
Step vi to Step viii are called initialization callbacks.
ix. Call the method BeanPostProcessor.postProcessAfterInitialization() to apply wrapper on original bean.
x. Perform the task using the bean because the bean instance is ready for use.
xi. Use registerShutdownHook() to shut down the ApplicationContext and call the method annotated with @PreDestroy.
xii. Call the DisposableBean.destroy() method on the bean.
xiii. Call the method specified by destroy-method attribute of bean in XML configuration. Step xi to Step xiii are called destruction callbacks.
xiv. Before garbage collection, call the finalize() method of object.

2.1.8 Autowiring in Spring

The Spring container can autowire relationships between collaborating beans without using <constructor-arg> and <property> elements. This eliminates the need of writing high amount of XML configuration for a big Spring-based application.

The autowire attribute of the <bean/> element is used to specify autowire mode for a bean definition. It is recommended to use autowiring consistently instead of wiring only one or two bean definitions across a project to avoid any confusion.

Following autowiring modes as listed in Table 2.2 instruct the Spring container to use autowiring for dependency injection:

Mode	Description
no	This is the default autowiring mode and refers to no autowiring by default.
byName	Autowiring is done based on the name of the property; therefore, Spring searches for a bean with the same name as the property that needs to be set.

Mode	Description
byType	Autowiring is done based on the type of the property; therefore, Spring searches for a bean with the same name as the property that needs to be set. If more than one match is found, the framework throws an exception. Use this mode to wire arrays and other typed-collections.
constructor	Autowiring is done based on constructor arguments, therefore, Spring searches for beans with the same type as the constructor arguments. Use this mode to wire arrays and other typed-collections.
autodetect	Autowiring is first done using autowire by constructor and then by byType, if required.

Table 2.2: AutoWiring Modes

2.1.9 Factory Method

Sometimes, it is necessary for the program to instantiate a class via another class (Factory class). In this case, Spring should not create the class on its own but should delegate the instantiation to the Factory class by using the **factory-method** attribute of a bean tag. The Factory class has a predefined static method that creates the instance of the required bean.

The factory method is used in application integration where objects are constructed in a more complex way using a Factory class. Also, if third-party libraries are used in the program and these libraries use Factory classes to instantiate other classes, the use of the **factory-method** attribute simplifies the integration of Spring applications with the third-party libraries.

Following sample code displays how to instantiate a class using the factory method:

```
<bean id="userService" class="com.concretepage.UserService" factory-
method="createInstance">
```

2.2 Aspect-Oriented Programming

One of the key components of the Spring framework is the Aspect-oriented Programming (AOP) framework.

DI helps to decouple the application objects from each other and AOP separates general code from aspects that cross the boundaries of an object or a layer. For example, the application log is not tied to any application layer. It applies to the complete program and should be present everywhere. This is called a crosscutting concern.

Before using AOP, it is recommended to understand the following terms:

- **Aspect**: A subprogram is associated with a specific property and is scattered across methods, classes, object hierarchies, or even object models in a program. As that property varies, the effect ripples through the entire program.

- **Join Point**: A point in the application where the AOP aspect can be plugged-in.

- **Advice**: The actual action to be taken either before or after the method execution.

  There are following types of advices:
  - **Before advice**: Executes before a join point, but that does not have the ability to prevent execution flow proceeding to the join point (unless it throws an exception).
  - **After returning advice**: Is executed after a join point completes. For example, if a method returns without throwing an exception.
  - **After throwing advice**: Is executed if a method exits by throwing an exception.
  - **After (finally) advice**: Is executed regardless of how a join point exits (normal or exceptional return).
  - **Around advice**: Surrounds a join point, such as a method invocation. This is the most powerful and general kind of advice.

- **Pointcut**: A set of one or more join points where an advice should be executed.

- **Introduction**: A place to add new methods or attributes to the existing classes.

- **Target Object**: The object being advised by one or more aspects.

- **Weaving**: The process of linking aspects with other application types or objects to create an advised object.

Spring AOP can be used by using any of the following three approaches but the widely used approach is Spring AspectJ Annotation Style:
- Spring1.2 Old style
- AspectJ annotation-style
- Spring XML configuration-style (schema based)

Following sample code displays an AOP Spring example, schooldemo with an assumption that we have already created the required aspects and data source and instantiated the Spring beans:

```
package schooldemo;

import org.springframework.context.support.ClassPathXmlApplicationContext;
import schooldemo.TeacherService;
```

```
public class SpringMain {
        public static void main(String[] args) {
        try {
                ClassPathXmlApplicationContext context = new
                                ClassPathXmlApplicationContext("spring.xml");
                TeacherService teacherService = context.getBean("teacherService",
                                                TeacherService.class);
                System.out.println(teacherService.getTeacher().getName());
                teacherService.getTeacher().setName("Emma Timmons");
                context.close();
        }
        catch(Exception e){
                // Display an error message to the user
                System.out.println("Error: "+ e.getMessage());
                }
        }
}
```

2.3 New Features in Spring 5.0

With each release, a set of new features is added in the product with discontinuation of some of the existing features and support. Spring Framework 5.0 is released with the following new features:

- **JDK Baseline Update**: The Spring Framework 5.0 codebase is released to run on Java 8. Initially, Spring 5.0 was expected to release on Java 9. However, with the delay in Java 9 release, the Spring framework 5.0 release is decoupled from Java 9.
- **Reactive Programming Model**: Spring 5.0 release provides the reactive stack Web framework to support reactive programming. This makes Spring 5.0 suitable for event-loop style processing that can scale with a small number of threads. Reactive programming is programming with asynchronous data streams. A stream is a sequence of ongoing events ordered in time. The Reactive Streams API will be a part of Java 9. In Java 8, users need to include a dependency for the Reactive Streams API specification.
- **Testing Improvements**: Spring 5.0 supports JUnit4 as well as new JUnit 5 Jupiter to write tests and extensions in JUnit 5. The Jupiter sub-project not only provides a programming and extension model but also provides a test engine to run Jupiter based tests on Spring.
- **Upgraded Library Support**: Spring 5.0 supports the following upgraded library versions:
    - Jackson 2.6+
    - •EhCache 2.10+ / 3.0 GA
    - •Hibernate 5.0+
    - •JDBC 4.0+
    - •XmlUnit 2.x+
    - •OkHttp 3.x+
    - •Netty 4.1+

- **Functional Programming with Kotlin**: Spring 5.0 supports JetBrains Kotlin language, which is a statically typed programming language for modern multiplatform applications. Kotlin runs on top of the JVM. With Kotlin support, developers can use functional Spring programming for functional Web endpoints and bean registration.
- **Core Container Support**: Spring 5.0 now supports Candidate Component Index (CCI) instead of classpath scanning. This support helps to bypass the candidate component identification step in the classpath scanner. This enhancement does not have significant differences for small projects with less than 200 classes. However, it has significant impacts on large projects.
- **Spring Webflux Support**: Spring 5.0 is released with a new Spring Web flux module that supports reactive HTTP and WebSocket clients. The new Spring Framework also supports reactive Web applications on servers, such as REST, HTML, WebSocket, and so on.

Summary

- IoC is a conceptual design principle in which the control of objects or custom-written portions of a program is transferred to a framework.
- IoC provides greater modularity for programs.
- The Spring container is the core component of Spring Framework that implements IoC.
- DI can be of following two types:
  - Constructor-based DI
  - Setter-based DI
- In Spring, beans are the objects that the Spring container contains, instantiates, assembles, and manages.
- Spring supports AOP framework.
- Spring Framework 5.0 provides the following new features:
  - JDK baseline update
  - Reactive programming model
  - Testing improvements
  - Upgraded library support
  - Functional programming with Kotlin
  - Core container support
  - Spring Webflux support

Check Your Progress

1. Which of the following is related to Spring Framework?

A.	Container	**B.**	Beans
C.	IoC	**D.**	All of these

2. Which of the following is not the Spring supported DI?

A.	Constructor-based	**B.**	Setter-based
C.	Destructor-based	**D.**	All of these

3. The bean scope helps to define and decide the type of bean instance to be returned from _____ back to the _____.

A.	Caller, container	**B.**	Abstracts, caller
C.	Container, caller	**D.**	Classes, caller

4. A bean contains the information for the container. This information is called _____.

A.	Metadata	**B.**	Configuration metadata
C.	Configuration classes	**D.**	Properties

5. Which JDK version is the Spring Framework 5.0 based on?

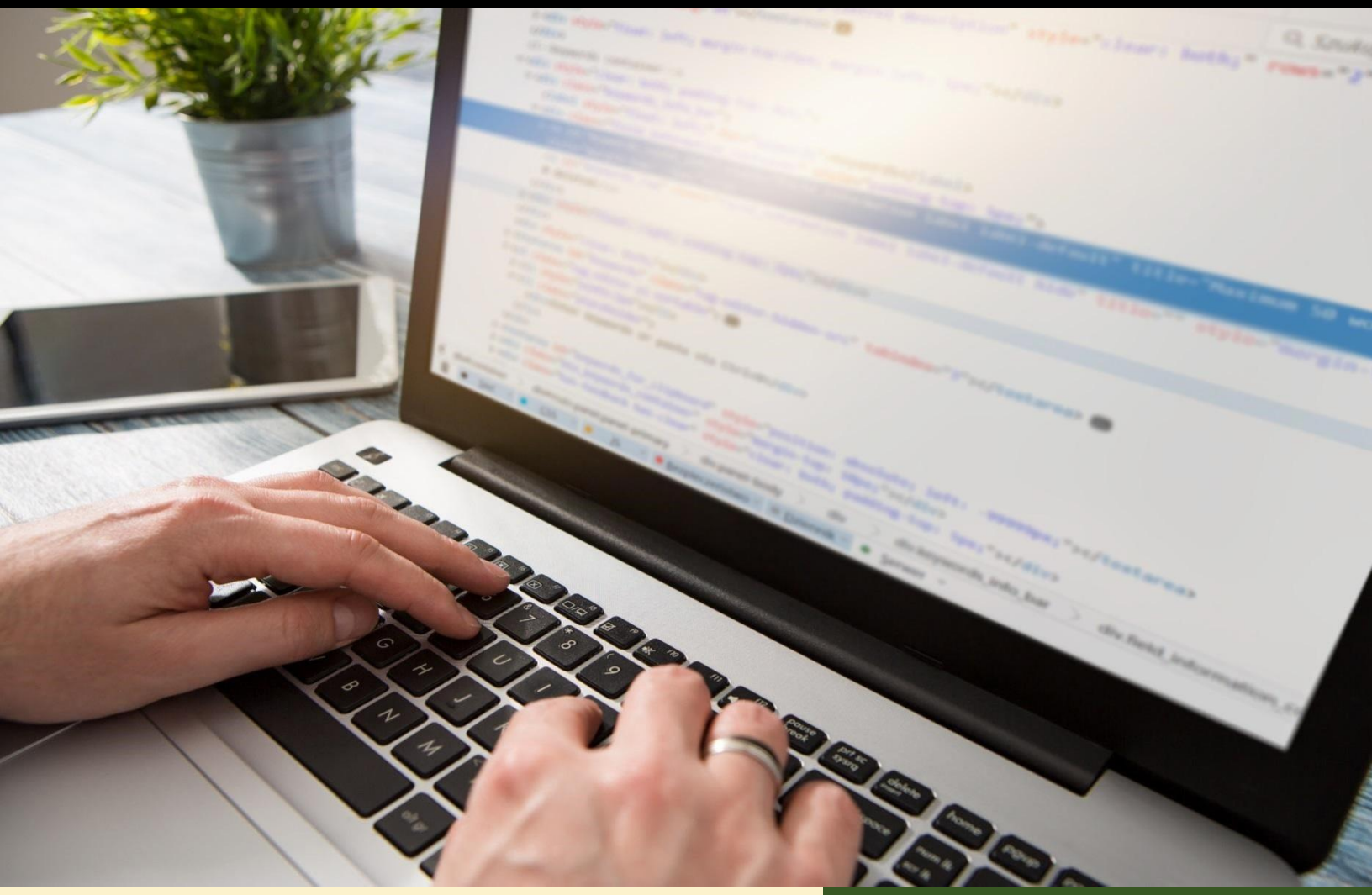
A.	Java 8	**B.**	Java 7
C.	Java 6	**D.**	Java 5

Answers

1.	D
2.	C
3.	C
4.	B
5.	A

Session 3

Spring Data Access



Objectives

Welcome to the session, Spring Data Access.

This session describes various concepts involved in the database and programming techniques support provided by Spring Framework.

In this session, students will learn to:

- Explain data access with Spring
- Describe Java Database Connectivity (JDBC) and Object-Relational Mapping (ORM) with MySQL
- Describe how to use ORM programming techniques

3.1 Accessing Data with Spring

Spring Framework provides programming support to access and manipulate data in databases.

3.1.1 Spring Transaction Support

For RDBMS-oriented applications, transaction management is an important activity to ensure data integrity and consistency. A database transaction must have following four key properties:

- **Atomicity**: Database modifications must follow an all or nothing rule. A transaction involves several low-level operations. As per this property, a transaction must be treated as an atomic unit, that is, either all its operations are executed or else, none are executed. There must be no state in database where the transaction is partially complete.
- **Consistency**: The database should remain in a consistent state before the start of the transaction and after its successful/unsuccessful completion. Due to some transaction failure, sometimes the database may get partially updated which means that there will be inconsistency. This needs to be avoided by taking appropriate measures.
- **Isolation**: Each transaction is unaware of other transactions executing concurrently in the system. This is required to maintain the performance as well as the consistency between transactions in a database.
- **Durability**: Once a transaction is successfully completed in a database and a notification is received, the transaction should always persist in database and should not be undone even if the system fails and restarts.

These characteristics are often abbreviated as ACID.

Spring Framework provides transaction capabilities to Plain Old Java Objects (POJOs), thereby, serving as an alternate to Enterprise JavaBeans (EJB) transactions. EJB is a managed, server-side software that helps developers to create modular enterprise applications. EJB is one among several Java APIs.

Spring Framework provides comprehensive transaction management that offers following advantages:

- Supports declarative transaction management, which is a non-programmatic demarcation of transaction boundaries. This allows to separate transaction management from the business code. In simple words, through declarative transaction management, developers can manage the transaction with the help of configuration instead of hard-coding in their source code. Declarative transaction management is preferred over programmatic transaction management though it is less flexible than the latter which allows developers to control transactions through their code. Declarative transaction management can be modularized with the AOP approach. Spring Framework declarative transaction management can be applied to any class, not just special classes such as EJBs.

- Supports simpler API for programmatic transaction management than complex transaction APIs such as JTA. The key to the Spring transaction abstraction is defined by the org.springframework.transaction.PlatformTransactionManager interface.

- Provides developers with a consistent programming model across several transaction APIs, such as Java Transaction API (JTA), JDBC, Hibernate, Java Persistence API (JPA), and Java Data Objects (JDO). This reduces the need to use different local and global programming models.

- Integrates well with Spring's data access abstractions.

3.1.2 DAO Support

A Data Access Object (DAO) is an object, which performs some specific data operations without revealing the details of the database to the user. This isolation separates the public interface of DAO from its implementation. The public interface may comprise data access methods that an application needs for the domain-specific objects and data types. The implementation comprises the way the domain-specific objects and data type needs are fulfilled with a specific DBMS and database schema.

Spring provides a DAO framework which contains interfaces, utility classes, and abstract classes. Developers have a choice to use either XML based configuration or annotation-based configuration to bind custom classes to Spring DAO framework. The latter makes it easy for the Spring application to work with different data access technologies, such as Hibernate, JDBC, JdbcTemplate, JPA, and JDO. In case, the Spring application uses a combination of data access technologies, configuration emphasizes on loose coupling between custom classes and the framework. This enables developers to easily switch between the two. In addition, extending these classes in their Spring application allows developers to write generic code, without generating errors that are specific to each data access technology included in the application.
Table 3.1 lists some of the abstract classes that are supported by Spring and provide the data source and other configuration settings specific to the relevant data-access technology.

Class	Description
JdbcDaoSupport	It is a super class for JDBC data access objects and requires a DataSource instance to be provided.
HibernateDaoSupport	It is a super class for Hibernate data access objects and requires a SessionFactory instance to be provided.
JdoDaoSupport	It is a super class for JDO data access objects and requires a PersistenceManagerFactory instance to be provided.
JpaDaoSupport	It is a super class for JPA data access objects and requires an EntityManagerFactory instance to be provided.

Table 3.1: Abstract Classes for DAO Support in Spring

Spring provides a convenient and easy translation from exceptions, such as SQLException (which are specific to a technology) to its own exception class hierarchy.

In this translation, DataAccessException acts as the root exception.

Figure 3.1 shows a subset of the DataAccessException hierarchy that Spring provides.

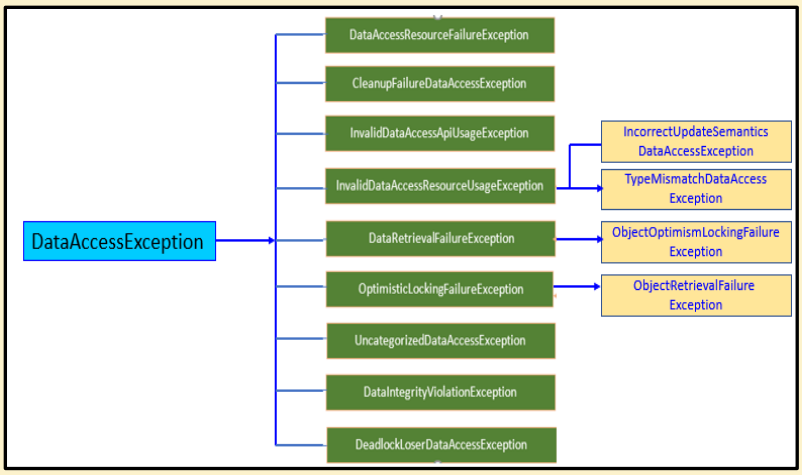


Figure 3.1: Exception Class Hierarchy in Spring for DAO Support

3.2 DAO Design Pattern

DAO design pattern is built on abstraction and encapsulation design principles to protect the application from any change developers choose to make in their database and technology. Consider an example wherein the MVC application authenticates a user using credentials from an on-premises database. Due to changing trends, developers have now upgraded to a cloud-based database. Normally, this would mean that their entire application would be affected by this minor change. However, if their application is developed using DAO design pattern, it will be easy and safe because the changes need to be made only to the data access layer keeping the view layer intact.

DAO pattern has the following members:

Data Access Object Interface	Defines the standard operations to be performed on a model object.
Data Access Object Concrete Class	Implements the DAO interface and is responsible to get data from a data source (database/XML or any other storage mechanism).
Model Object or Value Object	Refers to a simple POJO that contains get/set methods to store data, which is retrieved using DAO class.

For example, a developer can create a program that uses DAO pattern and displays names and IDs of employees of a company.

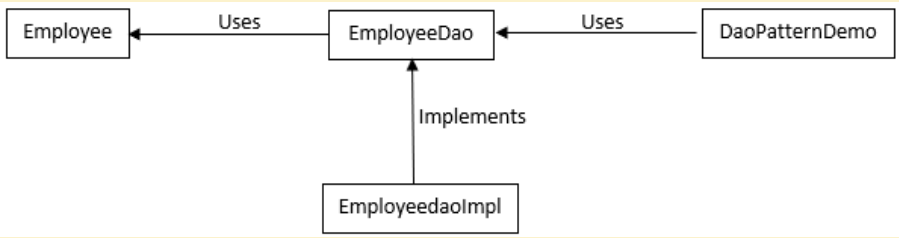


Figure 3.2: DAO Design Pattern Example

Here, a developer can create an Employee object acting as a Model or Value Object. EmployeeDao is Data Access Object Interface. EmployeedaoImpl is a concrete class implementing Data Access Object Interface. DaoPatternDemo, the class, will use EmployeeDao to demonstrate the use of Data Access Object pattern. Refer to Figure 3.2.

3.3 JDBC

Developers can use JDBC to perform database access in two-tier as well as three-tier architectures.

In a two-tier architecture having a client/server setup, a Java application interacts directly with the data source. Refer to Figure 3.3. This requires a JDBC driver to communicate with a data source.

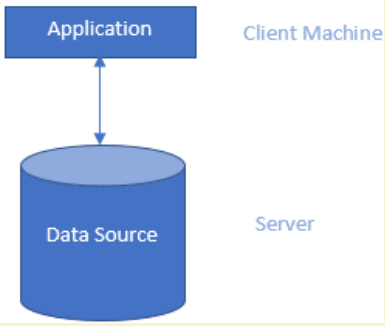


Figure 3.3: Two-Tier Architecture

In contrast to two-tier architecture, in three-tier architecture, an application and a data source communicate with each other through a middle tier (service tier).

The application sends commands to a middle tier first and then these commands are sent to the data source.

The data source also processes the commands and sends the results back to the middle tier first and then these results are sent back to the application. Refer to Figure 3.4.

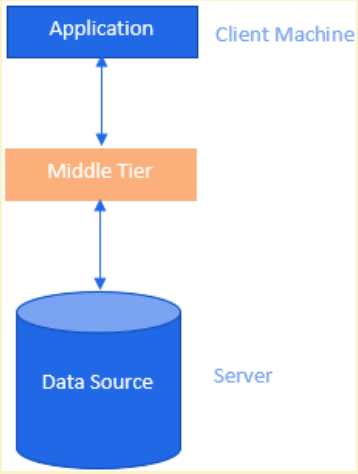


Figure 3.4: Three-Tier Architecture

To build a JDBC application, developers need to perform the following steps:

1. Import the package containing the JDBC classes needed for database programming.
2. Register the JDBC driver to open a communications channel with the database.
3. Open a connection to create a Connection object.
4. Execute a query by using an object of type Statement or PreparedStatement for submitting a database statement.
5. Extract data from result set to fetch data from the database.
6. Close all database resources to clean up the environment.

3.4 Spring JDBC Packages

Spring Framework JDBC makes it easier to work with JDBC in Spring applications. Spring Framework provides several in-built classes for JDBC that can be extended. The most common and popular is the JdbcTemplate class. This class helps to manage low-level operational activities, such as opening and closing connections, preparing and executing SQL statements, handling process exceptions, and so on.

Spring JDBC classes come under following four packages:

- **core**: This package has classes for the core functionality of JDBC. Some of the important classes include NamedParameterJdbcTemplate, JdbcTemplate, SimpleJdbcInsert, and SimpleJdbcCall.
- **datasource**: This package has utility classes to access data source. In addition, this package includes various data source implementations that can be used to test JDBC code beyond the Java EE container.
- **object**: This package has classes facilitating database operations such as queries and updates. It allows executing queries and returning the results as a business object.
- **support**: This package has support classes for the other classes defined under core and object packages.

3.5 Using JDBC With Spring

Developers can use JDBC with Spring Framework by performing the following steps:

1. Create a data source in the database. For example, one could create a database and a table in MYSQL. An easy way to do this is by using XAMPP, a free open-source cross platform Web server solution. Download the software from https://www.apachefriends.org/download.html and install it. Launch it and ensure Apache and MYSQL are running. Create a MYSQL database and table through the Web interface.
2. Create a Java project using STS and a package under the src folder in the created project.
3. Include required Spring libraries using Add External JARs option.
4. Download and add Spring JDBC specific latest libraries mysql-connector-java.jar, org.springframework.jdbc.jar, and org.springframework.transaction.jar in the project.
5. Create the DAO interface and identify all the required methods.
6. Create other required Java classes and the Beans configuration file titled **Beans.xml**, under the src folder.
7. Create the content of all the Java files and Bean Configuration file and run the application.

3.5.1 Using theJdbcTemplate Class

Using the JdbcTemplate class in an application eliminates the need to write code for operational activities. After using this class in an application, developers just need to define the connection parameters and specify the SQL statement.

The JdbcTemplate class performs the following functions:

- Executes SQL queries
- Updates statements
- Stores procedure calls
- Performs iteration over ResultSets

➢ Extracts returned parameter values

Additionally, the JdbcTemplate class helps to catch exceptions in applications and map them to the corresponding exception hierarchy categories defined in the org.springframework.dao package.

The JdbcTemplate class provides the following methods:

Method	Description
public int update(String query)	Helps to insert, update, and delete records.
public int update(String query, Object... args)	Helps to insert, update, and delete records using PreparedStatement using given arguments.
public void execute(String query)	Helps to execute DDL query.
public T execute(String sql, PreparedStatementCallback action)	Helps to execute the query by using PreparedStatement callback.
public T query (String sql, ResultSetExtractor rse)	Helps to fetch records using ResultSetExtractor.
public List query(String sql, RowMapper rse)	Helps to fetch records using RowMapper.

Table 3.2: Methods of JdbcTemplate Class

Assuming that developers have already created their data source, to use the JdbcTemplate class in their Spring application, they need to perform the following steps:

➢ Configure the data source credentials in the Spring configuration file
➢ Dependency-inject the shared data source bean into the DAO classes
➢ Create the JdbcTemplate in the setter for the data source

The MYSQL code for the table could be as follows:

```
create table student(
id bigint,
roll varchar(20),
name varchar(50),
marks integer,
primary key(id)
);
```

Code for the Student model (Student.java) is as follows:

```
//Creating Student.java file

package com.demo.model;

public class Student {
```

```
	private Long id;
	private String roll;
	private String name;
	private Integer marks;
	public Student() {}

	public Student(Long id, String roll, String name, Integer marks) {
		super();
		this.id = id;
		this.roll = roll;
		this.name = name;
		this.marks = marks;
	}

	public Long getId() {
		return id;
	}

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

	public String getRoll() {
		return roll;
	}

	public void setRoll(String roll) {
		this.roll = roll;
	}

	public String getName() {
		return name;
	}

	public void setName(String name) {
		this.name = name;
	}

	public Integer getMarks() {
		return marks;
	}

	public void setMarks(Integer marks) {
		this.marks = marks;
	}
}
```

Code for the DAO implementation (StudentDAO.java) using JdbcTemplate is as follows:

```
// Creating StudentDAO.java with JdbcTemplate framework

package com.demo;

import org.springframework.jdbc.core.JdbcTemplate;

import com.demo.model.Student;

public class StudentDAO {

	private JdbcTemplate jdbcTemplate;

	public void setJdbcTemplate(JdbcTemplate jdbcTemplate) {
		this.jdbcTemplate = jdbcTemplate;
	}

	public int saveStudent(Student s) {
		String query = "insert into student values('" + s.getId() + "','" + s.getName()
+ "','" + s.getMarks() + "')";
		return jdbcTemplate.update(query);
	}

	public int updateStudent(Student s) {
		String query = "update student set name='" + s.getName() + "',marks='" +
s.getMarks() + "' where id='"
				+ s.getId() + "' ";
		return jdbcTemplate.update(query);
	}

	public int deleteStudent(Student s) {
		String query = "delete from student where id='" + s.getId() + "' ";
		return jdbcTemplate.update(query);
	}
}
```

Code for applicationContext.xml which will use the JdbcTemplate object in the StudentDAO class by passing the setter method is as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
	xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
	xmlns:p="http://www.springframework.org/schema/p"
	xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">
```

```
	<bean id="ds"
class="org.springframework.jdbc.datasource.DriverManagerDataSource">
		<property name="driverClassName" value="com.mysql.jdbc.Driver" />
		<property name="url" value="jdbc:mysql://localhost:3306/springdb" />
		<property name="username" value="root" />
		<property name="password" value="" />
	</bean>
	<bean id="jdbcTemplate"
class="org.springframework.jdbc.core.JdbcTemplate">
		<property name="dataSource" ref="ds"></property>
	</bean>
	<bean id="sdao" class="com.demo.StudentDAO">
		<property name="jdbcTemplate" ref="jdbcTemplate"></property>
	</bean>
</beans>
```

Code for creating the Demo.java class that gets the bean from the applicationContext.xml file and calls the saveStudent() method is as follows:

```
package com.demo;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

import com.demo.model.Student;

public class Demo {

	public static void main(String[] args) {
		ApplicationContext ctx = new
ClassPathXmlApplicationContext("applicationContext.xml");
		StudentDAO dao = (StudentDAO) ctx.getBean("sdao");
		int status = dao.saveStudent(new Student(1008l, "ECE/21/2017-18",
"John", 1098));
		System.out.println(status);
	}
}
```

Make sure that the pom.xml of the project includes the following code snippet (check the codes in bold):

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

```
	<artifactId>session-3</artifactId>
	<version>0.0.1-SNAPSHOT</version>
	<properties>
		<!-- Generic properties -->
		<java.version>1.8</java.version>
		<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
		<project.reporting.outputEncoding>UTF-
8</project.reporting.outputEncoding>
		<!-- Spring -->
		<spring-framework.version>3.2.3.RELEASE</spring-framework.version>
		<!-- Hibernate / JPA -->
		<hibernate.version>4.2.1.Final</hibernate.version>
		<!-- Logging -->
		<logback.version>1.0.13</logback.version>
		<slf4j.version>1.7.5</slf4j.version>
		<!-- Test -->
		<junit.version>4.11</junit.version>
	</properties>
	<dependencies>
		<!-- Spring and Transactions -->
		<dependency>
			<groupId>org.springframework</groupId>
			<artifactId>spring-context</artifactId>
			<version>${spring-framework.version}</version>
		</dependency>
		<dependency>
			<groupId>org.springframework</groupId>
			<artifactId>spring-tx</artifactId>
			<version>${spring-framework.version}</version>
		</dependency>
		<dependency>
			<groupId>org.springframework</groupId>
			<artifactId>spring-jdbc</artifactId>
			<version>${spring-framework.version}</version>
		</dependency>
		<!-- Logging with SLF4J & LogBack -->
		<dependency>
			<groupId>org.slf4j</groupId>
			<artifactId>slf4j-api</artifactId>
			<version>${slf4j.version}</version>
			<scope>compile</scope>
		</dependency>
		<dependency>
			<groupId>ch.qos.logback</groupId>
			<artifactId>logback-classic</artifactId>
			<version>${logback.version}</version>
			<scope>runtime</scope>
```

```
            </dependency>
            <dependency>
                <groupId>mysql</groupId>
                <artifactId>mysql-connector-java</artifactId>
                <version>5.1.6</version>
                <scope>runtime</scope>
            </dependency>
            <!-- Hibernate -->
            <dependency>
                <groupId>org.hibernate</groupId>
                <artifactId>hibernate-entitymanager</artifactId>
                <version>${hibernate.version}</version>
            </dependency>
            <!-- Test Artifacts -->
            <dependency>
                <groupId>org.springframework</groupId>
                <artifactId>spring-test</artifactId>
                <version>${spring-framework.version}</version>
                <scope>test</scope>
            </dependency>
            <dependency>
                <groupId>junit</groupId>
                <artifactId>junit</artifactId>
                <version>${junit.version}</version>
                <scope>test</scope>
            </dependency>
        </dependencies>
</project>
```

Build and execute the application. Observe the database and table for the changes made. One row will be inserted into the table with the given data.

JDBC Batch Processing in Spring

JDBC supports batch processing, which means executing a batch (group) of queries and submitting them with one call to the database instead of executing a single query to enhance the processing performance.

java.sql.Statement and java.sql.PreparedStatement are batch processing interfaces. Both the interfaces and the corresponding classes based on the JDBC Driver of the Database are used to execute queries on the database. First, using addBatch(), one can add all SQL queries to a batch and then execute those SQL queries using executeBatch().

Following code shows how to establish a connection before calling the batch processing methods:

```
import java.sql.Connection;
import java.sql.DriverManager;
```

```
import java.sql.PreparedStatement;
import java.sql.SQLException;
. . .
String driver = "com.mysql.jdbc.Driver";
String connection = "jdbc:mysql://localhost:3306/mydatabase";
String user = "root";
String password = "";
Class.forName(driver);
Connection con;
con = DriverManager.getConnection(connection, user, password);
. . .
```

One may need to include relevant statements that create the connection within try-catch blocks in order to handle potential exceptions.

Following code snippet shows creating and executing a batch using Statement and how JdbcTemplate is used to execute a batch:

```
public int[] doBatchInsert() {
        String[] queries = new String[3];
        queries[0] = "INSERT INTO STUDENT" + "(ID, NAME, MARKS) VALUES
(1,'ECE/21/2017-18','StudentName1',540)";
        queries[1] = "INSERT INTO STUDENT" + "(ID, NAME, MARKS) VALUES
(2,'EEE/27/2017-18', 'StudentName2',570)";
        queries[2] = "INSERT INTO STUDENT" + "(ID, NAME, MARKS) VALUES
(3,'ECE/28/2017-18', 'StudentName3',590)";
        return jdbcTemplate.batchUpdate(queries);
}
```

Following code snippet shows usage of JDBC Statement to execute batch:

```
Statement = con.createStatement();
statement.addBatch("INSERT INTO STUDENT" + "(ID, NAME, MARKS) VALUES
(1,'StudentName',240)";
statement.addBatch("INSERT INTO STUDENT_ADDRESS" + "(ID, STU_ID, ADDRESS)
VALUES ('10','1','Address')");
statement.executeBatch();
```

Following code snippet shows how JDBC PreparedStatement is used to execute a batch:

```
String insertStudent = "INSERT INTO student ("
                        + id, roll, name, marks" ") VALUES(?, ?, ?, ?)";
PreparedStatement preparedStatement = con.prepareStatement(insertTableSQL);
preparedStatement.setLong(1, 101);
preparedStatement.setString(2, "ECE/20/2017-1");
preparedStatement.setString(3, "student 1");
```

```
preparedStatement.setInt(4, 540);
preparedStatement.addBatch();
preparedStatement.setLong(1, 102);
preparedStatement.setString(2, "EEE/29/2017-1");
preparedStatement.setString(3, "student 2");
preparedStatement.setInt(4, 570);
preparedStatement.addBatch();
preparedStatement.executeBatch();
con.commit();
```

Build and execute the application. Observe the database and table for the changes made.

3.6 MySQL Data Access with ORM

MySQL is an open-source Relational Database Management System (RDBMS) that uses Structured Query Language (SQL). SQL is the one of the most popular languages for adding, accessing, and managing content in a database. Spring Framework provides programming support with MySQL database.

The Spring programs can use several programming paradigms for data handling. Object-Relational Mapping (ORM) is one of them. MySQL can be combined with ORM through the Spring Framework.

3.6.1 Introducing ORM

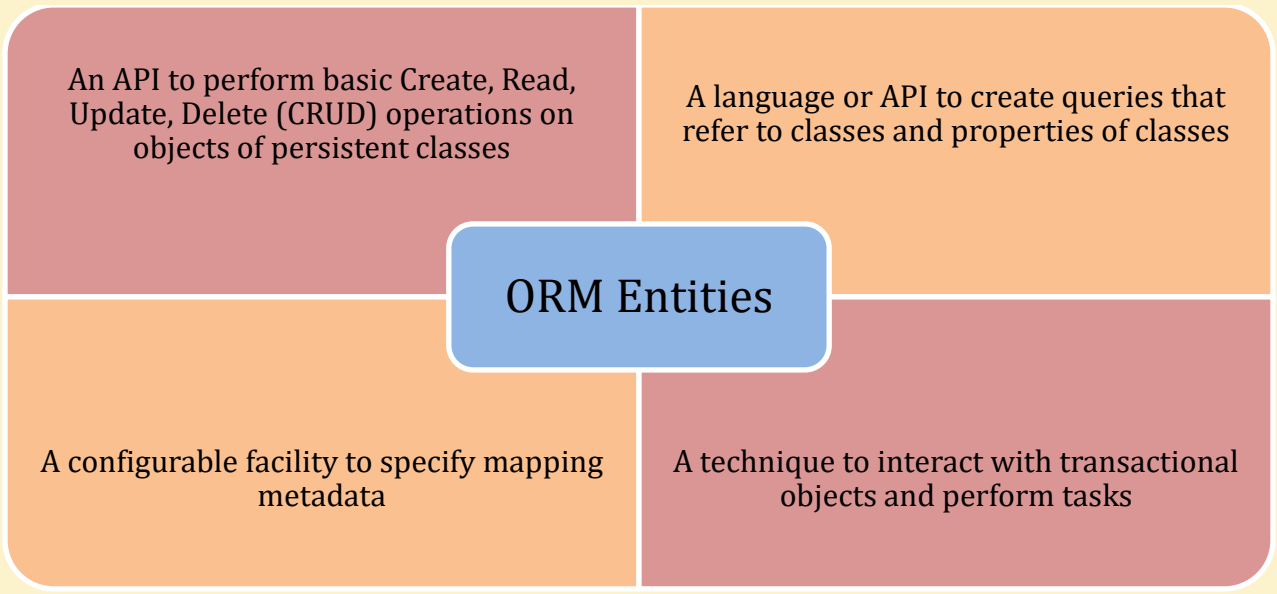
In simple terms, ORM is a programming technique based on OO principles and helps to convert data between incompatible or conflicting type systems in relational databases, XML repositories, other data sources, and OOP languages.

An ORM library is written in the same language that can be used to ease communication and interaction with an object. ORM also provides the capability to incorporate modern technology and capability without making any changes to the code of an application.

ORM provides the following advantages:

- Saves a lot of maintenance effort because the data model is stored at one place, which makes it easier to update, maintain, and reuse the code.
- Is more flexible to use because it fits in the natural way of coding, abstracts the database system. Developers can change it as per requirements, and the model is weakly bound to the rest of the application, so can be changed, or used anywhere else.

An ORM solution consists of four entities, as shown here:



Java supports the following persistent frameworks and ORM options:
- Castor
- Entity EJBs
- TopLink
- Java Data Objects
- Spring DAO
- Hibernate

The persistent framework stores and retrieves objects from a relational database and is an ORM service.

3.6.2 Introducing Hibernate

Hibernate is a powerful and high performance ORM solution created by Gavin King in 2001 for Java applications. Hibernate maps Java classes to database tables and Java data types to SQL data types and offers data query and retrieval facilities to reduce the amount of common data persistence-related programming tasks for developers. One of the best things of using Hibernate is that it does not need any application server to operate.

Hibernate replaces direct and persistence database accesses with high-level object handling functions.

In addition, Hibernate provides the following advantages:
- Maps Java classes to database tables using XML files without writing the code.
- Provides simple APIs to store and retrieve Java objects to and from the database.

- Provides code independence by enabling to change only XML file properties if there is change in the database or in any table.
- Manipulates complex associations of database objects.
- Supports smart fetching strategies that helps to minimize database access.

Hibernate supports almost any database with a JDBC driver, such as HSQL Database Engine, DB2/NT, MySQL, PostgreSQL, FrontBase, Oracle, and so on. Additionally, it supports a variety of technologies, such as XDoclet Spring, J2EE, Eclipse plug-ins, and Maven.

Hibernate has layered architecture, which helps to operate without knowing about the underlying APIs. Hibernate uses the database to provide persistence services and objects to the application. Figure 3.5 shows the Hibernate application architecture.

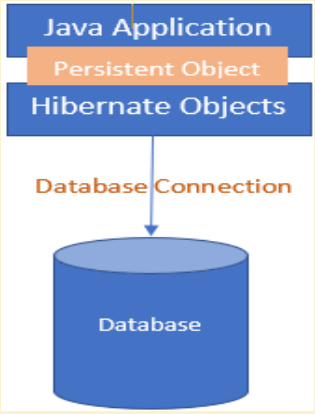


Figure 3.5: Hibernate Architecture

The Hibernate architecture has the following objects:

- **Configuration**: The Configuration object is usually created only once during application initialization. This object provides the following two keys components:
  - Database Connection
  - Class Mapping Setup
- **SessionFactory**: Configuration object creates the SessionFactory object to configure Hibernate for the application using the supplied configuration file.
- **Session**: The Session object is used to setup a physical connection with a database.
- **Transaction**: This is an optional object because Hibernate applications can manage transactions in their own application code.
- **Query**: The Query object uses SQL or Hibernate Query Language (HQL) string to retrieve data from the database and create objects.
- **Criteria**: The Criteria object is used to create and execute object-oriented criteria queries to retrieve objects.

3.6.3 Integrating Hibernate with Spring Framework

Developers can integrate a Hibernate application with Spring Framework. In Hibernate framework, they need to provide all the database information in the hibernate.cfg.xml file. Whereas, when integrating the Hibernate application with Spring, they do not need to create the hibernate.cfg.xml file, but just provide all the information in the applicationContext.xml file.

Here are code snippets demonstrating how Hibernate is integrated in Spring applications. The same 'student' table created earlier can be used.

Following code snippet shows the Hibernate entity:
(File is saved as com.demo.hibernate.model.Student.java)

```
package com.demo.hibernate.model;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.Table;
@Entity
@Table(name = "student")
public class Student {
	@Id
	@Column(name = "id", nullable = false)
	private Long id;

	@Column(name = "roll")
	private String roll;

	@Column(name = "name")
	private String name;

	@Column(name = "marks")
	private Integer marks;

	public Long getId() {
		return id;
	}
	public void setId(Long id) {
		this.id = id;
	}
	public String getRoll() {
		return roll;
	}
	public void setRoll(String roll) {
		this.roll = roll;
	}
	public String getName() {
		return name;
	}
	public void setName(String name) {
		this.name = name;
	}
	public Integer getMarks() {
		return marks;
```

```
	}
	public void setMarks(Integer marks) {
		this.marks = marks;
	}
	@Override
	public String toString() {
		return "Student [id=" + id + ", roll=" + roll + ", name=" + name + ", marks=" +
marks + "]";
	}
}
```

It is recommended that interface-based programming is the best practice because it provides loose coupling between implementation and consumer class. Following code snippet shows the Hibernate DAO (saved as com.demo.hibernate.StudentDAO.java):

```
package com.demo.hibernate;

import com.demo.hibernate.model.Student;

public interface StudentDAO {
	Student getStudent(final long id);
	void save(final Student student);
}
```

Following code snippet shows the Hibernate DAO implementation (saved as StudentDAOImpl.java) of the StudentDAO interface:

```
package com.demo.hibernate;

import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;

import com.demo.hibernate.model.Student;

public class StudentDAOImpl implements StudentDAO {

	private SessionFactory sessionFactory;

   public void setSessionFactory(SessionFactory sessionFactory) {
      this.sessionFactory = sessionFactory;
   }
	public Student getStudent(final long id) {
		Session session = this.sessionFactory.openSession();
		return (Student)session.get(Student.class, id);
	}
```

```
	public void save(final Student student) {
		Session session = this.sessionFactory.openSession();
		Transaction tx = session.beginTransaction();
		session.persist(student);
		tx.commit();
		session.close();
	}
}
```

Following code shows the configuration file, **spring.xml**, which acts as glue between customer application and Spring framework:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
	xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
	xmlns:aop="http://www.springframework.org/schema/aop"
	xmlns:tx="http://www.springframework.org/schema/tx"
	xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
		http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop-4.0.xsd
		http://www.springframework.org/schema/tx
http://www.springframework.org/schema/tx/spring-tx-4.0.xsd">

	<bean id="dataSource"
class="org.springframework.jdbc.datasource.DriverManagerDataSource" destroy-
method="">
		<property name="driverClassName" value="com.mysql.jdbc.Driver" />
		<property name="url" value="jdbc:mysql://localhost:3306/mydatabase"
/>
		<property name="username" value="root" />
		<property name="password" value="" />
	</bean>
	<!-- Hibernate 4 SessionFactory Bean definition -->
	<bean id="hibernate4AnnotatedSessionFactory"
		class="org.springframework.orm.hibernate4.LocalSessionFactoryBean" >
		<property name="dataSource" ref="dataSource" />
		<property name="annotatedClasses">
			<list>
				<value>com.demo.hibernate.model.Student</value>
			</list>
		</property>
		<property name="hibernateProperties">
		  <props>
		   <prop
		   key="hibernate.dialect">org.hibernate.dialect.MySQLDialect</prop>
		   <prop key="hibernate.current_session_context_class">thread</prop>
```

```
                    <prop key="hibernate.show_sql">false</prop>
                </props>
                </property>
        </bean>
        <bean id="studentDAO" class="com.demo.hibernate.StudentDAOImpl">
                <property name="sessionFactory"
ref="hibernate4AnnotatedSessionFactory" />
        </bean>
</beans>
```

Following is the code for SpringHibernateTest.java which will put into action the Hibernate implementation and perform the necessary database operations:

```
package com.demo;

import org.springframework.context.support.ClassPathXmlApplicationContext;

import com.demo.hibernate.StudentDAO;
import com.demo.hibernate.model.Student;

public class SpringHibernateTest {

        public static void main(String[] args) {
                ClassPathXmlApplicationContext context = new
                                ClassPathXmlApplicationContext("spring.xml");
      Student student = new Student();
      student.setId(652l);
      student.setName("Student 11");
      student.setRoll("ECE/21/2018-19");
      student.setMarks(902);
      StudentDAO studentDao = (StudentDAO)context.getBean("studentDAO");
      studentDao.save(student);
      Student stdResult = studentDao.getStudent (652l);
      System.out.println("Student enrolled successfully : " + stdResult);
      context.close();
   }
}
```

Through this code, two methods; getStudent() and save() of the StudentDao interface are invoked. The part of code uses Hibernate APIs and enables Spring to execute this code by Hibernate session's method.

Build and execute the application.

The processing of the Java code is as follows:

- When the application calls the method context.getBean("studentDAO"); Spring searches the Student DAO Bean by name and initializes the Hibernate session object, and it initializes Student DAO implementation object.
- When the Session Factory Bean object is constructed, the application resolves the data source information (database tables and the Java classes).

Summary

- Spring Framework has good support for data handling.
- A database transaction must have following four key properties, known as ACID:
  - Atomicity
  - Consistency
  - Isolation
  - Durability
- DAO is an object, which provides an abstract interface to a database. DAO pattern uses abstraction and encapsulation design principles.
- JDBC is a standard Java API for database-independent connectivity between Java and relational databases and supports both two-tier and three-tier architectures.
- All the Spring JDBC classes belong to any of the following four separate packages:
  - core
  - datasource
  - object
  - support
- The JdbcTemplate class provided by Spring Framework manages all the low-level details starting from opening the connection to closing the connection.
- MySQL is an open-source RDBMS that uses SQL. Spring Framework includes support for MySQL.
- ORM uses an object-oriented paradigm that enables to query and manipulate data from a database.
- Hibernate is a powerful, high performance ORM solution that does not need any application server support to operate.

Check Your Progress

1. Which of the following defines the key properties of a database transaction?

A.	Flexible	**B.**	Reliable
C.	ACID	**D.**	All of these

2. Which of the following are JdbcTemplate class methods?

A.	public int update	**B.**	public string execute()
C.	public void execute	**D.**	None of these

3. Batch processing means executing a _____ of queries and submitting them with one call to the database.

A.	Batch	**B.**	Database
C.	Statement	**D.**	Beans

4. Which of the following is a valid ORM option in Java?

A.	Castor	**B.**	TopLink
C.	SpringDAO	**D.**	All of these

Answers

1.	C
2.	A and C
3.	A
4.	D

Session 4

Spring Web MVC Framework

Objectives

Welcome to the session, Spring Web MVC Framework.

This session describes Model-View-Controller (MVC) architecture and MVC with Spring. It also discusses form validation and bootstrapping of Spring MVC applications.

In this Session, you will learn to:

- Explain MVC architecture
- Explain how to use MVC architecture in Spring applications
- Describe the steps to develop a Spring MVC application
- Describe Bootstrap in Spring MVC programs

4.1 MVC Architecture and 'Separation of Concern' Pattern

Model-View-Controller (MVC) is a software design pattern or technology framework to organize Web application code into interconnected units called model, view, and controller. This organizing is done on the basis of the functions each unit performs. This helps to separate internal representations of information from the ways it is presented to and accepted by the user.

Following are the components of the MVC pattern as shown in Figure 4.1:

- **Model**: It is the central component of the MVC design pattern. It can be defined as the business entity on which the overall application operates. This component directly manages the application's data, rules, and logic.

- **View**: It refers to the user interface that renders the model into a form of interaction. It is possible to display multiple views of the same information, such as a bar chart for senior management and a tabular view for accountants. These views are script-based templating systems, such as JSP, ASP, and PHP and easily integrated with AJAX technology.

- **Controller**: It refers to the software code that handles a request from a view and converts it to appropriate commands for the model or the view component. In other words, it controls the interactions between the model and the view components.

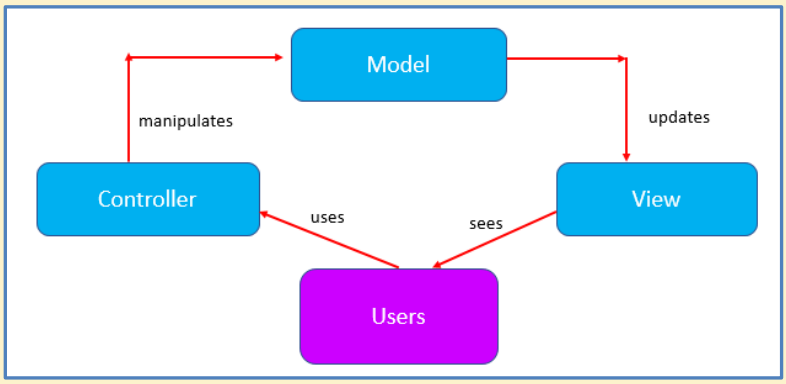


Figure 4.1: MVC Architecture

Consider an analogy to understand this better. Assume that you have invited a few guests for a simple dinner. You select a main course recipe from a cookbook.

You then purchase ingredients mentioned in the recipe and store them in the fridge. You'll use these materials when you are ready to cook the main course. Then, you arrange the tableware. Finally, you begin cooking.

Thus, before your guests arrive, you organize and separate each task neatly. Everything is set in place and there is no last-minute chaos. How does this relate to MVC?

The recipe tells you what materials are to be used, in what quantity, and how long they are to be cooked. Hence, the recipe acts like a **Controller**.

The fridge containing the raw materials which you'll use to cook is the **Model**.

The tableware, serving bowls, and plates are what the guests will see when they sit down to eat. The outcome of your efforts will be presented here.

They interact with the Model and the Controller. Hence, they act like a **View**.

The basic idea of MVC is that each part of your application code has a specific purpose.

MVC supports the Separation of Concerns (SoC) design principle. Separation of Concern (SoC) is a design principle as per which the software system or a program must be decomposed into distinct sections that overlap in functionality as little as possible and every section addressed as a separate concern. A concern is a set of information that affects the code of a computer program. SoC appears in various forms in the evolution of programming languages, methodologies, and best practices. A program that implements SoC is called modular program.

SoC provides following benefits:

- Allows people to work on individual pieces of the system in isolation
- Facilitates reusability
- Ensures the maintainability of a system
- Adds new features easily, which means that it ensures extensibility
- Enables users to better understand the system

MVC is also based on SoC principle by making business entities (Model), business logic (Controllers), and presentation logic (Views) independent of each other. The Views are just the presentation form of an application, they do not need to know specifically about the requests coming from the Controller. The Model is independent of Views and Controllers because it holds business entities. The Controller is independent of Views and Models; it is used to handle requests and send them as per the routes defined and as per the needs of the rendering Views.

4.2 Understanding Spring MVC

The Spring MVC framework provides ready components that can be used to develop flexible and loosely-coupled Web applications. In a Spring MVC application, the model component usually consists of domain objects (business entities). These domain objects are processed by the service layer and persisted by the persistence layer. The Views of a Spring MVC application are JSP templates written with Java Standard Tag Library (JSTL).

Spring MVC architecture allows developers to work with dynamic languages and implement effective software engineering principles. The central component of Spring MVC is a Spring controller. A controller is the only servlet that needs to be configured in a Java Web deployment descriptor web.xml file.

A Spring MVC controller is also known as front controller and generally referred to a single servlet titled **DispatcherServlet**. The front controller manages the entire request and response handling process and every Web request/response goes through it.

4.2.1 DispatcherServlet in Spring MVC

As mentioned earlier, the Spring MVC framework is designed around a DispatcherServlet, which handles all the HTTP requests and responses. DispatcherServlet inherits from the HttpServlet class and is a servlet. Its declaration can be found within the default web.xml of the Spring MVC application.

DispatcherServlet can perform the following tasks:

- Handle all incoming requests and route them through Spring.
- Use customizable logic to determine the controllers that handle incoming requests.
- Forward all responses to view handlers to determine the correct views to route responses to corresponding view.
- Expose all beans defined in Spring to controllers for dependency injection.

Figure 4.2 shows the request processing workflow of the Spring Web MVC DispatcherServlet. The workflow shows the following sequence of events:

- After receiving an HTTP request, DispatcherServlet refers the HandlerMapping to call the appropriate Controller.
- The Controller takes the request and calls the appropriate service methods. The service method sets the model data basis the pre-defined business logic and returns view name to the DispatcherServlet.

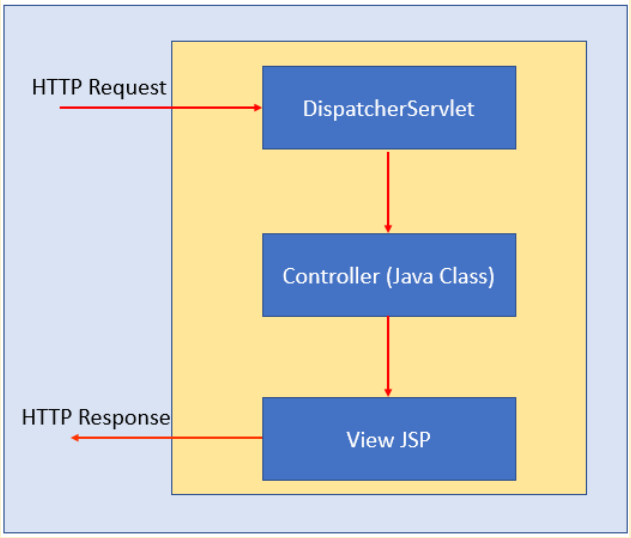


Figure 4.2: Request Processing Workflow

- The DispatcherServlet takes help from ViewResolver to pick-up the defined view for the request. Once view is finalized, the DispatcherServlet passes the model data to the view that is finally rendered on the browser.

To view the use of DispatcherServlet in programs, perform the following steps:

1) Launch STS.

2) Create a Spring MVC application by first clicking **File → New → Other**, scroll down and select **Spring Legacy Project**.

3) Click **Next**.

4) Then, scroll down and select **Spring MVC Project**. Refer to Figures 4.3 and 4.4.

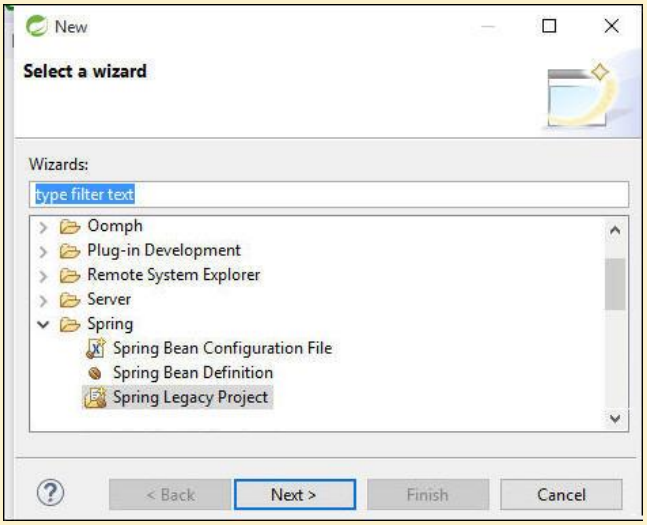


Figure 4.3: Selecting Wizard

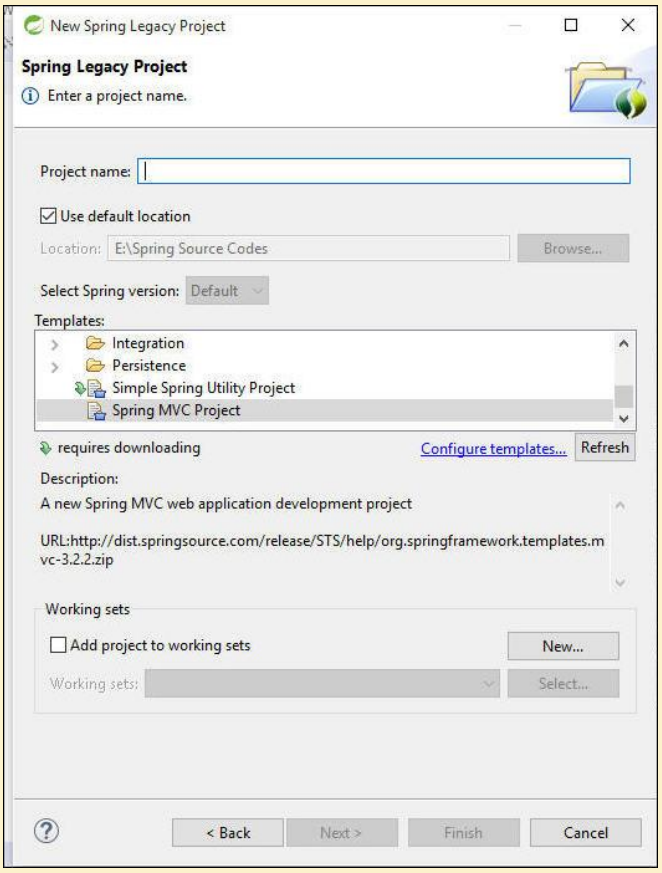


Figure 4.4: Selecting the Spring MVC Template

(Alternatively, you can choose the Simple Spring Web Maven template which will generate an application structure somewhat similar to an MVC application. In this case, you will have to create some of the files such as controllers yourself as Spring will not auto-generate them.)

5) Open the controller file present in src\main\java followed by the package path. For example, if the package is named com.session.four, then the controller file can be found under the path: src\main\java\com\session\four\HomeController.java

   Following is a snippet from the default auto-generated **HomeController.java**:

```
@RequestMapping(value = "/", method = RequestMethod.GET)
public String home(Locale locale, Model model) {
        Date date = new Date();
        DateFormat dateFormat =
                DateFormat.getDateTimeInstance(DateFormat.LONG,
                DateFormat.LONG, locale);
                String formattedDate = dateFormat.format(date);
                model.addAttribute("serverTime", formattedDate);
                return "home";
        }
```

6) Open the default view for the application, namely, **home.jsp**, which will be under the path /src/main/webapp/WEB-INF/views.

   Following code will be present under **home.jsp**:

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<%@ page session="false" %>
<html>
<head>
        <title>Home</title>
</head>
<body>
<h1>
        Hello world!
</h1>
<P>  The time on the server is ${serverTime}. </P>
</body>
</html>
```

The default **web.xml** file will map the DispatcherServlet appropriately as shown in the following code: (Observe the lines in bold, they are vital.)

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.5" xmlns="http://java.sun.com/xml/ns/javaee"
	xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
	xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd">

<!-- The definition of the Root Spring Container shared by all Servlets and Filters -->

	<context-param>
		<param-name>contextConfigLocation</param-name>
		<param-value>/WEB-INF/spring/root-context.xml</param-value>
	</context-param>
		<!-- Creates the Spring Container shared by all Servlets and Filters -->
	<listener>
		<listener-
class>org.springframework.web.context.ContextLoaderListener</listener-class>
	</listener>

	<!-- Processes application requests -->
	<servlet>
		<servlet-name> session-4 </servlet-name>
		<servlet-
class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
		<init-param>
			<param-name>contextConfigLocation</param-name>
			<param-value>/WEB-INF/spring/appServlet/servlet-
context.xml</param-value>
		</init-param>
		<load-on-startup>1</load-on-startup>
	</servlet>
	<servlet-mapping>
		<servlet-name>appServlet</servlet-name>
		<url-pattern>/</url-pattern>
	</servlet-mapping>
</web-app>
```

The **web.xml** file can be found in the /WEB-INF directory of the Web application.

Upon initialization of DispatcherServlet, the framework loads the application context from a file named servlet-context.xml located in the application's /WEB-INF directory.

The project structure is illustrated in Figure 4.5.

When this application is executed, DispatcherServlet knows it has to call the method **home** present in the controller.

DispatcherServlet knows that when a browser requests the page, it has to combine its results with the matching JSP file to make an HTML document.

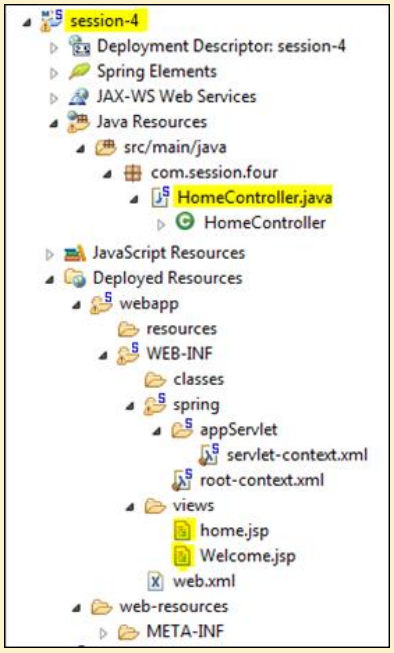


Figure 4.5: Project Structure

Include the following jar files in the WebContent/WEB-INF/lib folder by right-clicking the project name, selecting **Build Path → Configure Build Path**:

- commons-logging-x.y.z.jar
- org.springframework.asm-x.y.z.jar
- org.springframework.beans-x.y.z.jar
- org.springframework.context-x.y.z.jar
- org.springframework.core-x.y.z.jar
- org.springframework.expression-x.y.z.jar
- org.springframework.web.servlet-x.y.z.jar
- org.springframework.web-x.y.z.jar
- spring-web.jar

Upon building and executing the application on the server, the output will be as shown in Figure 4.6.

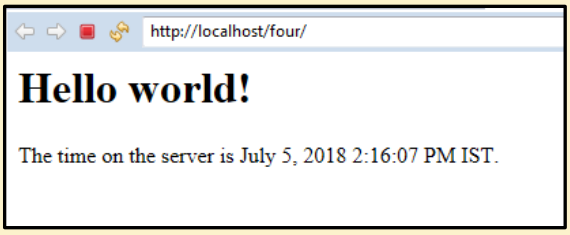


Figure 4.6: Output of the MVC Application

To change the default behavior of the DispatcherServlet and specify a different file, developers need to map requests that need to be handled by the DispatcherServlet. They can do this by using a URL mapping in the web.xml file.

Following code shows the declaration and mapping for DispatcherServlet in an application named session-4 (observe the lines given in bold):

```
<web-app id = "WebApp_ID" version = "2.4"  xmlns = "http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation = "http://java.sun.com/xml/ns/j2ee
  http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">

  <display-name>Spring MVC Application</display-name>

  <servlet>
    <servlet-name>session-4</servlet-name>
    <servlet-class>
      org.springframework.web.servlet.DispatcherServlet
    </servlet-class>
    <load-on-startup>1</load-on-startup>
  </servlet>
<!-- The servlet-mapping tag indicates what URLs will be handled by which
 DispatcherServlet.  -->
<servlet-mapping>
    <servlet-name>session-4</servlet-name>
    <url-pattern>*.jsp</url-pattern>
  </servlet-mapping>
</web-app>
```

If the default filename (servlet-context.xml) and default location (WebContent/WEB-INF) needs to be changed, developer can customize this by adding the servlet listener ContextLoaderListener in the web.xml file. Following code shows how to change the file name and location:

```
<web-app...>
  <!------- DispatcherServlet definition ----->
  ....
  <context-param>
   <param-name>contextConfigLocation</param-name>
   <param-value>/WEB-INF/spring/root-context.xml</param-value>
 </context-param>
 <listener>
   <listener-class>org.springframework.web.context.ContextLoaderListener</listener-
class>
 </listener>
</web-app>
```

The last step is to check the required configuration for servlet-context.xml file located in Web application's /WEB-INF directory.

Following code shows how to check the configuration:

```
<beans xmlns = "http://www.springframework.org/schema/beans"
  xmlns:context = "http://www.springframework.org/schema/context"
  xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation = "http://www.springframework.org/schema/beans
  http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
  http://www.springframework.org/schema/context
  http://www.springframework.org/schema/context/spring-context-3.0.xsd">
  <context:component-scan base-package = "com.TestSpring" />

 <bean class =
"org.springframework.web.servlet.view.InternalResourceViewResolver">
    <property name = "prefix" value = "/WEB-INF/jsp/" />
    <property name = "suffix" value = ".jsp" />
  </bean>
</beans>
```

In this code, the servlet-context.xml file creates the beans defined, which overrides the bean definitions defined with the same name in the global scope. The <context:component-scan...> tag activates the Spring MVC annotation scanning capability to make use of annotations such as @Controller and @RequestMapping. The InternalResourceViewResolver has a set of rules defined to resolve the view names.

4.2.2 Controllers in Spring MVC

The simplest way for creating a controller class is to use the **@Controller** annotation to handle one or multiple requests. The @Controller annotation defines the controller class as a Spring MVC controller. The DispatcherServlet delegates the request to the controllers, which execute the functionality specific to them. The **@RequestMapping** annotation is used to map a URL to either a specific handler method or with an entire class, as per the specification in the program.

Following code displays how to define a Spring controller:

```
@Controller
@RequestMapping("/welcome")
public class WelcomeController {
  @RequestMapping(method = RequestMethod.GET)
  public String printWelcome(ModelMap model) {
    model.addAttribute("message", "Welcome to Spring MVC Framework!");
    return "Welcome";
  }
}
```

Upon building and executing the application, the output of the controller method displayed by the corresponding view is shown in Figure 4.7.

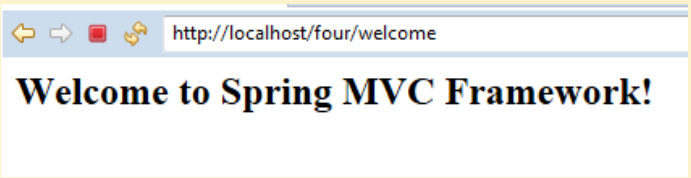


Figure 4.7: Controller Method Output

4.2.3 View and View Resolvers in Spring MVC

Spring MVC supports several types of views for different presentation technologies, such as JSPs, HTML, PDF, Excel worksheets, XML, Velocity templates, XSLT, JSON, Atom and RSS feeds, and JasperReports. The most common type of view is JSP templates written with JSTL.

Following code creates a Welcome view through /WEB-INF/welcome/welcome.jsp:

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<%@ page session="false" %>

<html>
  <head>
    <title>Welcome Spring MVC</title>
  </head>
    <body>
    <h2>${message}</h2>
  </body>
</html>
```

The ${message} attribute is defined inside the corresponding controller class as follows:
model.addAttribute("message", "Welcome to Spring MVC Framework");

Developers can include multiple attributes inside a specific view for display.

Output of this view is shown in Figure 4.8.

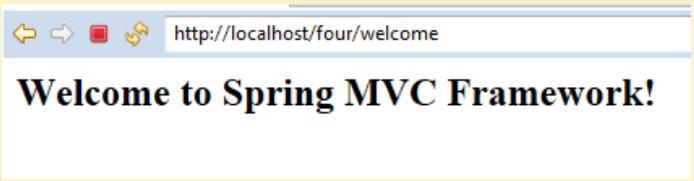


Figure 4.8: Output of Welcome View

The DispatcherServlet takes help from ViewResolver interface to pick up the defined view for the request. ViewResolver enables the user to render models in the browser without linking or associating the implementation with a specific view technology.

ViewResolver maps view names to actual views.

Spring Framework provides following ViewResolvers:

- **AbstractCachingViewResolver**: Is the ViewResolver that creates a cache copy of the views.
- **XmlViewResolver**: Is the ViewResolver that accepts a configuration file written in XML with the same DTD as Spring's XML bean factories. The default configuration file is /WEB-INF/views.xml.
- **ResourceBundleViewResolver**: Is the ViewResolver that uses bean definitions in a ResourceBundle, specified by the bundle base name. Generally, the bundle is defined in the properties file titled view.properties and is located in the classpath. The view.properties is the default name of the properties file.
- **UrlBasedViewResolver**: Is the ViewResolver that affects the direct resolution of logical view names to URLs, without an explicit mapping definition.
- **InternalResourceViewResolver**: Is the URL-based ViewResolver that supports InternalResourceView (Servlets and JSPs) and sub-classes, such as JstlView and TilesView.
- **VelocityViewResolver/FreeMarkerViewResolver**: Is the sub-class of UrlBasedViewResolver that supports VelocityView or FreeMarkerView.
- **ContentNegotiatingViewResolver**: Is the ViewResolver that resolves a view based on the request file name or Accept header.

Following two code snippets represent the use of InternalResourceViewResolver, the most common ViewResolver in the Web configuration. The first code snippet shows the Web configuration with the @EnableWebMvc, @Configuration, and @ComponentScan annotations. Since, the wizard creates configuration in XML, you need not create it explicitly for default configuration. However, the WebConfig class is required if complex configuration is needed.

```
@EnableWebMvc
@Configuration
@ComponentScan("org.baeldung.web")
public class WebConfig extends WebMvcConfigurerAdapter {
   // All Web configuration will go here
}
```

The second snippet shows the use of InternalResourceViewResolver:

```
@Bean
public ViewResolver internalResourceViewResolver() {
   InternalResourceViewResolver bean = new InternalResourceViewResolver();
   bean.setViewClass(JstlView.class);
   bean.setPrefix("/WEB-INF/view/");
   bean.setSuffix(".jsp");
   return bean;
}
```

This class is also not required for simple projects and corresponding configuration is created by the wizard.

4.2.4 Models in Spring MVC

In Spring MVC, the model is typically a JavaBean or a POJO that has getters and setters to manipulate its contents. It can be a simple or complicated object with dozens of sub-objects.

4.3 Developing a Simple Application in Spring MVC

To understand the Spring and Spring MVC concepts better, you can create an MVC application from scratch. For this, steps are as follows:

- Creating a Spring MVC application through Wizard
- Creating the .java file for controller
- Verifying Web.xml file
- Verifying DispatcherServlet
- Modifying the home.jsp File

The pom file includes dependency jar files. So, you need not add manually.

4.3.1 Creating a Spring MVC Application

Create a Spring MVC application through the wizard with a name 'student'. Make sure that Spring MVC Project template is chosen for the project since it requires Model too.

Create a package named 'com.learning.student'in the wizard as shown in Figure 4.9.

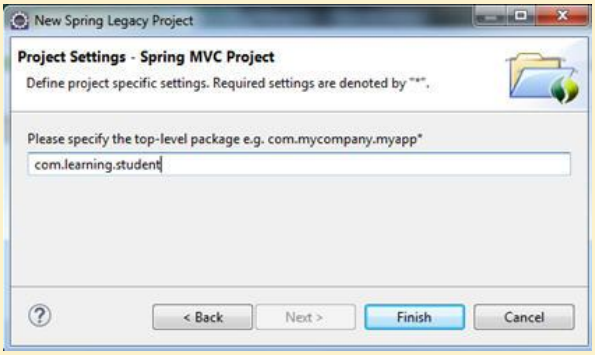


Figure 4.9: Specifying Base Package

4.3.2 Controller Code

Following code illustrates the auto-generated HomeController.java file:

```
package com.learning.student;
import java.util.Locale;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.RequestMapping;
```

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

/**
 * Handles requests for the application home page.
 */
@Controller
public class HomeController {

	private static final Logger logger =
LoggerFactory.getLogger(HomeController.class);

	/**
	 * Simply selects the home view to render by returning its name.
	 */
	@RequestMapping(value = "/", method = RequestMethod.GET)
	public String home(Locale locale, Model model) {
		logger.info("Welcome home! The client locale is {}.", locale);
		return "home";
	}
}
```

This is the code that will decide which view to display and what action is to be done. In this controller here, an instance of Logger is created and it logs a message, "Welcome home! The client locale is" followed by the actual locale name.

4.3.3 Web.xml File

The web.xml file will be auto-generated in the WEB-INF directory. You can open and view the configuration present in it.

Following section in the web.xml specifies the location of the root-context.xml file which defines context specific XML configuration.

```
<context-param>
   <param-name>contextConfigLocation</param-name>
   <param-value>/WEB-INF/spring/root-context.xml</param-value>
  </context-param>
```

Following section in web.xml contains the Servlet declaration:

```
<servlet>
   <servlet-name>session-4</servlet-name>
   <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
   <init-param>
     <param-name>contextConfigLocation</param-name>
     <param-value>/WEB-INF/spring/appServlet/servlet-context.xml</param-value>
   </init-param>
   <load-on-startup>1</load-on-startup>
  </servlet>
```

In this code, you are declaring a servlet that is named 'session-4' and is an instance of org.springframework.web.servlet.DispatcherServlet. You are specifying that it will be initialized with a parameter named contextConfigLocation which contains the path to the configuration XML. Here, the parameter value is the path to the file. The path is: /WEB-INF/spring/appServlet/servlet-context.xml

Further, the configuration file specifies load-on-startup which is an integer value that indicates order for multiple servlets to be loaded. If the application comprises more than one servlet to be loaded, you can define the order in which they will be initialized. 1 in this case indicates top most priority for this servlet.

Lastly, you declare a servlet-mapping and indicate the HTTP requests that will be handled by it:

```
<servlet-mapping>
   <servlet-name>session-4</servlet-name>
   <url-pattern>/</url-pattern>
  </servlet-mapping>
```

4.3.4 Defining DispatcherServlet

Following code illustrates the generated default servlet.xml in the WEB-INF directory:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans:beans xmlns="http://www.springframework.org/schema/mvc"
	xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
	xmlns:beans="http://www.springframework.org/schema/beans"
	xmlns:context="http://www.springframework.org/schema/context"
	xsi:schemaLocation="http://www.springframework.org/schema/mvc
http://www.springframework.org/schema/mvc/spring-mvc.xsd
		http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
		http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context.xsd">

	<!-- DispatcherServlet Context: defines this servlet's request-processing
infrastructure -->

	<!-- Enables the Spring MVC @Controller programming model -->
	<annotation-driven />

	<!-- Handles HTTP GET requests for /resources/** by efficiently serving up static
resources in the ${webappRoot}/resources directory -->
	<resources mapping="/resources/**" location="/resources/" />

	<!-- Resolves views selected for rendering by @Controllers to .jsp resources in
the /WEB-INF/views directory -->
	<beans:bean
class="org.springframework.web.servlet.view.InternalResourceViewResolver">
		<beans:property name="prefix" value="/WEB-INF/views/" />
		<beans:property name="suffix" value=".jsp" />
	</beans:bean>

	<context:component-scan base-package="com.learning.student" />
</beans:beans>
```

4.3.5 Creating the .jsp File

Following code illustrates how to modify auto-generated home.jsp file in the WEB-INF folder:

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<%@ page session="false" %>
<html>
<head>
        <title>Home</title>
</head>
<body>
<h1>
        Welcome, Student!
</h1>
</body>
</html>
```

4.3.6 Including jar Files for the Project

Spring STS or Maven both include various jar files needed for the project through the pom.xml file. The wizard generates the pom file and can be used as is, unless any specific additional jar is required. Alternatively, you can also add jar files using **Build Path** → **Configure Path** as described earlier.

Finally, all the steps for the application are completed.

Save and run the application to view the input.

A page is displayed as shown in Figure 4.10.

Figure 4.10: Output of Home View

4.4 Validating Forms

Spring MVC provides support for HTML form validation, which means it validates the data that is specified in the HTML form. The validation is also done by the code that is written.

Now extend the Student application in order to understand how to implement validation.

Perform the following steps:

- Create the .java Classes
- Configure Spring Configuration File

- Create the .jsp Files
- Run the Application

4.4.1 Create the .java Classes

In this application, a model class named **Student** will be created. This class defines the Student entity. Following code displays how to create the **Student.java** file under the **model** folder:

```
package com.learning.student.model;
public class Student {

	private Integer age;
	private String name;
	private Integer id;
	public Integer getAge() {
		return age;
	}
	public void setAge(Integer age) {
		this.age = age;
	}
	public String getName() {
		return name;
	}
	public void setName(String name) {
		this.name = name;
	}
	public Integer getId() {
		return id;
	}
	public void setId(Integer id) {
		this.id = id;
	}
}
```

The default **HomeController.java** can be edited to remove the date message as follows:

```
package com.learning.student;
import java.util.Locale;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
/**
```

```
 * Handles requests for the application home page.
 */
@Controller
public class HomeController {

	private static final Logger logger =
LoggerFactory.getLogger(HomeController.class);

	/**
	 * Simply selects the home view to render by returning its name.
	 */
	@RequestMapping(value = "/", method = RequestMethod.GET)
	public String home(Locale locale, Model model) {
		logger.info("Welcome home! The client locale is {}.", locale);
	  return "home";
	}
}
```

Following code displays how to create the **StudentController.java** file under the package:

```
package com.learning.student;

import org.springframework.stereotype.Controller;
import org.springframework.ui.ModelMap;
import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.servlet.ModelAndView;

import com.learning.student.model.Student;

@Controller
public class StudentController {

	@RequestMapping(value = "/registration", method = RequestMethod.GET)
	public ModelAndView student() {
		return new ModelAndView("student", "command", new Student());
	}

	@RequestMapping(value = "/addStudent", method = RequestMethod.POST)
	public String addStudent(@ModelAttribute("SpringWeb") Student student,
ModelMap model) {
		model.addAttribute("name", student.getName());
		model.addAttribute("age", student.getAge());
		model.addAttribute("id", student.getId());
		return "result";
```

```
        }
}
```

4.4.2 Configure Spring Configuration File

The Spring Project wizard automatically generates the web.xml file. You will not need to make any changes to it.

4.4.3 Create the .jsp Files

Following code displays how to create the **home.jsp** file under WEB-INF/views directory:

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<%@ page session="false" %>
<html>
<head>
        <title>Home</title>
</head>
<body>
<h1>
        Welcome, Student!
</h1>
 <h2> Click Here for  <a href="/student/registration">Self Registration</a> </h2>
</body>
</html>
```

Following code displays how to create the **student.jsp** file under WEB-INF/views directory:

```
<%@taglib uri = "http://www.springframework.org/tags/form" prefix = "form"%>
<html>
  <head>
    <title>Spring MVC Form Handling</title>
  </head>
  <body>
    <h2>Student Registration</h2>
    <form:form method = "POST" action = "/student/addStudent">
      <table>
        <tr>
          <td><form:label path = "name">Name:</form:label></td>
          <td><form:input path = "name" /></td>
        </tr>
        <tr>
          <td><form:label path = "age">Age:</form:label></td>
          <td><form:input path = "age" /></td>
        </tr>
        <tr>
```

```
            <td><form:label path = "id">ID:</form:label></td>
            <td><form:input path = "id" /></td>
         </tr>
         <tr>
            <td colspan = "2">
               <input type = "submit" value = "Submit"/>
            </td>
         </tr>
      </table>
   </form:form>
</body>
</html>
```

Following code displays how to create the **result.jsp** file under the same directory in which the student.jsp file is created:

```
<%@page contentType="text/html;charset = UTF-8" language="java"%>
<%@page isELIgnored="false"%>
<%@taglib uri="http://www.springframework.org/tags/form" prefix="form"%>
<html>
<head>
<title>Spring MVC Form Handling</title>
</head>
<body>
        <h2>Submitted Student Information</h2>
        <table>
                <tr>
                        <td>Name</td>
                        <td>${name}</td>
                </tr>
                <tr>
                        <td>Age</td>
                        <td>${age}</td>
                </tr>
                <tr>
                        <td>ID</td>
                        <td>${id}</td>
                </tr>
        </table>
</body>
</html>
```

4.4.4 Run the Application

Run the project. This will show the application in the IDE as illustrated in Figure 4.11.

Click the Self Registration link.

Figure 4.11: Output Showing Student View

The view shown in Figure 4.12 will be displayed.

Add some student details here.

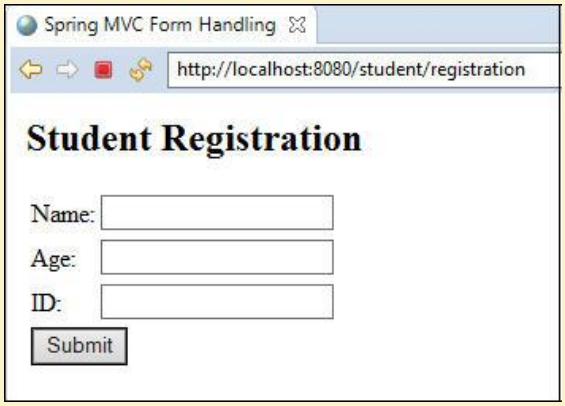


Figure 4.12: Student Registration Screen

After submitting the information by clicking Submit, the newly added student details are displayed, as shown in Figure 4.13.

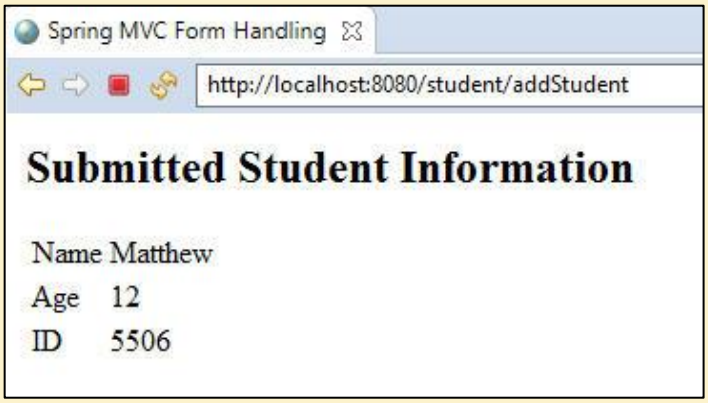


Figure 4.13: Student Details Submitted

Summary

- MVC is a software design pattern, which can be used to develop user interfaces of Web applications and divides an application into the interconnected components: Model, View, and Controller.
- Separation of Concern (SoC) is a design principle as per which the software system or a program is divided into distinct sections with minimum overlap in functionalities between these sections.
- A concern is a set of information that affects the code of a computer program.
- The Spring MVC framework is designed around a DispatcherServlet, which handles all the HTTP requests and responses.
- Spring MVC provides support for HTML form validation, which means it validates the data that user enters in a HTML form.

Check Your Progress

1. Which of the following defines MVC?

A.	A programming technique	B.	A software design pattern
C.	A software design document	D.	A software program

2. Which of the following is supported by Spring?

A.	SoC	B.	MVC
C.	Bootstrap	D.	All of these

3. SoC is a design principle as per which a _____ must be decomposed into distinct _____.

A.	Batch, groups	B.	Group, batches
C.	Program, sections	D.	Bean, sections

4. DispatcherServlet handles all the _____ requests and responses.

A.	HTTP	B.	HTML
C.	XML	D.	JSON

5. Which of the following is not the Spring-supported ViewResolver?

A.	AbstractCachingViewResolver	B.	UrlBasedViewResolver
C.	InternalResourceViewResolver	D.	BootstrapResolver

Answers

1.	B
2.	D
3.	C
4.	A
5.	D

Session 5

Spring Security

Objectives

Welcome to the session, Spring Security.

This session introduces Spring Security framework and describes the use of various Spring Security components in Web applications and Java configuration setup. It also describes the Spring-supported validation of beans.

In this session, students will learn to:

- Explain the Spring Security framework and Java Configuration
- Explain how to use Spring Security concepts in a Spring Web application
- Describe how to use validation in Spring-supported Web applications

5.1 Spring Security

Security is an important aspect of enterprise applications. Developers need to ensure that applications are designed to be secure. Spring Security is a powerful and highly customizable Java/Java EE framework that focuses on providing both authentication and authorization to secure Spring-based enterprise applications.

Enterprise applications are not easily portable; therefore, when developers switch server environments, a lot needs to be done to reconfigure application's security in the new target environment. Spring Security overcomes these problems and brings forth several other useful and customizable security features.

5.1.1 Major Operations in Spring Security

Spring Security targets two major areas of application security, namely, **authentication** and **authorization**.

Authentication is the process of validating a user, device, or some other system for who they claim to be. Consider a real-world example of authentication. Employees in an organization are provided swipe cards with unique IDs, using which they can authenticate themselves as they enter/exit the office premises.

Authorization refers to the process of deciding whether a user, device, or some other system is permitted to perform an action within the application. For example, a University's students may not be authorized to view Web pages dedicated to professors and administration. Professors may be allowed to view some Web pages more than what students can, but not all Web pages. An administrator would be authorized to view and edit any Web page of the University. Thus, each set of users have different permissions, based on authorization.

Figure 5.1: Secure Login

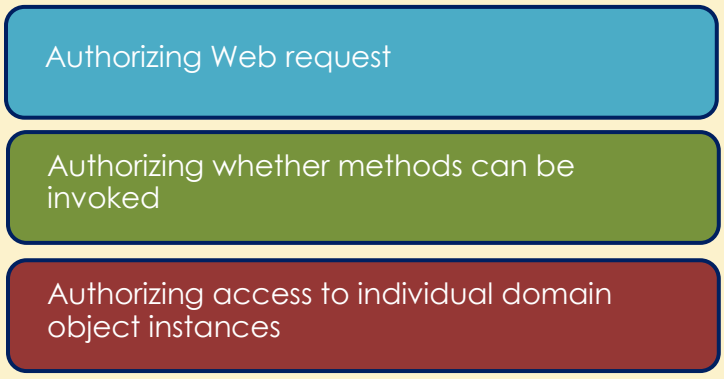
Figure 5.1 demonstrates an example of a secure login screen of a Web application.

Authentication operation is used by a server when it needs to know who is accessing the information. Authentication is done by using a user name and password, but can also be performed through cards, voice recognition, and fingerprints. Authentication does not determine the tasks that a user can do or the files that the user can access.

On the contrary, authorization is coupled with authentication to let server know who the user is, when the user tries to access an application. Authorization is used to control viewer access to certain pages of an application.

To understand the difference between authentication and authorization in Spring, consider a Spring application that manages the details of the marketing team. When a user logs on to the application and enters the credentials, Spring authenticates the user using various authentication models, such as HTTP basic authentication and form-based authentication. It is also possible that different users have different access level, such as manager has all the access and team member has access to only some parts of the application.

Spring determines authorization levels of users through three primary areas:



Spring Security supports a wide range of authentication models and authorization capabilities. Most of the authentication models are either provided by third parties or are developed by standards bodies, such as the Internet Engineering Task Force. Spring Security also provides its own set of authentication features. Spring Security currently supports authentication integration with various technologies, such as HTTP BASIC authentication headers, Lightweight Directory Access Protocol (LDAP), Form-based authentication, OpenID authentication, and so on.

Independent software vendors adopt Spring Security because of this significant choice of flexible authentication models. With Spring Security, vendors can quickly integrate their solutions with whatever their clients need without doing a lot of engineering or requesting the client to change their environment.

The Spring Security authorization capabilities include authorizing Web requests (servlet specification Web pattern security), methods that can be invoked (EJB container managed security), and access to individual domain object instances (file system security).

5.2 Spring Security Configuration

Spring Security framework provides pre-defined classes, methods, and interfaces that help developers to design their Web applications.

5.2.1 Spring Security Setup

Spring Security framework includes the following configuration components:

- AbstractSecurityWebApplicationInitializer
- @EnableWebSecurity
- Configurations imported by WebSecurityConfiguration
- FilterChainProxy
- Web Security and HTTP Security
- WebSecurityConfigurerAdapter
- SpringWebMvcImportSelector
- @EnableGlobalAuthentication

5.2.2 Namespace Configuration

A namespace is a separate program region that developers can use to restrict the scope of variables, functions, classes, and so on within a Java program. Using a namespace provides a way to implement hidden information and helps to avoid conflict with the existing definitions. In other words, a namespace uniquely identifies a set of names to prevent ambiguity when developers use the objects having different origins but the same names together.

A Spring Configuration file may contain one or more of the namespaces as listed in Table 5.1. Each namespace is described and its usage is demonstrated through a code snippet.

Make sure that there is one 'beans' tag and subsequent code snippets in the table that can be used by copying schema location(s) only.

Namespace	Description	Code Snippet
aop	Provides the elements that can be used to declare Spring aspects and use them as proxies.	<?xml version="1.0" encoding="UTF-8"?> <beans xmlns="http://www.springframework.org/schema/beans" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:aop="http://www.springframework.org/schema/aop" xsi:schemaLocation=" http://www.springframework.org/schema/beans

Namespace	Description	Code Snippet
		http://www.springframework.org/schema/beans/spring-beans-2.0.xsd http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop-2.0.xsd"> <!-- <bean/> definitions here --> </beans>
beans	Allows developers to declare and configure beans and establish a connection among them.	<?xml version="1.0" encoding="UTF-8"?> <beans xmlns="http://www.springframework.org/schema/beans" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation=" http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-2.0.xsd"> </beans>
context	Provides elements to support dependency injection in a program. Spring containers use them to auto-detect and auto-wire beans.	<beans xmlns="http://www.springframework.org/schema/beans" xmlns:context="http://www.springframework.org/schema/context" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation=" http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context-2.5.xsd"> </beans>
Mvc	Provides MVC framework support.	<?xml version="1.0" encoding="UTF-8"?> <beans xmlns="http://www.springframework.org/schema/beans"

Namespace	Description	Code Snippet
		xmlns:mvc="http://www.springframework.org/schema/mvc" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation=" http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-3.1.xsd http://www.springframework.org/schema/mvc http://www.springframework.org/schema/mvc/spring-mvc-3.1.xsd"> <mvc:annotation-driven/> <beans>

Table 5.1: Namespace Configuration

5.3 Java Configuration

Spring Security Java Configuration support enables developers to easily configure Spring Security (authentication and authorization) without the use of any XML.

The basic Spring Security Java Configuration adds the following features to a Web application:

- Authenticates all the URLs used in the application
- Generates a login form and authenticates the user
- Allows the user to logout
- Provides session fixation protection
- Integrates with the following Servlet API methods
    - HttpServletRequest#getRemoteUser()
    - HttpServletRequest.html#getUserPrincipal()
    - HttpServletRequest.html#isUserInRole(java.lang.String)
    - HttpServletRequest.html#login(java.lang.String, java.lang.String)
    - HttpServletRequest.html#logout()

Make sure that maven file (pom.xml) contains code as follows:

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>org.springframework.samples.service.service</groupId>
  <artifactId>sesion-5-1</artifactId>
```

```
  <version>0.0.1-SNAPSHOT</version>
  <packaging>war</packaging>
    <properties>
                <!-- Generic properties -->
                <java.version>1.8</java.version>
                <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
                <!-- Web -->
                <jsp.version>2.2</jsp.version>
                <jstl.version>1.2</jstl.version>
                <servlet.version>2.5</servlet.version>
                <!-- Spring -->
                <spring-framework.version>4.0.0.RELEASE</spring-framework.version>
                <springsecurity.version>5.0.5.RELEASE</springsecurity.version>
                <!-- Hibernate / JPA -->
                <hibernate.version>4.2.1.Final</hibernate.version>
                <!-- Logging -->
                <logback.version>1.0.13</logback.version>
                <slf4j.version>1.7.5</slf4j.version>
                <!-- Test -->
                <junit.version>4.11</junit.version>
        </properties>
        <dependencies>
                <dependency>
                        <groupId>org.springframework</groupId>
                        <artifactId>spring-webmvc</artifactId>
                        <version>${spring-framework.version}</version>
                </dependency>
        <!-- Spring Security -->
                <dependency>
                        <groupId>org.springframework.security</groupId>
                        <artifactId>spring-security-core</artifactId>
                        <version>${springsecurity.version}</version>
                </dependency>
                <dependency>
                        <groupId>org.springframework.security</groupId>
                        <artifactId>spring-security-web</artifactId>
                        <version>${springsecurity.version}</version>
                </dependency>
                <dependency>
                        <groupId>org.springframework.security</groupId>
                        <artifactId>spring-security-config</artifactId>
                        <version>${springsecurity.version}</version>
                </dependency>
        </dependencies>
</project>
```

Java Configuration allows developers to implement the following concepts in their Web application:

- Authorize Requests
- Authentication
- HTTP Security
- Handling Logouts
- Webflux security
- OAuth 2.0 Login
- Multiple HTTP Security
- Method Security

5.3.1 Authorize Requests

The HTTP.authorizeRequests() method helps developers to secure URLs in their Web application.
Using this method ensures that the user requesting access to their application is authenticated. Spring Security's default configuration does not explicitly set a URL for the login page rather it generates one automatically. Even though, login page is generated automatically, a few applications provide their own login page.

For this, developers can use the following code snippet in the application's WebSecurityConfigurerAdapter:

```
package com.learning;
import java.util.ArrayList;
import java.util.List;
import java.util.Properties;
import org.springframework.context.annotation.Bean;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import
org.springframework.security.config.annotation.web.configuration.WebSecurityConf
igurerAdapter;
import org.springframework.security.core.GrantedAuthority;
import org.springframework.security.core.userdetails.User;
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.security.provisioning.InMemoryUserDetailsManager;

public class WebSecurityConfig extends WebSecurityConfigurerAdapter {
	@Bean
	public UserDetailsService userDetailsService() {
		final Properties users = new Properties();
		users.put("user", "pass,ROLE_USER,enabled");
		InMemoryUserDetailsManager manager = new
InMemoryUserDetailsManager(users);
		List<? extends GrantedAuthority> roles = new
ArrayList<GrantedAuthority>();
		manager.createUser(new User("user", "password", roles));
```

```
                return manager;
        }
        @Bean
        protected void configure(HttpSecurity http) throws Exception {
                http.authorizeRequests().anyRequest()
                                                .authenticated().and()
        .formLogin().loginPage("/login").permitAll();
        }
}
```

In this code, the configuration specifies the location of the login page. The formLogin().permitAll() method grants access to all the users for all the URLs. This code authenticates user for every URL in the application. However, developers can specify custom requirements for the URLs by adding multiple children to the http.authorizeRequests() method.

Following code snippet shows the role-based access to URLs:

```
protected void configure(HttpSecurity http) throws Exception {
        http.authorizeRequests()
                        .antMatchers("/resources/**", "/signup", "/about").permitAll()
                        .antMatchers("/admin/**").hasRole("ADMIN")
                        .antMatchers("/db/**").access("hasRole('ADMIN') and
hasRole('DBA')")
                        .anyRequest().authenticated()
                .and()
                // ...
                .formLogin();
}
```

As per this code, any URL that starts with 'resources', 'signup', or 'about' is accessible to all users, whereas, URLs starting with 'Admin' are accessible only to the user with admin role. Similarly, URLs starting with 'db' are accessible only to the user with admin and DBA roles.

5.3.2 Authentication

As stated earlier, Spring supports various authentication models. Developers can choose any model and configure that via the <authentication-manager></authentication-manager> tag and the AuthenticationProvider interface in the Web application.

Following code snippet represents how to add two users with password as 'password' and roles as ROLE_USER and ROLE_ADMIN to be authenticated:

```
<authentication-manager>
    <authentication-provider>
     <user-service>
```

```
        <user name="user1" password="password" authorities="ROLE_USER,
ROLE_ADMIN" />
        <user name="user2" password="password" authorities="ROLE_USER" />
        <user name="dba" password="root123" authorities="ROLE_ADMIN,ROLE_DBA" />
        </user-service>
      </authentication-provider>
    </authentication-manager>
```

Consider that a Spring application is created to demonstrate security, with a home page that includes a secure link as well as a free access link, as shown in Figure 5.2.

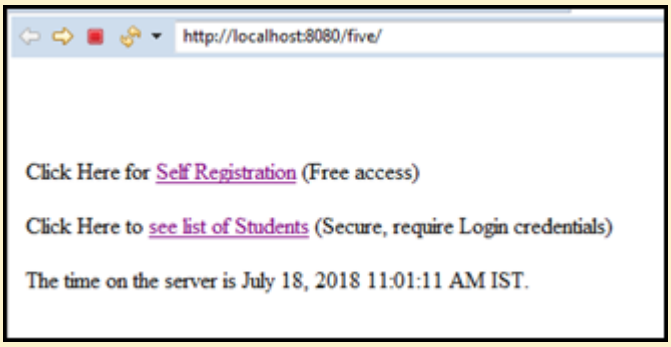


Figure 5.2: Application Demonstrating Spring Security

When the user clicks the second link that requires secure access, a login page appears as shown in Figure 5.3.

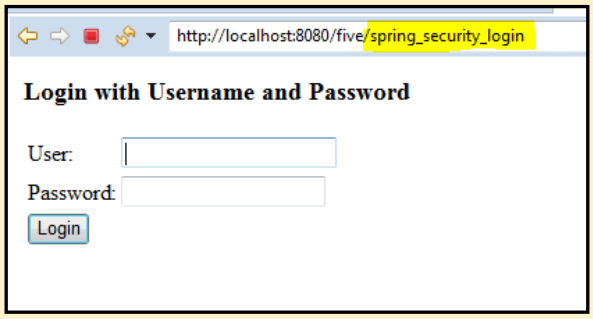


Figure 5.3: Login Page

Now, only the user with correct credentials and right access level will be able to access the next page. This information is mentioned in the Spring configuration file, as shown by the following code snippet:

```
<beans:beans xmlns="http://www.springframework.org/schema/security"
        xmlns:beans="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-4.0.xsd
        http://www.springframework.org/schema/security
        http://www.springframework.org/schema/security/spring-security-3.2.xsd">
        <http auto-config="true">
                <intercept-url pattern="/studentList**" access="ROLE_USER" />
        </http>
        <authentication-manager>
```

```
        <authentication-provider>
            <user-service>
            <user name="student" password="student" authorities="ROLE_USER" />
                </user-service>
            </authentication-provider>
        </authentication-manager>
</beans:beans>
```

Enter 'student' in both the fields **User:** and **Password:** text boxes as shown in Figure 5.4, to access the secured page.

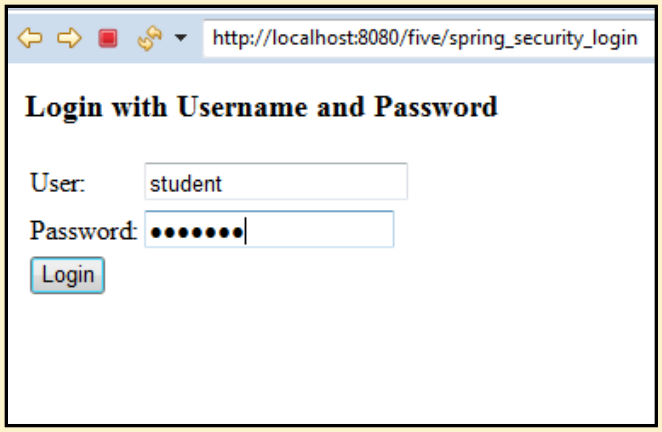


Figure 5.4: Login Credentials for Secure Access

After sucessfully authentication, the student details are displayed, as shown in Figure 5.5.

This result shows the data on the results.jsp page. The data is entered by the student while registering from the home page.

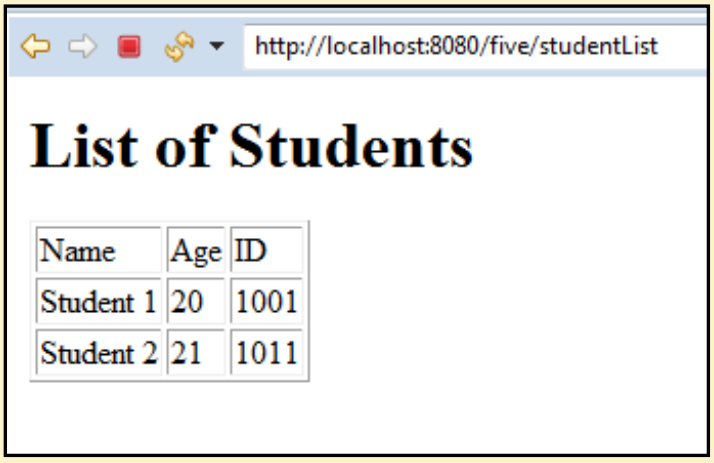


Figure 5.5: Student Details Displayed With Secure Access

In case the authentication fails while entering details on the Login page, an error message appears, as shown in Figure 5.6.

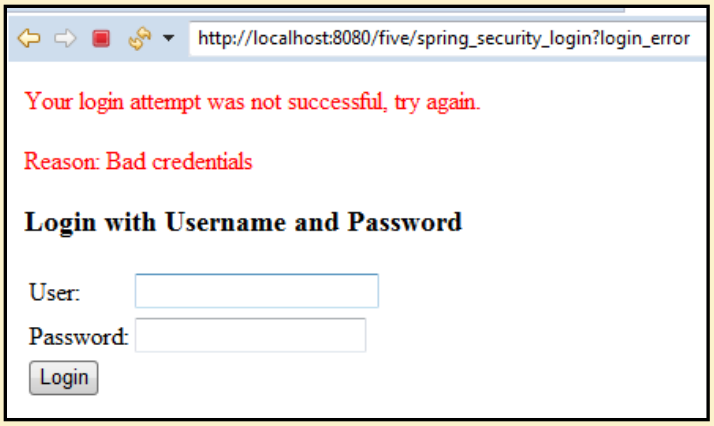


Figure 5.6: Message Displayed on Unsuccessful Authentication

In this example, users were authenticated based on hard-coded values.

Developers can also modify the code to authenticate users based on the data in the database tables. They just need to create the data source and add it to the code.

Following code snippet displays the use of a data source:

```
<authentication-manager>
<authentication-provider>
<jdbc-user-service data-source-ref="dataSource" />
</authentication-provider>
</authentication-manager>
```

Logging into Web Application

Developers can add the following code to configure the login page for their Web application:

```
<form-login login-page='/login.jsp' default-target-url='/home.jsp' always-use-default-target='true' />
```

In this code, the login.jsp is the login page and home.jsp is the page where user would be redirected after a successful login process. The default-target-url tag helps developers to set whether user gets redirected to the default URL every time or not. If true, the user is redirected to home.jsp even if user was accessing any other page at the end of previous session. If false, user is redirected to the same page that the user was accessing at the end of previous session.

Users' Authentication

The Authentication interface represents the token for an authentication request. This interface extends the Principal and Serializable super interfaces and implements several classes.

Some of the classes that the Authentication interface implements are as follows:

When a user is authenticated, the authentication mechanism stores the details in a thread-local SecurityContext managed by the SecurityContextHolder. Instead of using one of Spring Security's authentication mechanisms, developers can create their own authentication mechanism by creating an Authentication instance and using the code.

Following snippet code shows how to create a custom authentication mechanism:

```
SecurityContextHolder.getContext().setAuthentication(anAuthentication);
```

This code snippet sets any authentication mechanism (say anAuthentication) in Security Context Holder (SecurityContextHolder) through the setAuthentication method. This is just for demonstration purpose and not part of working code.

Developers can also define custom authentication by exposing AuthenticationProvider and UserDetailsService as beans. Following code snippet shows how to define AuthenticationProvider as a custom bean:

```
@Bean
public AuthenticationProvider springAuthenticationProvider() {
        return new SpringAuthenticationProvider();
}
```

Following code snippet shows how to define UserDetailsService as a custom bean:

```
@Bean
public UserDetailsService springDataUserDetailsService() {
        return new SpringDataUserDetailsService();
}
```

After adding these two snippets, WebSecurityConfig file includes the following code:

```
package com.learning;
import java.util.ArrayList;
import java.util.List;
import java.util.Properties;
import org.springframework.context.annotation.Bean;
import org.springframework.security.authentication.AuthenticationProvider;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import
org.springframework.security.config.annotation.web.configuration.WebSecurityConf
igurerAdapter;
import org.springframework.security.core.GrantedAuthority;
import org.springframework.security.core.userdetails.User;
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.security.provisioning.InMemoryUserDetailsManager;

public class WebSecurityConfig extends WebSecurityConfigurerAdapter {
        @Bean
        public UserDetailsService userDetailsService() {
                final Properties users = new Properties();
                users.put("user", "pass,ROLE_USER,enabled");
                InMemoryUserDetailsManager manager = new
InMemoryUserDetailsManager(users);
                List<? extends GrantedAuthority> roles = new
ArrayList<GrantedAuthority>();
                manager.createUser(new User("user", "password", roles));
                return manager;
        }
        @Override
        protected void configure(HttpSecurity http) throws Exception {
                http.authorizeRequests().anyRequest()
                                                        .authenticated().and()
        .formLogin().loginPage("/login").permitAll();
        }
        @Bean
        public AuthenticationProvider springAuthenticationProvider() {
                return new SpringAuthenticationProvider();
        }
```

```
        @Bean
        public UserDetailsService springDataUserDetailsService() {
                return new SpringDataUserDetailsService();
        }
}
```

SpringAuthenticationProvider includes the following code:

```
package com.learning;
import org.springframework.security.authentication.AuthenticationProvider;
import org.springframework.security.core.Authentication;
import org.springframework.security.core.AuthenticationException;

public class SpringAuthenticationProvider implements AuthenticationProvider {
        @Override
        public Authentication authenticate(Authentication authentication) throws
AuthenticationException {
                // perform business logic
                return null;
        }
        @Override
        public boolean supports(Class<?> arg0) {
                //perform business logic
                return false;
        }
}
```

Following is the code for SpringDataUserDetailsService:

```
package com.learning;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.security.core.userdetails.UsernameNotFoundException;
public class SpringDataUserDetailsService implements UserDetailsService {
        @Override
        public UserDetails loadUserByUsername(String argument) throws
UsernameNotFoundException {
                //perform business logic
                return null;
        }
}
```

5.3.3 HTTP Security

HTTP Security is a default security configuration provides by Spring Security framework. Through this configuration, Spring Security is aware that all users are to be authenticated with HTTP Basic authentication and Form-based authentication needs to

be supported. Following code snippet represents the default HTTP Security configuration:

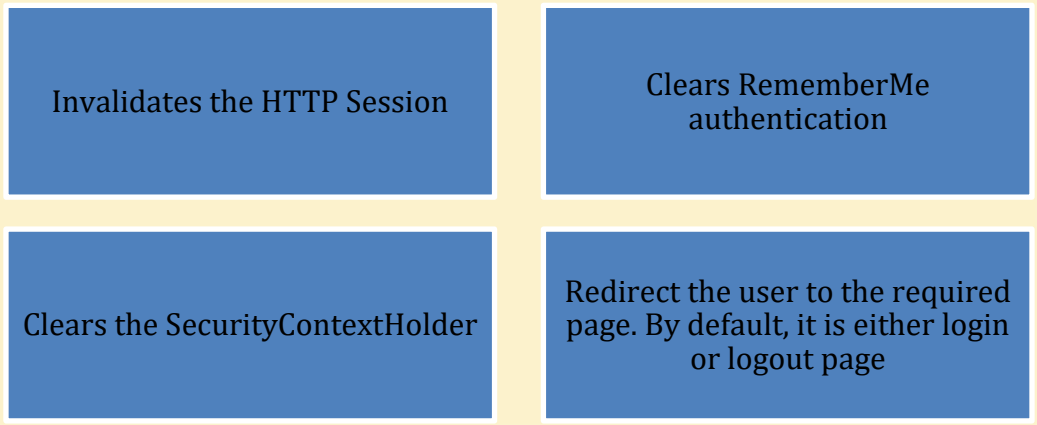
```
protected void configure(HttpSecurity http) throws Exception {
        http
                .authorizeRequests()
                        .anyRequest().authenticated()
                        .and()
                .formLogin()
                        .and()
                .httpBasic();
}
```

In this code, the and() method allows developers to continue configuring the parent class.

5.3.4 Handling Logouts

If developers want to apply default logout capabilities, they are required to use the WebSecurityConfigurerAdapter interface.

Some of the tasks that the WebSecurityConfigurerAdapter interface performs are as follows:



Like the login process, developers can customize their logout requirements, such as adding URL for redirecting user after logging out and clearing cookies after logout. Following code represents how to customize the logout mechanism:

```
protected void configure(HttpSecurity http) throws Exception {
        http
                .logout()
                        .logoutUrl("/my/logout")
                        .logoutSuccessUrl("/my/index")
                        .logoutSuccessHandler(logoutSuccessHandler)
                        .invalidateHttpSession(true)
                        .addLogoutHandler(logoutHandler)
```

```
                .deleteCookies(cookieNamesToClear)
                .and()
            ...
}
```

5.3.5 Webflux Security

Spring framework provides the Webflux module, which contains support for reactive HTTP and WebSocket clients as well as for reactive server Web applications, such as REST, HTML browser, and WebSocket style interactions.

Webflux is used to create filters. Developers can use the default Webflux configurations using WebFilter. However, developers can create following two types of custom configurations:

- **Minimal Webflux Security Configuration**: This configuration provides form and HTTP basic authentication and sets up the following:
  - Authentication and authorization
  - Default login page and a default log out page
  - Security-related HTTP headers and CSRF protection

  Following code snippet shows how to create this configuration:

```
@EnableWebFluxSecurity
public class HelloWebfluxSecurityConfig {
    @Bean
    public MapReactiveUserDetailsService userDetailsService() {
        UserDetails user = User.withDefaultPasswordEncoder()
            .username("user")
            .password("user")
            .roles("USER")
            .build();
        return new MapReactiveUserDetailsService(user);
    }
}
```

- **Explicit Webflux Security Configuration**: This configuration explicitly sets up all the things same as minimal Webflux security configuration.

  Following code snippet shows how to create this configuration:

```
@EnableWebFluxSecurity
public class HelloWebfluxSecurityConfig {
    @Bean
    public MapReactiveUserDetailsService userDetailsService() {
        UserDetails user = User.withDefaultPasswordEncoder()
            .username("user")
            .password("user")
```

```
                .roles("USER")
                .build();
        return new MapReactiveUserDetailsService(user);
    }
    @Bean
    public SecurityWebFilterChain springSecurityFilterChain(ServerHttpSecurity
http) {
        http
            .authorizeExchange()
                .anyExchange().authenticated()
                .and()
            .httpBasic().and()
            .formLogin();
        return http.build();
    }
}
```

5.3.6 OAuth 2.0 Login

OAuth 2.0 is the industry-standard protocol that provides specific authorization flows for Web and desktop applications, mobile phones, and other living room devices, such as a LCDs and laptops. This protocol is used for authorization.

The OAuth 2.0 Login feature of Spring framework provides the capability to login to an application by using existing account at an OAuth 2.0 pre-defined Provider, such as GitHub or OpenID Connect 1.0 Provider, such as Google.

Make sure that pom.xml file contains OAuth Client references as shown in the following code snippet:

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>org.springframework.samples.service.service</groupId>
  <artifactId>sesion-5-1</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <packaging>war</packaging>

  <properties>

    <!-- Generic properties -->
    <java.version>1.8</java.version>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
```

```
  <!-- Web -->
  <jsp.version>2.2</jsp.version>
  <jstl.version>1.2</jstl.version>
  <servlet.version>2.5</servlet.version>
  <!-- Spring -->
  <spring-framework.version>5.0.7.RELEASE</spring-framework.version>
  <springsecurity.version>5.0.5.RELEASE</springsecurity.version>

  <!-- Hibernate / JPA -->
  <hibernate.version>4.2.1.Final</hibernate.version>

  <!-- Logging -->
  <logback.version>1.0.13</logback.version>
  <slf4j.version>1.7.5</slf4j.version>

  <!-- Test -->
  <junit.version>4.11</junit.version>

 </properties>

 <dependencies>

  <!-- Spring MVC -->
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-webmvc</artifactId>
    <version>${spring-framework.version}</version>
  </dependency>
  <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-beans</artifactId>
      <version>${spring-framework.version}</version>
  </dependency>
   <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-context</artifactId>
      <version>${spring-framework.version}</version>
  </dependency>
  <!-- Spring Security -->
  <dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-core</artifactId>
    <version>${springsecurity.version}</version>
  </dependency>
  <dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-web</artifactId>
```

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

   <!-- Spring OAuth-2 Security -->
   <dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-oauth2-client</artifactId>
    <version>5.0.0.RELEASE</version>
   </dependency>
   <dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-oauth2-jose</artifactId>
    <version>5.0.0.RELEASE</version>
   </dependency>
    </dependencies>
</project>
```

Following code snippet represents how to configure Google in CommonOAuth2Provider:

```
package com.learning;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import
org.springframework.security.config.annotation.web.configuration.EnableWebSecurit
y;
import
org.springframework.security.config.annotation.web.configuration.WebSecurityConfi
gurerAdapter;
import org.springframework.security.config.oauth2.client.CommonOAuth2Provider;
import
org.springframework.security.oauth2.client.InMemoryOAuth2AuthorizedClientService;
import org.springframework.security.oauth2.client.OAuth2AuthorizedClientService;
import org.springframework.security.oauth2.client.registration.ClientRegistration;
import
org.springframework.security.oauth2.client.registration.ClientRegistrationRepository;
import
org.springframework.security.oauth2.client.registration.InMemoryClientRegistrationRe
pository;
```

```
@Configuration
public class OAuth2LoginConfig {
	@EnableWebSecurity
	public static class OAuth2LoginSecurityConfig extends
WebSecurityConfigurerAdapter {
		@Override
		protected void configure(HttpSecurity http) throws Exception {

	http.authorizeRequests().anyRequest().authenticated().and().oauth2Login();
		}
		@Bean
		public ClientRegistrationRepository clientRegistrationRepository() {
			return new
InMemoryClientRegistrationRepository(this.googleClientRegistration());
		}
		@Bean
		public OAuth2AuthorizedClientService authorizedClientService() {
			return new
InMemoryOAuth2AuthorizedClientService(this.clientRegistrationRepository());
		}
		private ClientRegistration googleClientRegistration() {
			return
CommonOAuth2Provider.GOOGLE.getBuilder("google").clientId("google-client-
id").clientSecret("google-client-secret").build();
		}
	}
}
```

5.3.7 Multiple HTTP Security

Spring Security framework allows to configure multiple HTTP Security instances by extending the WebSecurityConfigurationAdapter multiple times.

Following code snippet shows how to have different configurations for URLs that start with /test/ (http.antMatcher):

```
package com.learning;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import
org.springframework.security.config.annotation.web.configuration.EnableWebSe
curity;
import
org.springframework.security.config.annotation.web.configuration.WebSecurityC
onfigurerAdapter;
```

```
import org.springframework.security.core.userdetails.User;
import org.springframework.security.core.userdetails.User.UserBuilder;
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.security.provisioning.InMemoryUserDetailsManager;

@EnableWebSecurity
public class MultiHttpSecurityConfig {

	@Bean
	public UserDetailsService userDetailsService() throws Exception {
		// ensure the passwords are encoded properly
		UserBuilder users = User.withDefaultPasswordEncoder();
		InMemoryUserDetailsManager manager = new
InMemoryUserDetailsManager();

	manager.createUser(users.username("user").password("password").roles("US
ER").build());

	manager.createUser(users.username("admin").password("password").roles("
USER", "ADMIN").build());
		return manager;
	}

	@Configuration
	public static class ApiWebSecurityConfigurationAdapter extends
WebSecurityConfigurerAdapter {
		protected void configure(HttpSecurity http) throws Exception {

	http.antMatcher("/test/**").authorizeRequests().anyRequest().hasRole("ADM
IN").and().httpBasic();
		}
	}

	@Configuration
	public static class FormLoginWebSecurityConfigurerAdapter extends
WebSecurityConfigurerAdapter {
		@Override
		protected void configure(HttpSecurity http) throws Exception {

	http.authorizeRequests().anyRequest().authenticated().and().formLogin();
		}
	}
}
```

In this code, after configuring authentication, an instance of WebSecurityConfigurerAdapter is created that contains @Order to specify which WebSecurityConfigurerAdapter is to be considered first. Another instance of

WebSecurityConfigurerAdapter is created. This configuration will work if the URL does not start with /test/.

5.3.8 Method Security

Spring Security provides ways to add authorization semantics to service layer methods. It allows to apply security to a single bean using the intercept-methods element to decorate the bean declaration or apply security to multiple beans across the entire service layer using the AspectJ style pointcuts.

Following code snippet shows how to use a custom MethodSecurityExpressionHandler:

```
@EnableGlobalMethodSecurity(prePostEnabled = true)
public class MethodSecurityConfig extends GlobalMethodSecurityConfiguration {
        @Override
        protected MethodSecurityExpressionHandler createExpressionHandler() {
                // ... create and return custom MethodSecurityExpressionHandler ...
                return expressionHandler;
        }
}
```

Developers can use the @Secured annotation to specify a list of roles with which a user can access a method. Following code snippet shows how to use the @Secured annotation:

```
@Secured("ROLE_READER")
public String getUsername() {
   SecurityContext securityContext = SecurityContextHolder.getContext();
   return securityContext.getAuthentication().getName();
}
```

As per this code, only users who have the role ROLE_READER, are able to execute the getUsername() method.

Developers can use the @Secured annotation to define a list of roles. Following code snippet shows the use of @Secured annotation:

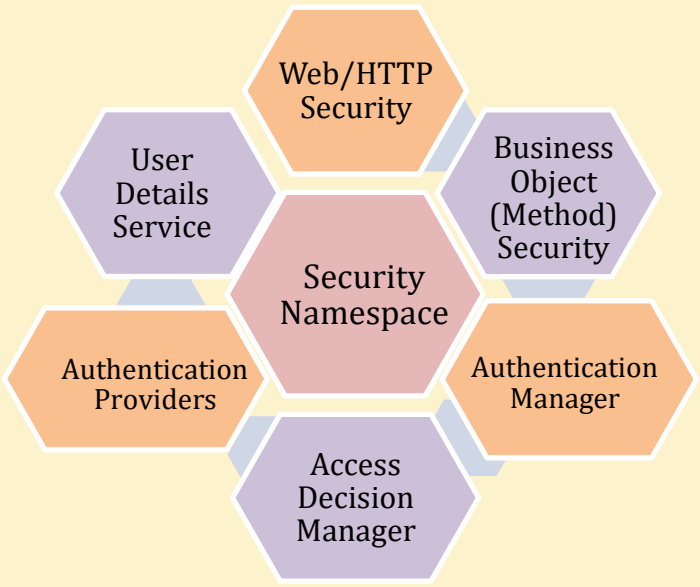
```
@Secured({ "ROLE_VIEWER", "ROLE_WRITER" })
public boolean isValidUsername(String username) {
   return userRoleRepository.isValidUsername(username);
}
```

As per this code, only users with either ROLE_READER or ROLE_WRITER can invoke the isValidUsername() method.

5.4 Security Namespace Configuration

Security Namespace configuration allows to improve the Spring beans application context syntax with elements from additional XML schema by providing several shortcuts to hide the complexity of the framework. This provides a simplified and concise syntax to enable the uses of framework within an application.

Security Namespace can be divided into following areas:



Developers can use a namespace element to configure an individual bean or to define an alternative configuration syntax that hides the complexity of the framework. Using a namespace element refers to adding multiple beans and processing steps to the application context file. For this, perform the following steps:

1. Add the spring-security-config jar in the class path and the schema declaration to the application context file
2. Add the filter declaration to the XML file
3. Enable Web security and add data

Following code snippet shows how to add the spring-security-config jar and add scheme declaration in the application context file:

```
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:security="http://www.springframework.org/schema/security"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-4.0.xsd
        http://www.springframework.org/schema/security
        http://www.springframework.org/schema/security/spring-security-4.0.xsd">
    ...
</beans>
```

Once this code is added, use security as the default namespace rather than beans. Following code snippet shows how to add a filter configuration in the Web.xml file:

```
<filter>
<filter-name>springSecurityFilterChain</filter-name>
<filter-class>org.springframework.web.filter.DelegatingFilterProxy</filter-class>
</filter>
<filter-mapping>
<filter-name>springSecurityFilterChain</filter-name>
<url-pattern>/*</url-pattern>
</filter-mapping>
```

Following code snippet shows how to enable the Web security and should be separate file, for example, spring-security.xml:

```
<beans:beans xmlns="http://www.springframework.org/schema/security"
xmlns:beans="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
  http://www.springframework.org/schema/beans/spring-beans-4.0.xsd
  http://www.springframework.org/schema/security
  http://www.springframework.org/schema/security/spring-security-5.0.xsd">
  <http>
    <intercept-url pattern="/**" access="hasRole('USER')" />
    <form-login />
    <logout />
  </http>
</beans:beans>
```

After completing these steps, developers can now define roles, create data set, and add the log in and log out support to their application.

5.5 Validation

Validation means ensuring that the application or software is doing what it is intended to do. The best way to do this is to validate user inputs to the forms in a Web application. The first step in collecting the data in a Web application is to build forms. Validating user input helps avoid wasted processing and reduce the application response time. Validation occurs at all stages of an application from the presentation layer to the persistence layer.

Developers can validate the user input at client-side as well as at server-side. By default, Spring framework supports JSR-303 specification for bean validation to standardize the validation of Java beans through annotations. To use this, developers just need to add JSR-303 and its implementation dependencies in a Spring application. Spring framework also provides the @Validator annotation for bean validation.

In other words, developers can create validations in any of the following two ways:
- Creating an annotation that confirms to the JSR-303 specs and implements its Validator class.
- Implementing the org.springframework.validation.Validator interface and add it as validator in the Controller class using the @InitBinder annotation.

5.5.1 Bean Validation

The Bean Validation specification defines a framework for declared constraints on JavaBean classes, fields, and properties. Developers can validate only the objects that meet the following requirements:
- Properties to be validated must follow the method signature conventions for JavaBeans read properties, as defined by the JSR 303 specification.
- Static fields and static methods are not validated.
- Constraints can be applied only to interfaces and super classes.

Following list shows a few standard JSR annotations:
- **@NotNull**: Allows to validate that the annotated property value is not null.
- **@AssertTrue**: Allows to validate that the annotated property value is true.
- **@Min**: Allows to validate that the annotated property has a minimum value.
- **@Max**: Allows to validate that the annotated property has a maximum value.
- **@Email**: Allows to validate that the annotated property is a valid email address.

Developers can simply use these annotations in their code. For example, to use the @NotNull annotation, they can add the following code statement to validate their application for null value:

```
import javax.validation.constraints.AssertTrue;
```

5.5.2 Validation Using Spring's Validator Interface

Spring provides a flexible mechanism known as Validator interface for validation. This approach is also extensible. It is mostly used to validate application-specific command objects. An instance of the Validator interface helps validate instances of beans and their associated objects. This interface can be used to validate objects in any layer of an application and supports the encapsulation of validation logic.

Spring Validation is the combination of both, the Validator interface and DataBinder. As stated, Validator interface is used to validate an object and uses Errors object to report any error occurred during validation.

The Validator (org.springframework.validation.Validator) interface provides the following two methods:
- **supports(Class)**: Refers to the class or instance of a class this validator supports.
- **validate(Object, org.springframework.validation.Errors**): Refers to the object being validated that reports the validation error.

Following validation code snippet displays the use of these two methods to validate details of a person:

```
<code class="language-css">public class DetailsValidator implements Validator {
   public boolean supports(Class Test) {
    return Details.class.equals(Test);
   }
   public void validate(Object TestObj, Errors err) {
      ValidationUtils.rejectIfEmpty(err, "name", "name.empty");
         Details  dt = (Details) TestObj;
      if (dt.getPhone) < 0) {
         err.rejectValue("phone", "negativevalue");
      }
   }
</code>
```

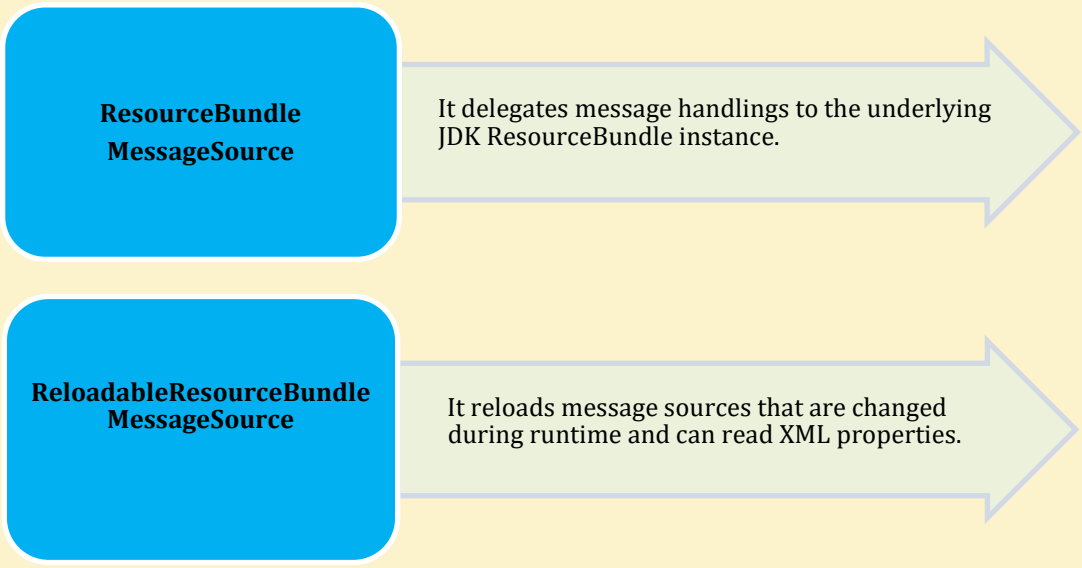
Developers can reuse a validation code in another program if the object in application has the reference to another object. Following code snippet displays how to use the pre-defined Details validator in the new Customer validator:

```
<code class="language-css">public class CustomerValidator implements Validator
{
   private final Validator detailsValidator;
   public CustomerValidator (Validator detailsValidator) {
      if (detailsValidator== null) {
         throw new IllegalArgumentException(
           "The supplied Validator is required and must not be null.");
      }
      if (!detailsValidator.supports(Details.class)) {
         throw new IllegalArgumentException(
           "The supplied must support the validation of Details instances.");
      }
      this.detailsValidator = detailsValidator;
   }
   /**
   * This Validator validates Customer instances, and any subclasses of Customer
too
   */
   public boolean supports(Class Test) {
   return Customer.class.isAssignableFrom(Test);
   }
   public void validate(Object target, Errors errors) {
      ValidationUtils.rejectIfEmptyOrWhitespace(errors, "firstName", "field.required");
      ValidationUtils.rejectIfEmptyOrWhitespace(errors, "surname", "field.required");
      Customer ldv = (Customer) target;
      try {
```

```
            errors.pushNestedPath("address");
            ValidationUtils.invokeValidator(this.detailsValidator, ldv.getAddress(), errors);
        } finally {
            errors.popNestedPath();
        }
    }
</code>
```

5.5.3 Resolving Codes to Error Message

When a validation is executed, Spring returns error messages in the BinderResult object of the BinderResult interface in Controller. This interface extends the interface for error registration capabilities, allows a Validator to be applied, and adds binding-specific analysis and model building. Developers can resolve messages from this object through the FieldError class implementing the MessageSourceResolvable interface, which is a dynamic message resolver interface. Following classes are supported by this interface:



5.5.4 Bean Manipulation and Bean Wrapper

Bean manipulation is the process of getting and setting values from Java beans. The org.springframework.beans package contains interfaces and classes for manipulating beans. BeanWrapper is one such interface that is used by BeanFactory and DataBinder to manipulate Java beans. This interface uses the **BeanWrapper.setPropertyValue()** and **BeanWrapper.getPropertyValue()** methods to manipulate a Java bean. BeanWrapper is a core concept in the Spring Framework.

As the name suggests, BeanWrapper creates a wrapper/envelope which contains an object and help to transfer that object from source to destination without having reference or knowing about object. In simple words, it wraps a bean to carry out actions on that bean, such as setting and retrieving properties. The BeanWrapper is rarely used in application code directly.

Summary

- Spring Security framework focuses on authentication and authorization for securing Spring-based enterprise applications.
- Authentication is the process of validating a user, device, or some other system for whom they claim to be. Authorization refers to the process of deciding whether a user, device, or some other system is permitted to perform an action within the application.
- Java configuration for security allows developers to implement several concepts in the Web application, such as Authorize Requests, Authentication, HTTP Security, and Method Security.
- HTTP Security is a default security configuration provided by Spring Security framework.
- Webflux is a Spring Framework module that contains support for reactive HTTP and WebSocket clients as well as for reactive server Web applications.
- Spring Security framework supports validation to ensure that an application works in the expected manner.
- Developers can create a validation in an application in following two ways:
    - Create an annotation that confirms to the JSR-303 specs and implement its Validator class.
    - Implement the org.springframework.validation.Validator interface and add it as validator in the Controller class using the @InitBinder annotation.

Check Your Progress

1. Which of the following technologies is supported by Spring Security?

A.	HTTP	B.	LDAP
C.	OpenID	D.	All of these

2. Which of the following is not Sprint security supported component?

A.	Bootstrap	B.	FilterChainProxy
C.	Web Security and HTTP Security	D.	WebSecurityConfigurerAdapter

3. Authorization refers to the process of deciding whether a user, _____, or some other system is permitted to perform an action within the _____.

A.	Batch, groups	B.	Device, batches
C.	Device, application	D.	method, application

4. Using a namespace provides a way to implement hidden _____ and helps to avoid conflict with the existing _____.

A.	Classes, definitions	B.	Information, methods
C.	Classes, methods	D.	Information, definitions

5. Which of the following is not related to bean validation?

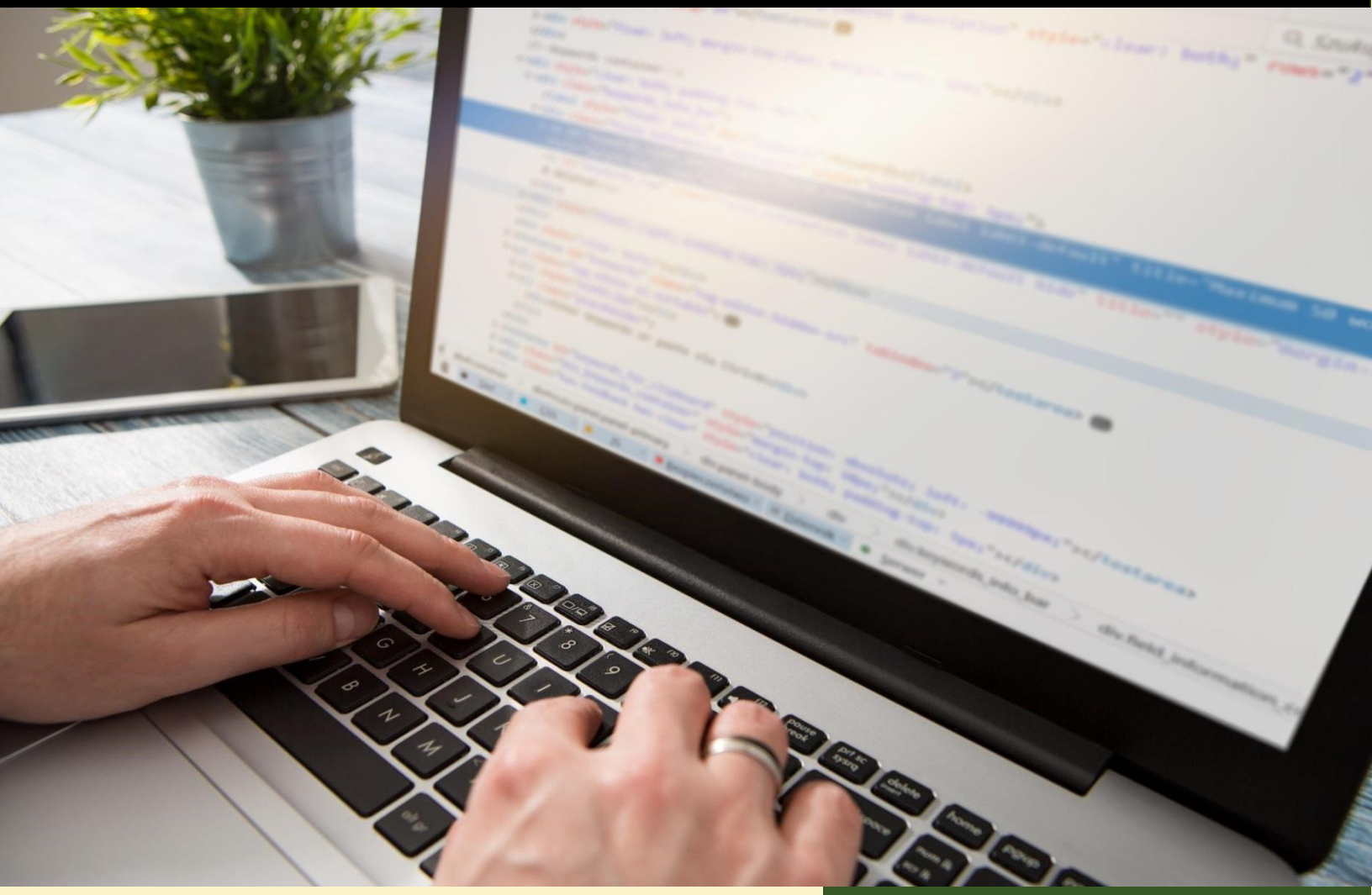
A.	Validator interface	B.	OAuth
C.	Bean Wrapper	D.	Data Binder

Answers

1.	D
2.	A
3.	C
4.	A
5.	B

Session 6

Spring Testing



Objectives

Welcome to the session, Spring Testing.

This session describes the Junit4 and Agile software testing process for Spring applications. In addition, this session explains how to create unit tests for Spring MVC Controller.

In this session, students will learn to:

- Explain testing of Spring applications
- Explain Agile software testing process
- Describe how to test Spring applications using Junit4
- Explain how to create a Spring MVC controller unit test

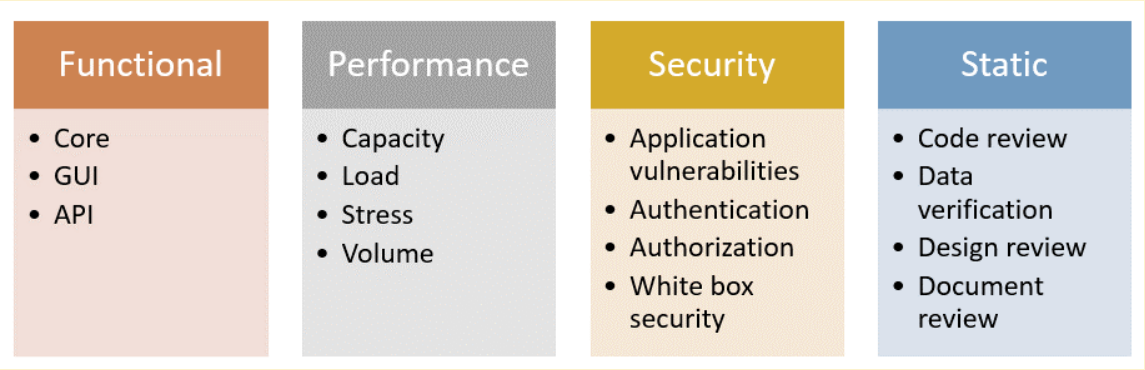
6.1 Testing in Spring Framework

One of the major phases of the Software Development Lifecycle (SDLC) is testing. Testing plays a crucial role in ensuring that a software application is fully functional in any customer environment. Testing also ensures that the application matches the required standards of quality and performance.

So, where does the testing process begin in SDLC? Testing starts at the design phase where the test plan is created. Test plan details the testing methodology and the module or part of the module which will be tested.

The testing process can be both automated and manual. In the manual testing process, as the name suggests, test cases are executed by the tester manually, without using any testing automation tools. In the automated testing process, test cases are executed through the automation tools, thereby saving a lot of time and effort of the tester. One major advantage of the automated testing process is that it can be performed multiple times during separate phases of the software development lifecycle.

Testing is of following types:



In today's software testing world, testing phase is implemented practically by using a testing framework. A testing framework is an environment that is independent of the application being tested and can be easily maintained and customized.

A testing framework provides the following:

- Define the format for test cases and results
- Define the testing process
- Execute the test cases
- Report test results

The Java platform supports several testing frameworks, such as JUnit, TestNG, JTest, and others. JUnit and TestNG are the most commonly used frameworks to test Java based applications.

Testing frameworks are of four types: Modular, Data-driven, Keyword-driven, and Hybrid.

Spring Framework also provides support for Agile testing methods.

6.2 Agile Software Testing Methodology

Agile is a well-known methodology used extensively in the software development and testing phases. Agile methodology refers to the process in which requirements and solutions evolve though collaboration between cross-functional teams. These cross-functional teams include developers, testers, product managers, sales and marketing team, and the software documentation team.

Agile methodology is based on following 12 principles:

- Ensure customer satisfaction by continuous delivery of software modules
- Welcome change in requirements, even in later phases of software development, this helps developers to improve the product they have to deliver to the customer
- Deliver a functional version of the software frequently, probably weekly rather than monthly, this helps in identifying the issues when the product works in an uncontrolled external environment
- Ensure daily collaboration between business stakeholders and developers
- Ensure that the team members are motivated and are trusted by the managers and senior managers
- Have a face-to-face conversation, if the team is co-located, and it is the best form of communication
- Functional software is the primary measure of progress
- Ensure that the application development is done at a sustainable and a constant pace
- Ensure complete and continuous attention to technical excellence and good application design
- Keep the application simple to design and deliver
- Build a self-organizing team to develop best architectures, requirements, and designs
- Brainstorm and reflect on how to become more effective as a team and make changes accordingly

When testing is done within the Agile methodology framework, it includes the following two testing techniques:

- **Unit testing**: Is defined as the process in which each code unit is tested separately to ensure that it is error-free and fully functional.
- **Integration testing**: Is defined as the process in which code units or modules are tested together to ensure that all modules are fully functional within an application.

Figure 6.1 explains the difference between unit and integration testing.

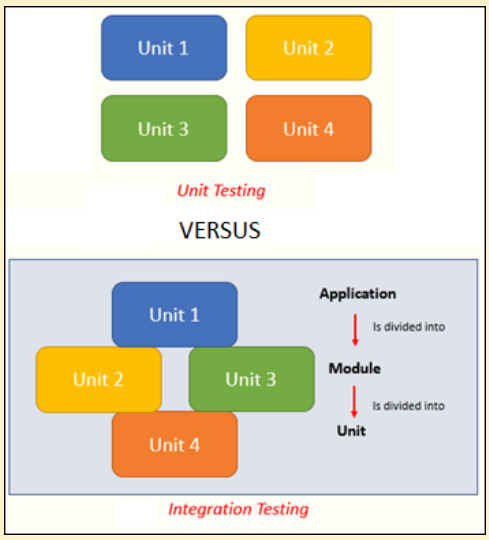


Figure 6.1: Unit vs. Integration Testing

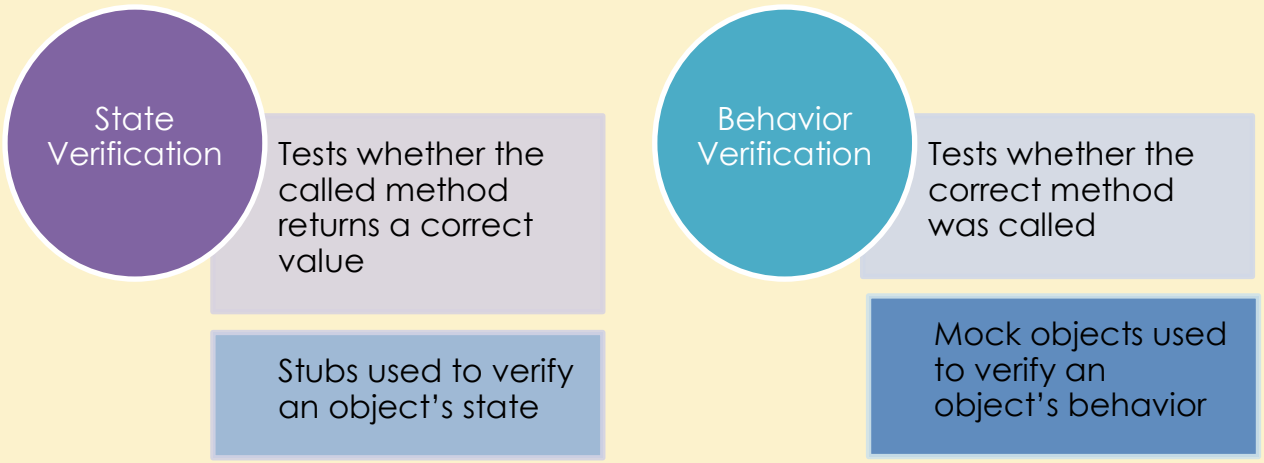
6.2.1 Unit Testing of Spring-based Java Applications

Unit testing is the first step in the software testing process and is generally performed using testing frameworks. Unit testing is the process of testing each individual method, class, module, which are considered units or fundamental functionalities of an application. Each individual unit performs a specific function. Unit testing is typically done to ensure that each unit is error-free and fully functional, thereby improving the quality of the application code.

One such commonly used unit testing framework is JUnit, which is the Java unit testing framework. There are other frameworks such as TestNG which is also popular and these unit testing framework can be integrated with Java IDE to manage test cases.

In Java, unit tests are written for each method or class used in the code. Unit testing can be done on isolated classes, independent classes, and classes with dependencies. Isolated classes are those which are not directly dependent on any other class and are fairly easy to test. However, with classes that are dependent on other class or classes, testing becomes a little difficult. In real-world scenario, units in a Java application very rarely work in isolation and almost all the units are dependent on each other.

The testing process includes testing the both the **state** and **behavior** of a unit, which ensures a complete testing of the unit being tested. There are several ways to test an object's behavior and state, and one such commonly used technique is to simulate unit's dependencies using stubs and mock objects.



- **Stub**: Is a dummy object that simulates a real object and stores pre-defined hard-coded data. A stub uses this data to answer calls during the testing process. A stub can be configured to a specific return value, which can be done by implementing a method in a predetermined way. A stub is generally used when developers do not want the objects to be tested to deal with real data. It must be noted that a stub will never fail a unit test because the stub already has pre-defined data.
- **Mock**: Is an object that imitates an actual application object. A mock object enables the user to test whether the real object in the application interacted with the mock objected as planned, and all actions were performed as specified in the code. A mock object can fail a unit test because it just imitates a real-world object, and it works as per the logic of the application. Java provides several libraries that enable developers to implement mock objects, such as JMock, EasyMock, and Mockito.

The stub and mock objects are used because almost all units being tested have dependencies. Using stubs and mock objects reduces complexities in creating actual objects with same dependencies.

The Mockito framework enables developers to create mock objects that can be used for unit testing of Java applications.

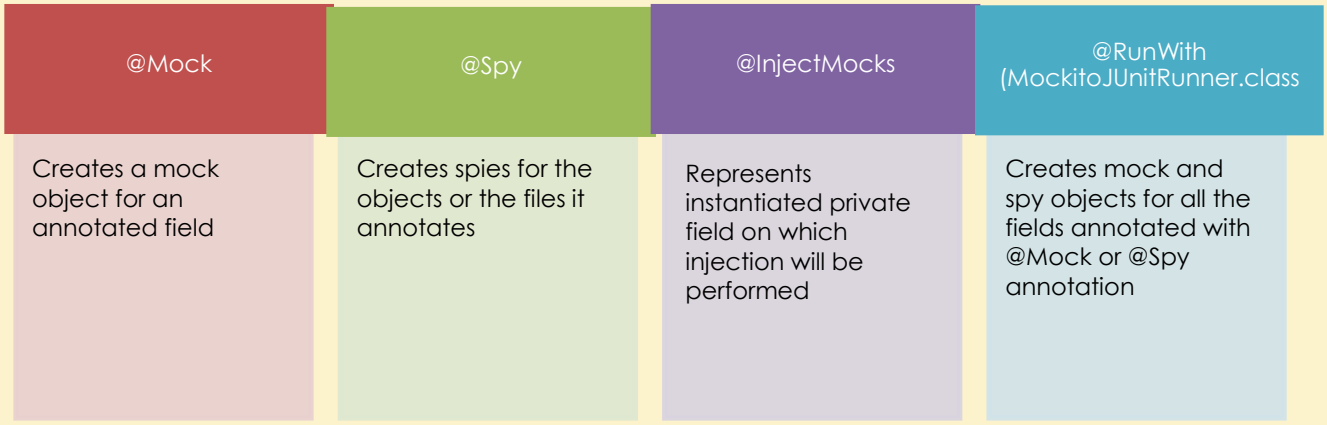
Mockito Framework

The Mockito framework is an open source framework to create and configure mock objects for unit testing, and can be downloaded from:

http://mockito.org/
OR
https://code.google.com/p/mockito/
To use the Mockito framework, developers need to add Mockito JAR to the classpath along with JUnit in their Spring Java project. All the classes and methods developers might use to create and configure mock objects can be imported from org.mockito.Mockito class.

The Mockito framework can also be used in combination with other testing tools.

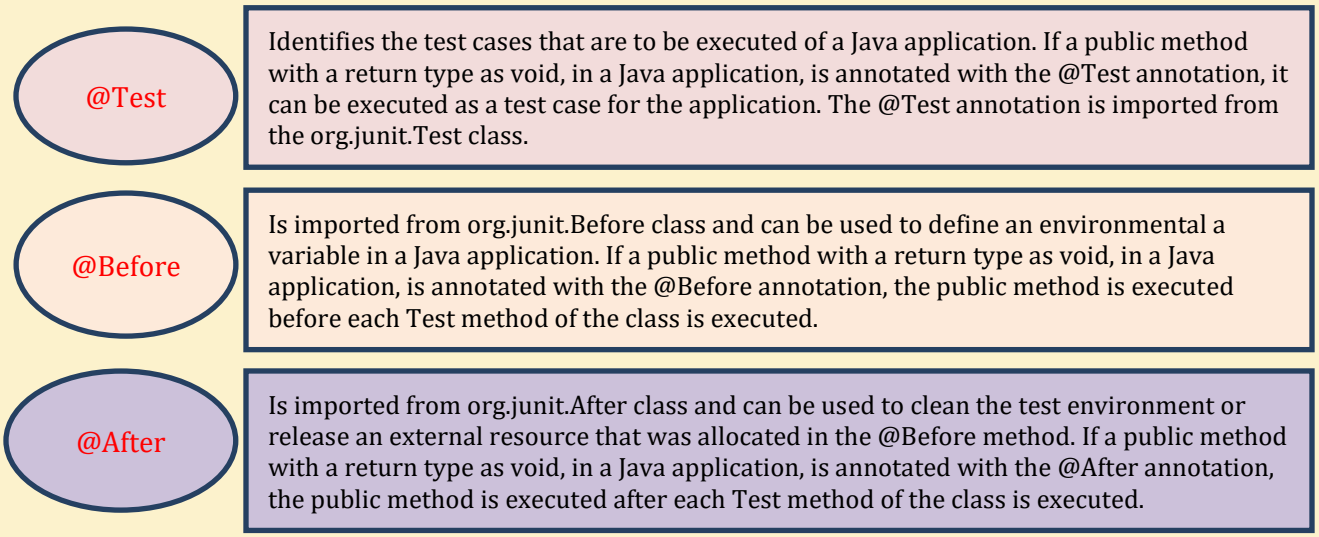
The Mockito JAR uses the following annotations:

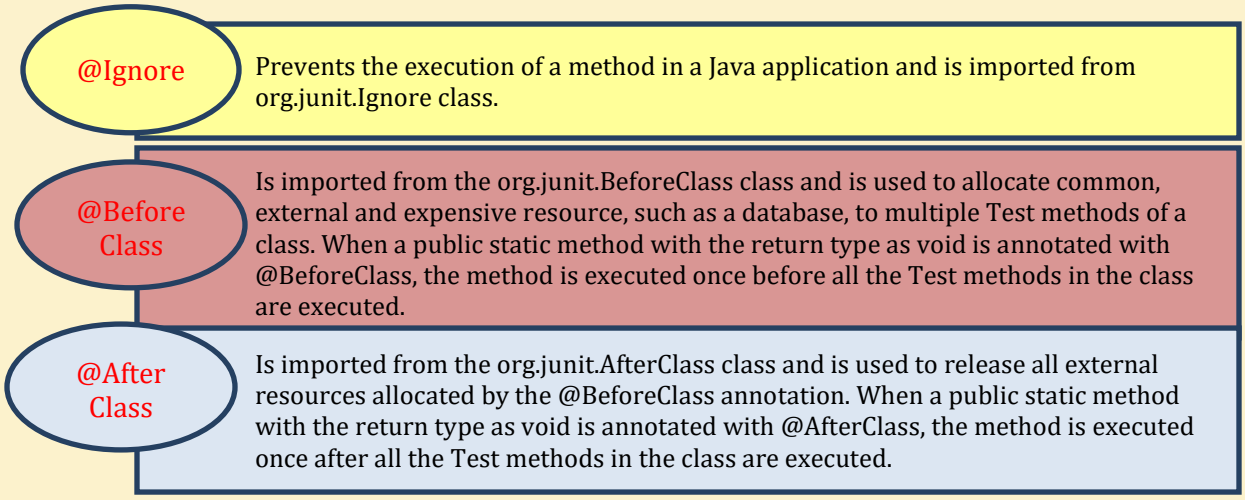


6.2.2 Testing in JUnit4: Annotations and Assert Methods

JUnit4 is the commonly used unit testing framework for Java applications and uses annotations to create automated test cases for Java applications. JUnit4 uses the @Test annotation to annotate the methods that need to be tested. Using annotations ensures that test cases can be executed several times to provide a bug-free Java application.

Some of the commonly used JUnit4 annotations are as follows:





Along with annotations, JUnit4 uses assert methods that enable developers to test specific conditions for the Java applications and compares the expected value with the value returned by the Test method. The assert methods are static methods that are declared in the org.junit.Assert class. These assert methods with the return type as void, are useful in writing Java testcases. These methods verify actual result with expected result and on the basis of which the test cases are marked as pass or fail.

Some of the assert methods are as follows:

- **assertTrue(boolean expected, boolean actual)**: Checks if the Boolean condition is true or not.
- **assertEquals(boolean expected, expected, actual)**: Compares if the two object references are equal using the equals() method, as shown in the following code snippet:

```
@Test
public void testWithServiceResult() {
        assertEquals(TEST_POINT, test.locate(1, 1));
}
```

- **assertNull(Object object)**: Tests if the value of the single object is equal to null, as shown in the following code snippet:

```
@Test
        public void test() {
                String str = null;
                //logic to assign value in String variable
                assertNotNull(str);
        }
```

- **assertFalse(boolean condition)**: Checks if the Boolean condition is false.
- **assertNotNull(Object object)**: Checks if the value of the single object is not equal to null.
- **assertSame(Object object1, Object object2)**: Checks if two object references refer to the same object.
- **assertNotSame(Object object1,Object object2)**: Checks if two object references do not refer to the same object.

6.2.3 Integration Testing of Spring-based Java Applications

Integration testing, as explained earlier, is the testing of all the modules or units combined, to ensure that all the individual modules work in harmony with each other. Integration testing guarantees that all the individual units or modules are integrated properly and interact correctly with each other, giving a fully functional and an error-free application which can work consistently in other environments, such as the production environment.

The main purpose of integration testing is to test the non-functional requirements such as performance, reliability, and so on, of the entire application, with all units working together. In addition, to perform integration tests, developers need to set up the environment similar to the environment in which the application will eventually be deployed.

6.3 Understanding the Spring MVC Test Framework

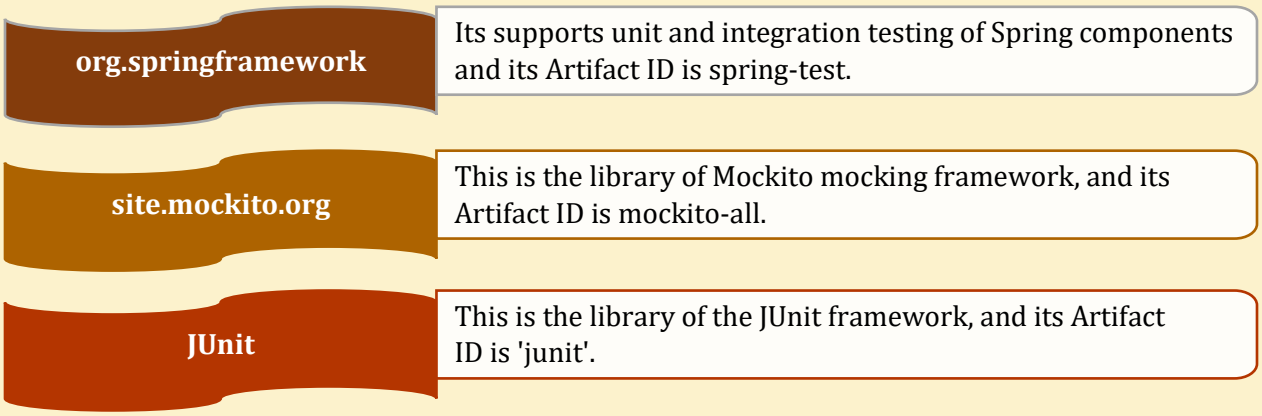
One of the main components of a Java Spring application is the Spring MVC Controller. The Spring MVC test framework allows developers to test Spring MVC applications using the JUnit framework. The Spring MVC test framework was introduced with the Spring Framework 3.2 release.

Before the Spring Framework 3.2 release, the most commonly used way to test a Spring MVC Controller was to create a unit test that instantiates the controller, inject it with mock dependencies. After this, the unit test calls the methods directly using the MockHttpServletRequest and MockHttpServletResponse.

The Spring MVC test framework is used to perform a complete integration testing of the Spring MVC applications through the Spring Configuration Loader. Using the Spring Configuration Loader approach, the Spring application configuration is loaded using the TestContext framework. The TestContext framework loads the spring configuration and injects WebApplicationContext to the MockMvc.

Dependencies

The Spring TestContext framework needs to be configured with the following dependencies:



Following code needs to be included in the XML file of the application to configure dependencies:

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

<dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>${junit.version}</version>
    <scope>test</scope>
  </dependency>

<!-- Mockito specific dependency -->
<dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-all</artifactId>
    <version>1.9.5</version>
  </dependency>
```

Complete pom.xml is generated by STS automatically and looks as follows:

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>org.springframework.samples.service.service</groupId>
  <artifactId>session-6</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <packaging>war</packaging>
  <properties>
    <!-- Generic properties -->
    <java.version>1.6</java.version>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
    <!-- Web -->
    <jsp.version>2.2</jsp.version>
    <jstl.version>1.2</jstl.version>
    <servlet.version>2.5</servlet.version>
    <!-- Spring -->
    <spring-framework.version>5.0.7.RELEASE</spring-framework.version>
    <!-- Logging -->
    <logback.version>1.0.13</logback.version>
    <slf4j.version>1.7.5</slf4j.version>
    <!-- Test -->
    <junit.version>4.11</junit.version>
  </properties>
  <dependencies>
    <!-- Spring MVC -->
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-webmvc</artifactId>
      <version>${spring-framework.version}</version>
    </dependency>
    <!-- Other Web dependencies -->
    <dependency>
      <groupId>javax.servlet</groupId>
      <artifactId>jstl</artifactId>
      <version>${jstl.version}</version>
    </dependency>
    <dependency>
      <groupId>javax.servlet</groupId>
      <artifactId>servlet-api</artifactId>
      <version>${servlet.version}</version>
      <scope>provided</scope>
    </dependency>
```

```
    <dependency>
      <groupId>javax.servlet.jsp</groupId>
      <artifactId>jsp-api</artifactId>
      <version>${jsp.version}</version>
      <scope>provided</scope>
    </dependency>
    <!-- Spring and Transactions -->
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-tx</artifactId>
      <version>${spring-framework.version}</version>
    </dependency>
    <!-- Logging with SLF4J & LogBack -->
    <dependency>
      <groupId>org.slf4j</groupId>
      <artifactId>slf4j-api</artifactId>
      <version>${slf4j.version}</version>
      <scope>compile</scope>
    </dependency>
    <dependency>
      <groupId>ch.qos.logback</groupId>
      <artifactId>logback-classic</artifactId>
      <version>${logback.version}</version>
      <scope>runtime</scope>
    </dependency>
    <!-- Test Artifacts -->
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-test</artifactId>
      <version>${spring-framework.version}</version>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>org.mockito</groupId>
      <artifactId>mockito-all</artifactId>
      <version>1.9.5</version>
    </dependency>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>${junit.version}</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>
```

Annotations

The Spring Framework provides the following two annotations that are used to perform unit and integration testing with the TestContext framework:

- **@ContextConfiguration**: This annotation is used to set the ApplicationContext for the test classes by specifying the actual configuration file and its file path. This annotation allows developers to specify multiple configuration files using a comma separator. This annotation can either refer to a class or an XML file. For example, in the following code snippet, a single XML file is referenced.

```
package com.learning;
import static org.junit.Assert.*;
import org.junit.AfterClass;
import org.junit.BeforeClass;
import org.junit.Test;
public class MyTest {
	@BeforeClass
	public static void setUpBeforeClass() throws Exception {
	}
	@AfterClass
	public static void tearDownAfterClass() throws Exception {
	}
	@Test
	public void test() {
		fail("Not yet implemented");
	}
}
```

Similarly, in the following code snippet, a configuration class is being referenced:

```
@ContextConfiguration(class=TestConfig.class)
public class SampleClass {
	//code to test
}
```

In the following code snippet, a configuration class and an XML file are referenced:

```
package com.learning;
import static org.junit.Assert.*;
import org.junit.AfterClass;
import org.junit.BeforeClass;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;
import org.springframework.test.context.web.WebAppConfiguration;
```

```
@ContextConfiguration(classes = SampleConfiguration.class)

public class MyTestWithContext {

	@BeforeClass
	public static void setUpBeforeClass() throws Exception {
	}
	@AfterClass
	public static void tearDownAfterClass() throws Exception {
	}
	@Test
	public void test() {
		fail("Not yet implemented");
	}
}
```

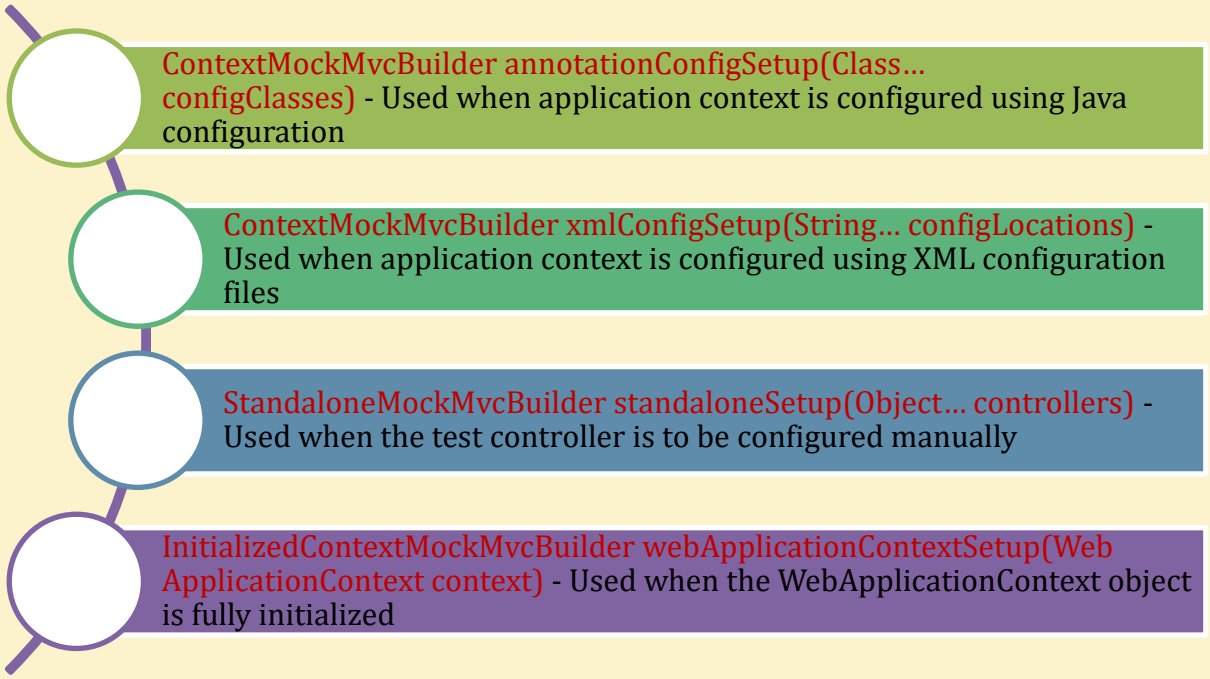
@WebAppConfiguration: This annotation is used to create a Web version of the application context in the Spring Framework and is a class-level annotation. This annotation is always used with the @ContextConfiguration annotation. The use of this annotation indicates that the ApplicationContext (which is loaded for an integration test and used by the specified class) is the instance of WebApplicationContext.

Following code snippet illustrates the use of this annotation:

```
package com.learning;
import static org.junit.Assert.*;
import org.junit.AfterClass;
import org.junit.BeforeClass;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;
import org.springframework.test.context.web.WebAppConfiguration;
@WebAppConfiguration
@ContextConfiguration(classes = SampleConfiguration.class)
public class MyTestWithContext {
	@BeforeClass
	public static void setUpBeforeClass() throws Exception {
	}
	@AfterClass
	public static void tearDownAfterClass() throws Exception {
	}
	@Test
	public void test() {
		fail("Not yet implemented");
	}
}
```

MockMvc

The MockMvc is used to create test cases for Spring applications developed using Spring MVC and is a key component of the Spring MVC Test framework. To start using the Spring MVC Test framework, developers need to create an instance of MockMvc. The MockMvc imitates the entire Spring MVC infrastructure and is created using the implementations of the MockMvcBuilder interface. The MockMvcBuilder interface consists of the following four static methods:



Following code snippet illustrates how a MockMvc instance is created using the MockMvcBuilders. The testSetup method is called after the instance of the controller class is passed as the parameter and then built using the build() method.

```
private MockMvc mockMvc;
@Before
public void setup() {
        this.mockMvc = MockMvcBuilders.testSetup
(employeeController).build();
}
```

@RunWith(SpringJUnit4ClassRunner.class)

This JUnit annotation executes the tests included in a class that is annotated by the @RunWith annotation. This annotation extends a class annotated by the @RunWith annotation by invoking the class which is passed as the parameter. As a result, the tests in the annotated class are executed by a runner class instead of the JUnit framework's built-in API. If developers need to use JUnit's class runner to execute the test cases

within Spring's ApplicationContext environment, developers need to pass SpringJUnit$ClassRunner as the parameter.

Following is an example of implementing the Mockito framework for unit testing. In this example, we will create a unit test with Mockito and Junit. The steps to create this unit test are as follows:

Step 1: Create a Java project **TestingDemo** with three files namely, **TestPoint.java**, **Point.java**, and **TestService.java**.

The project structure should look as shown in Figure 6.2.

Developers will be performing JUnit tests on code specified in these three files using a Mock class.

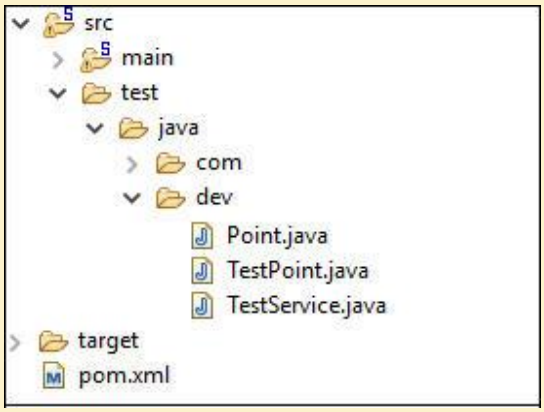


Figure 6.2: Project Structure for TestingDemo

Add the following codes to the respective Java files.

```
//TestPoint.java
package dev;
public class TestPoint {

        private final TestService testService;

        public TestPoint(TestService  testService) {
      this.testService = testService;
   }
        public Point locate(int x, int y) {
                if (x < 0 || y < 0) {
                        return new Point(Math.abs(x), Math.abs(y));
                } else {
                        return testService.geoLocate(new Point(x, y));
                }
        }
}
```

```
//Point.java
package dev;
public class Point {
	Integer x;
	Integer y;
		public Point(Integer x, Integer y) {
		this.x = x;
		this.y = y;
	}
	public Integer getX() {
		return x;
	}
	public void setX(Integer x) {
		this.x = x;
	}
	public Integer getY() {
		return y;
	}
	public void setY(Integer y) {
		this.y = y;
	}
}
```

```
//TestService.java
package dev;

public class TestService {
	public Point geoLocate(Point point) {
		return point;
	}
}
```

Step 2: Add libraries for JUnit and Mockito that are required for running the Mock in the project.

Step 3: Use the code shown here to initialize the Mock class that will be used for JUnit test cases.

```
package dev;
import org.junit.runner.RunWith;
import org.mockito.Mock;
import org.mockito.runners.MockitoJUnitRunner;
@RunWith(MockitoJUnitRunner.class)
public class FunctionalityTest {
	@Mock
	private TestService testServiceMock;
}
```

Step 4: Now, in the Mock class created in Step 3, add the JUnit test scenarios which will test our code written in Step 1. A sample implementation code is shown here.

```
FunctionalityTest.java
package test;
import org.junit.Before;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.mockito.Mock;
import org.mockito.runners.MockitoJUnitRunner;
package dev;

import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertTrue;
import static org.mockito.Matchers.any;
import static org.mockito.Mockito.when;

import org.junit.Before;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.mockito.Mock;
import org.mockito.runners.MockitoJUnitRunner;

@RunWith(MockitoJUnitRunner.class)
public class FunctionalityTest {
	private static final Point TEST_POINT = new Point(11, 11);
	@Mock
	private TestService testServiceMock;
	private TestPoint test;

	@Before
	public void setUp() {
	when(testServiceMock.geoLocate(any(Point.class))).thenReturn(TEST_POINT);
		test = new TestPoint(testServiceMock);
	}

	@Test
	public void testWithServiceResult() {
		assertEquals(TEST_POINT, test.locate(1, 1));
	}
	@Test
	public void testLocalResult() {
		Point expected = new Point(1, 1);
		assertTrue(arePointsEqual(expected, test.locate(-1, -1)));
	}
	private boolean arePointsEqual(Point p1, Point p2) {
		return p1.getX() == p2.getX() && p1.getY() == p2.getY();
```

```
        }
}
```

Step 5: Now, run the project as JUnit Test, as shown in Figure 6.3.

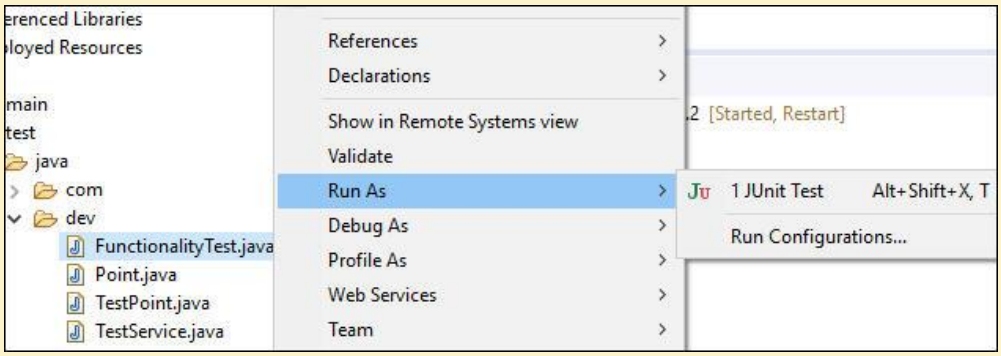


Figure 6.3: Project Executed as Junit Test

After the project is run, the test results indicate that two tests are passed, as shown in Figure 6.4.

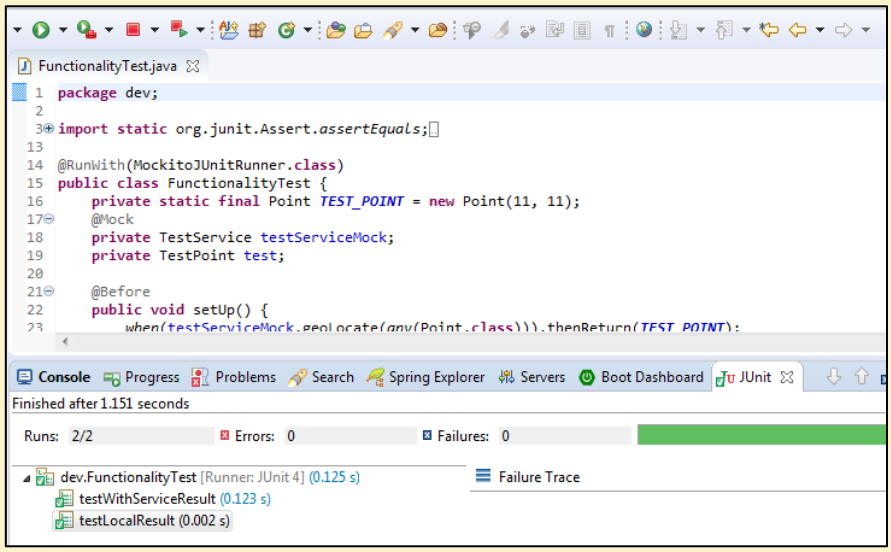


Figure 6.4: Test Results of the Project

Summary

- Testing plays a crucial role during SDLC in ensuring that a software application is fully functional in any customer or external environment.
- Testing can be manual or automated. In Manual testing, the tester executes all the test cases, where as in automation testing, automation tools are used. Automated testing is fast and can be performed multiple times.
- With the evolution of testing as a process, testing is generally performed through testing frameworks, which are environments independent of application and can be easily customized.
- Agile methodology refers to the process in which requirements and solutions evolve though collaboration across cross-functional teams.
- Agile testing is typically performed through unit and integration testing techniques.
- Unit testing is the process of testing each individual method, class, module, which are considered units or fundamental functionalities of an application.
- JUnit4 is the commonly used unit testing framework for Java applications and uses annotations to create automated test cases for Java applications.
- Integration testing is the testing of all the modules or units combined, to ensure that all the individual modules work in harmony with each other.
- The Spring MVC test framework allows developers to test Spring MVC applications using the JUnit framework.

Check Your Progress

1. Which of the following is used to create test cases for Spring applications developed using Spring MVC and is a key component of the Spring MVC Test framework?

A.	MvcTest	**B.**	MockMvc
C.	MvcMockTest	**D.**	Junit Test Runner

2. The assertFalse() method checks if the _____ is false.

A.	Object condition	**B.**	Null value of the object
C.	Boolean condition	**D.**	Object value

3. Which of the following testing types is the first step of an application testing process?

A.	White box testing	**B.**	Unit testing
C.	Integration testing	**D.**	Regression testing

4. Which of the following is used to verify an object's behavior?

A.	Mock	**B.**	Stub
C.	Mod	**D.**	Block

5. Which of the following are not supported by the Mockito JAR?

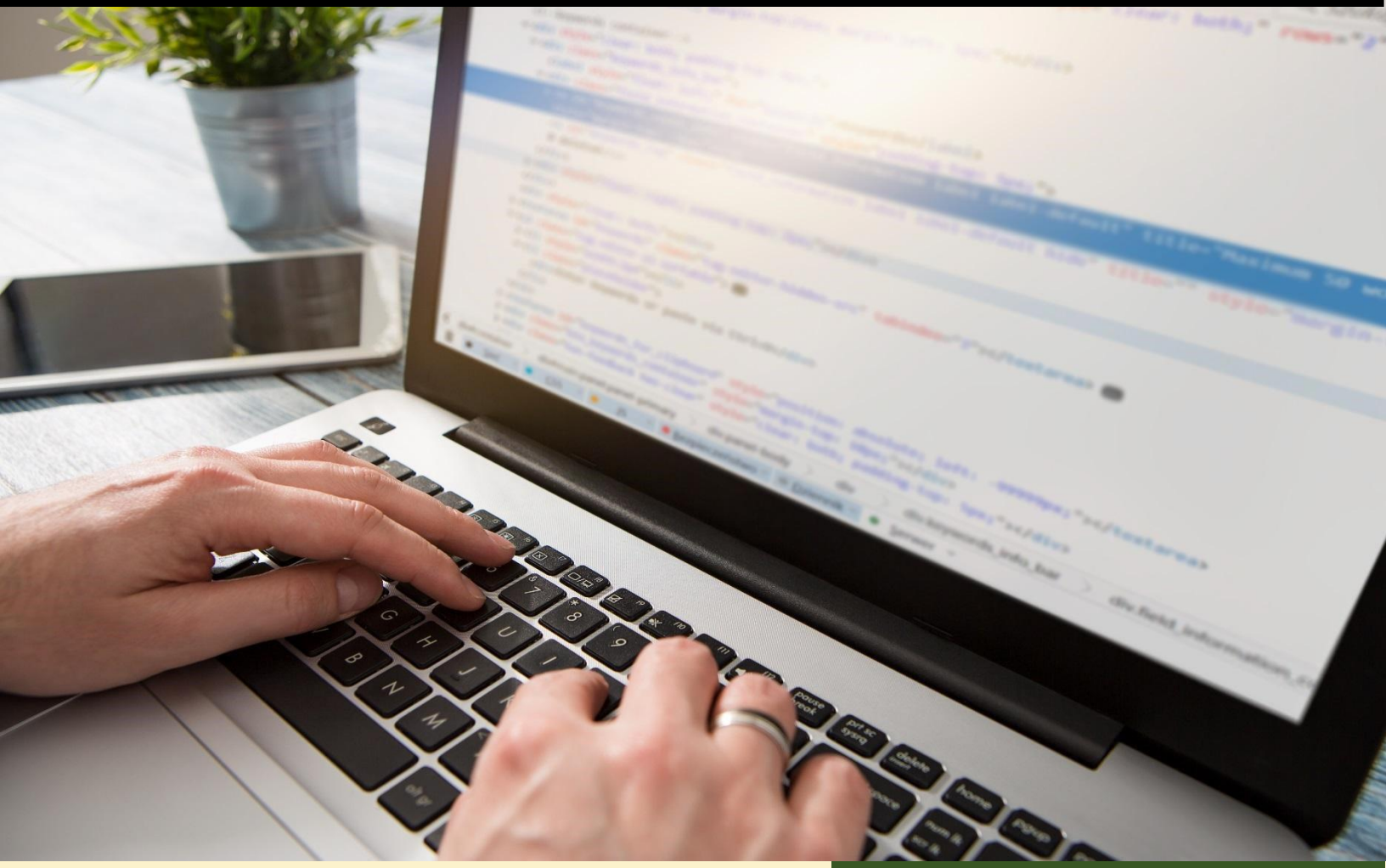
A.	@Spy	**B.**	@InjectsMock
C.	@Spring	**D.**	@RunWith

Answers

1.	B
2.	C
3.	B
4.	A
5.	B and C

Session 7

Spring Boot



Objectives

Welcome to the session, Spring Boot.

This session introduces Spring Boot as an extended Spring Framework. It also explains how to create secured applications using Spring Boot. In addition, it describes managing the API and database in Spring Boot.

In this session, students will learn to:
- Explain the Spring Boot Framework
- Explain how to use Spring Boot to create a secured Web application and API
- Explain how to use Spring Boot to access database

7.1 Spring Boot Introduction

Spring Boot Framework is a framework that is implemented over the existing Spring Framework. Spring Boot makes use of an approach which provides pre-defined code and annotation configuration that can be used to build, package, and deploy new Spring applications with minimal configurations. In other words, Spring Boot helps developers to build and deliver Jar files to run as stand-alone applications.

Following equation explains Spring Boot in simple terms:
Spring Boot = Spring Framework – XML Configuration + Integrated Server

Figure 7.1 illustrates how Spring Boot acts as an interface between Spring and end user.

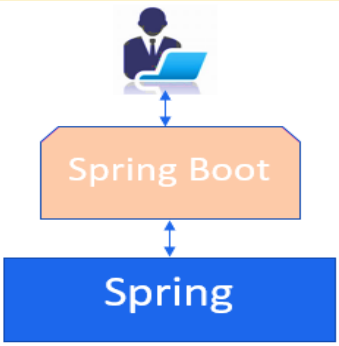
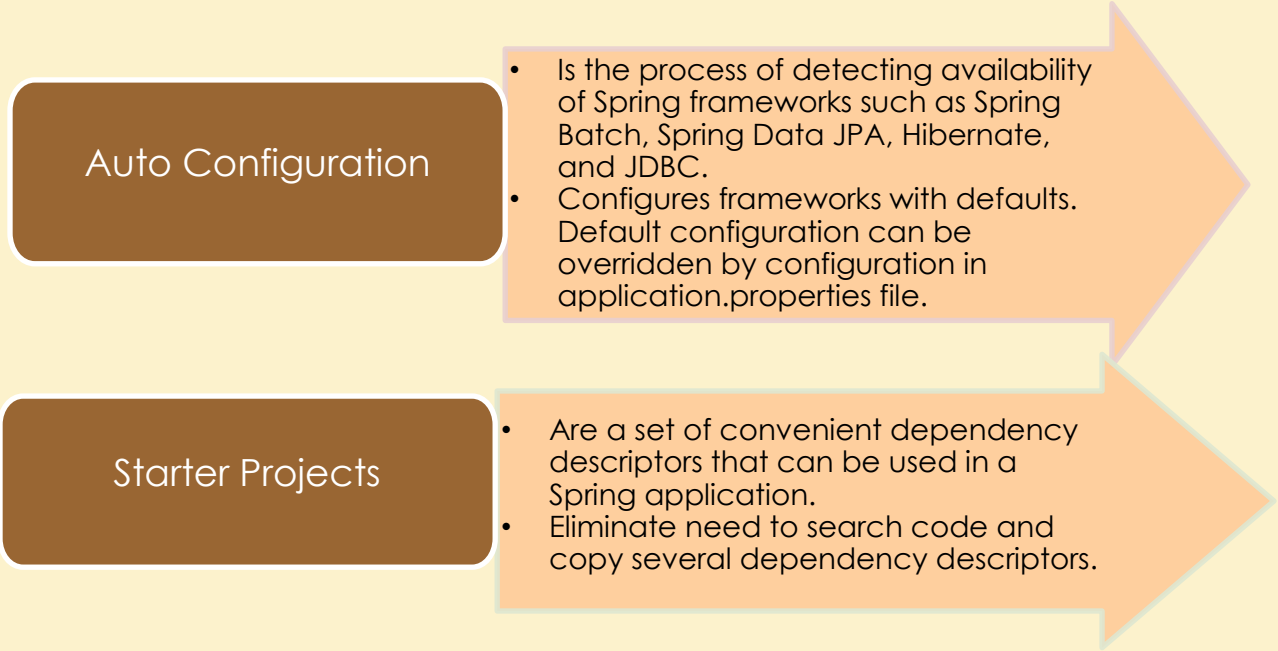


Figure 7.1: Spring Boot Placement

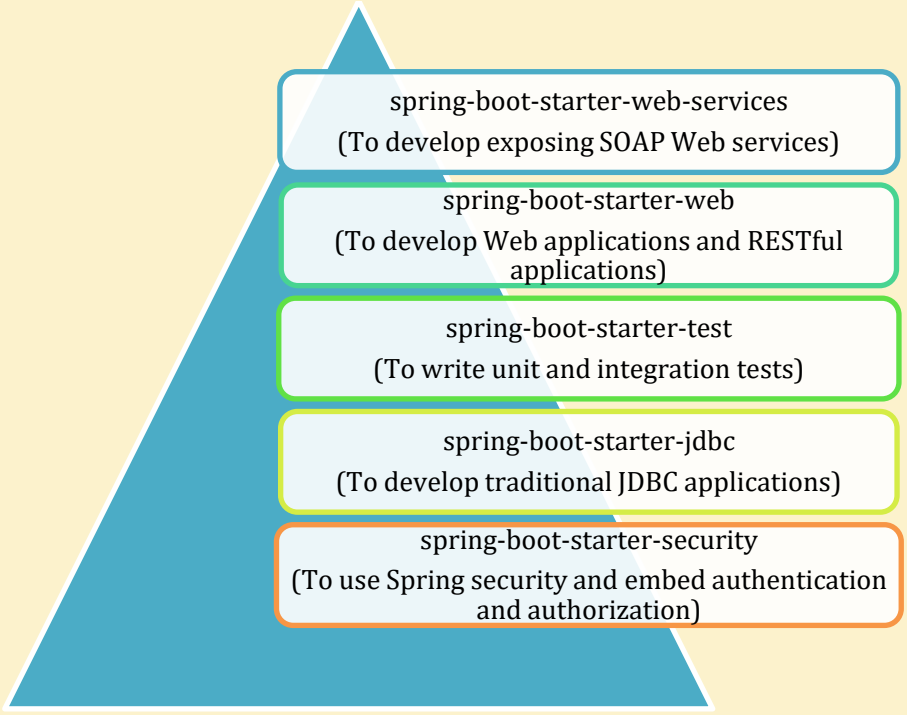
7.1.1 Understanding Spring Boot

Spring applications contain a huge amount of XML or Java Bean configuration. On the other hand, Spring Boot reduces the amount of configuration required through a combination of its two components: auto configuration and starter projects.



For example, if developers want to add JPA for database access in a Spring application, they can just include spring-boot-starter-data-jpa dependency.

Few other starter projects provided by Spring Boot are as follows:



In addition to auto configuration and project starters, Spring Boot has following components:

Spring Boot Core	• Serves as the base for other modules • Also provides functionality that can be implemented on its own, such as automatically binding environment properties to Spring bean properties
Spring Boot Command Line Interface	• Helps start and stop the Spring Boot applications
Spring Boot Actuator	• Can be added to the application to enable enterprise features, such as Security, and so on • Is similar to auto configuration components; it uses auto detection feature

The latest version of Spring Boot is 1.3.3. The minimum supported version of Spring is Spring Framework 4.2.2.

7.1.2 Features of Spring Boot

Spring Boot offers a host of features, some of the prominent ones are as follows:

- **Web application development**: Spring Boot is suitable for Web application development because it allows developers to create a self-contained HTTP server using embedded Tomcat, Jetty, or Undertow and use the spring-boot-starter-web module to run the application quickly.
- **Application properties Files**: Spring Boot provides wide range of application properties that developers can use in the properties file of their project to organize application properties, such as server-port.
- **Application Events and Listeners**: Spring Boot uses events to manage several tasks by allowing developers to create factories files in META-INF folder (META-INF/spring.factories) to add listeners. Developers can use these factory files by using the ApplicationListener key.
- **Remote application management:** Spring Boot provides admin-related features to access and manage applications remotely. Developers can use the spring. application. admin.enabled property to enable this feature.
- **Type-safe Configuration:** Spring Boot supports type-safe configuration to monitor and validate application configuration.
- **Security**: Spring Boot applications are secured with basic authentication on all HTTP endpoints.
- **YAML and Externalized Configuration Support**: Spring Boot provides the SpringApplication class, which automatically supports YAML and allows developers to specify hierarchical configuration. In addition, Spring Boot allows developers to use YAML files to externalize configuration so that developers can use the same application in different environments.

> *YAML is a data serialization language, which is typically used for creating configuration files or in applications where data is being stored or transmitted across a network. Data serialization is the process of converting data structures or object states in a format that can be stored or transmitted and later reconstructed to the original format.*
>
> *YAML was initially named as 'Yet Another Mark-up Language'. However, it was later renamed as YAML Ain't Mark-up Language.*

7.1.3 Benefits of Spring Boot

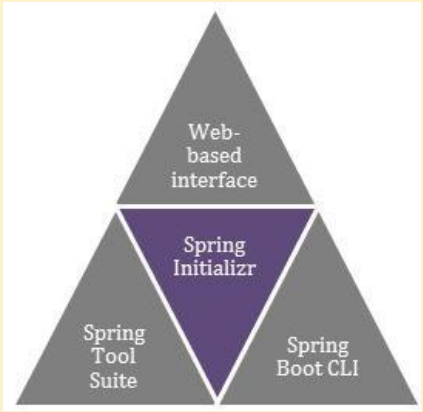
Spring Boot provides the following benefits:

- Spring Boot helps increase individual and team productivity by removing all XML based configurations and providing annotations for using the Spring Framework. It allows developers to start their application with a very minimum annotation in almost no time.
- Spring Boot provides cloud support for application configuration, tools, and clients. It is also compatible with Cloud Native and provides seamless integration with the Cloud Foundry and Pivotal cloud application platforms.

- Spring Boot helps developers to expedite the development process by using the integrated server. For this, Spring Boot provides an attached Tomcat/Jetty server with the compiled Jar.
- Spring Boot provides a unified ecosystem of libraries and standards. This is helpful if developers and their teams are using the Microservices methodologies. This in turn provides common platform and library support, cloud support, and reduced setup and configuration time in a development environment.
- Spring Boot provides support for third-party Open Source Library, such as Netflix OSS, No-SQL DB, Distributed Cache, and so on.

7.2 Creating a Spring Boot Project

Using Spring Initializr is one of the simplest ways to create a Spring Boot application. Spring Initializr is a Web application that does not generate any application code, but creates a project structure (the build system and its coordinates, the language and version, the packaging, and finally the dependencies to add to the project). Developers just need to write their application code. Spring Initializr is available at start.spring.io.



Developers can create their own instance of Spring Initializr by using the Jars as libraries in their application. Spring Initializr provides an extensible API to generate projects and inspect the metadata used to generate projects. Figure 7.2 shows the ways in which developers can use Spring Initializr.

Figure 7.2: Ways to Use Spring Initialzr

7.2.1 Using Spring Initializr Web Interface

To use Spring Initializr via Web interface, developers need to enter http://start.spring.io in the Address bar of their browser.

Figure 7.3 displays the Spring Initializr form that is displayed when developers enter the URL.

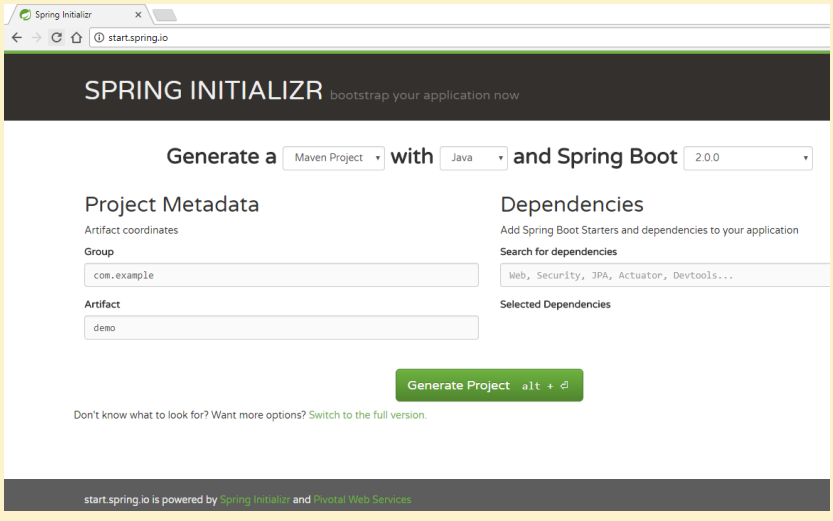


Figure 7.3: Spring Initializr Web Interface

The left panel on the form allows developers to enter project metadata, such as project name, base package name, and version of Java to build project. The right panel on the form allows developers to specify project dependencies. In other words, developers can specify the kind of functionality that they want to develop through the application. Enter the required details and click **Generate Project** to generate a project structure as a zip file.

The generated project includes the following files:

- Empty **static** directories (as depicted in Figure 7.4, which contains app-controllers, app-directives, and so on) where developers can keep static content (JavaScript, stylesheets, images, and so on).
- Template directories where developers can keep templates that render model data.
- build.gradle (if developers chose Gradle) or pom.xml (if developers chose Maven).
- Application.java class with a main() method.
- ApplicationTests.java class (an empty JUnit test class to load Spring).
- Application context file using Spring Boot auto-configuration.
- Application.properties file to add configuration properties.

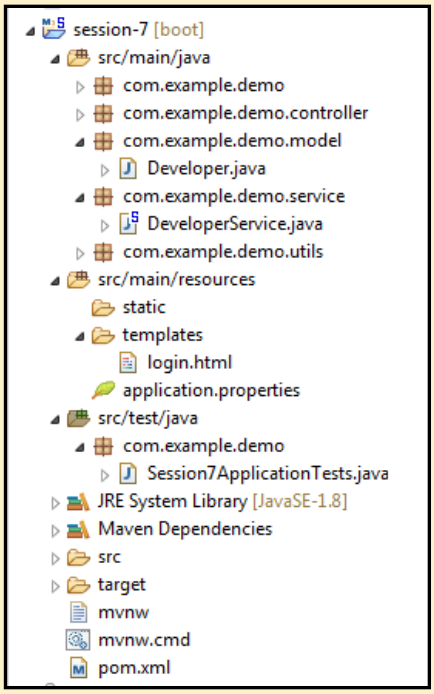


Figure 7.4: Project Structure of Spring Boot Application

7.2.2 Using Spring Tool Suite

Spring Tool Suite is an IDE that integrates with Spring Initializr to help developers to create Spring Boot applications.

Developers must be connected to Internet to use Spring Tool Suite. To download Spring Tool Suite (STS), use the URL http://spring.io/tools/sts.

Once STS is launched, use the Spring Boot template wizard to create a Spring Boot application easily. Refer to Figure 7.5.

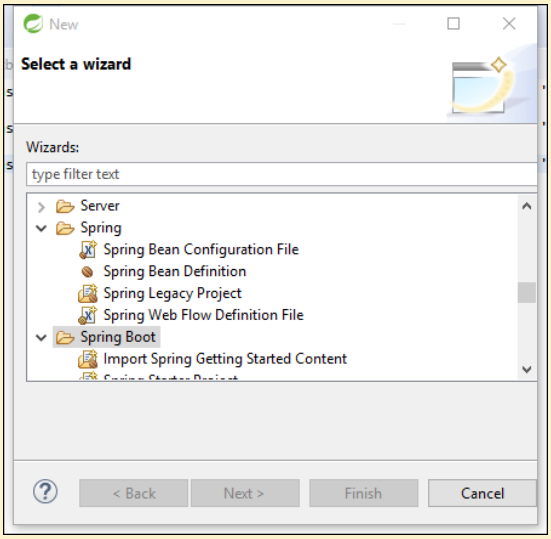


Figure 7.5: Spring Boot Wizard

7.2.3 Using Spring Boot Command Line Interface (CLI)

Spring Boot CLI is a Spring Boot utility software to run and test Spring Boot applications from the command prompt. It can be downloaded from the following URL: https://repo.spring.io/release/org/springframework/boot/spring-boot-cli/2.0.3.RELEASE/spring-boot-cli-2.0.3.RELEASE-bin.zip

When developers use this tool to run Spring Boot applications, internally it uses Spring Boot Starter and Spring Boot AutoConfigurate components to sort out all dependencies and execute the application. Once it is installed, developers can run various commands from the command prompt. For example, developers can use the init command to create a baseline Spring Boot project. When developers run this command, the system invokes Spring Initializr and creates a zip file having the default project structure with a Maven pom.xml build specification.

The syntax to use the init command is as follows:

```
spring init
```

The file created will have the default name, demo.zip. Refer to Figure 7.6.

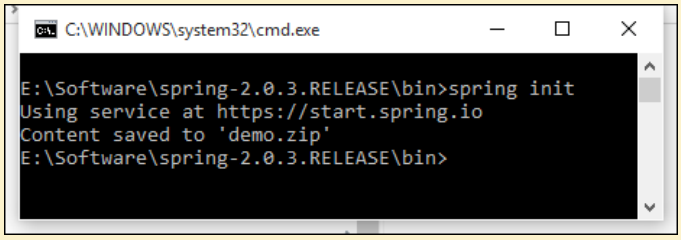


Figure 7.6: Running init Command

Developers can also add dependencies to their project and specify the build type as Gradle as shown here:

```
spring init -dweb,jpa,security
```

This command adds Web, JPA, and security starters as dependencies in pom.xml.

As shown in Figure 7.7, one can also specify a name for the zip file.

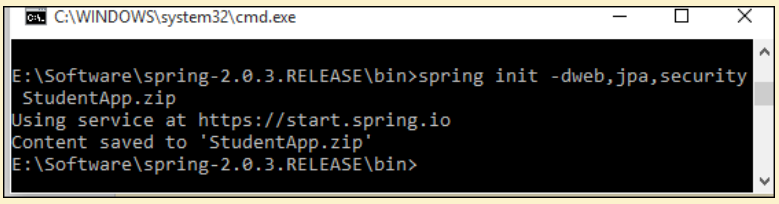


Figure 7.7: Specifying Dependencies and Project Name

The command to specify the build type as Gradle is as follows:

```
spring init -dweb,jpa,security --build gradle
```

Developers can use the following command to run the application:

```
spring run <<application name>>.groovy
```

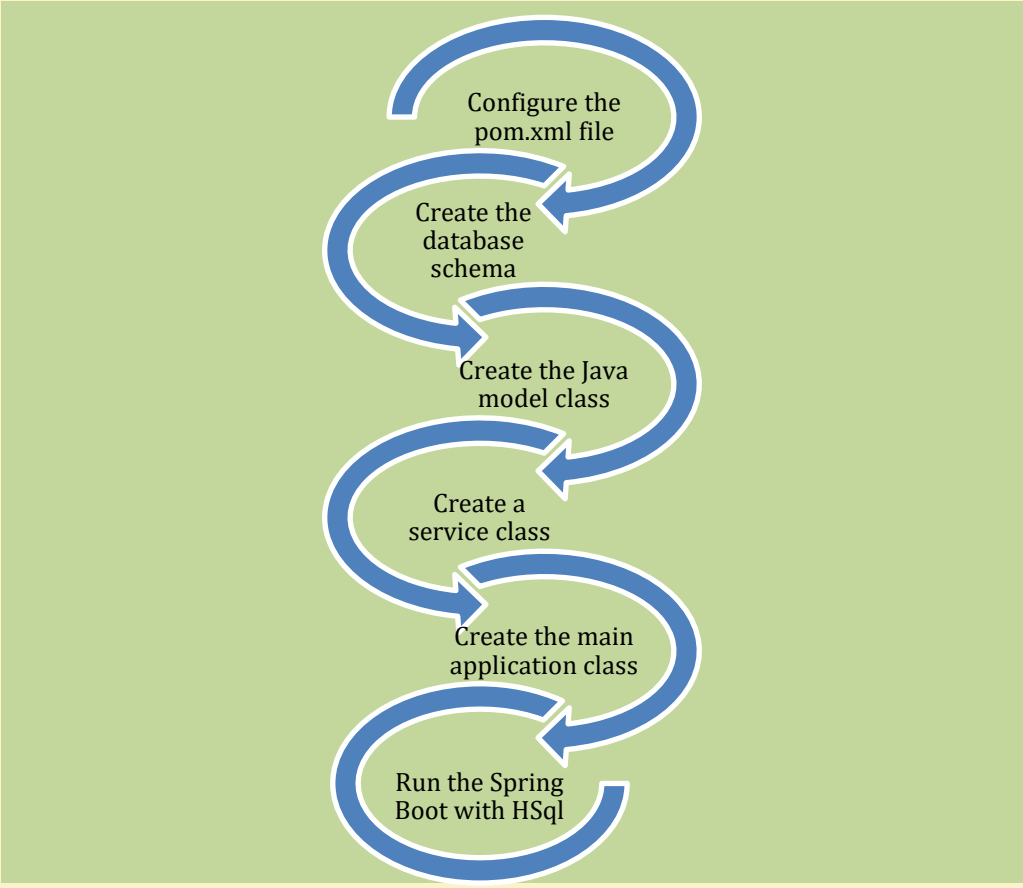
7.3 Debugging and Managing the Application

Spring Boot allows developers to debug and manage their Web applications. To do that, developers can create their APIs and store the relevant data in their database. Spring Boot allows developers to create an API and manage their database.

7.3.1 Storing Data in Database

Spring Boot supports several relational databases, such as Oracle, MySQL, and H2 (an in-memory database). By default, if Spring Boot finds any in-memory database, such as H2, HSQLDB, or Derby in the classpath, it sets up an in-memory embedded database for use. An in-memory database is one that has a lifespan between start and stop of an application. In other words, it is created when an application starts and destroyed when the application is stopped. Alternatively, developers can decide to use any other relational database.

An example shown here will use HSql as an in-memory database and Java Persistence API (JPA) for Spring Boot applications. The steps to configure an in-memory database in Spring Boot and store data in it are as follows:



These steps are discussed in detail as follows:

1. **Open the generated Spring Boot application.**
2. **Configure the pom.xml file.**
   The pom file specifies various dependencies of the project.

   For the current application, developers need to include the JDBC dependency in the pom.xml file to get access to JdbcTemplate and other JDBC libraries. They need to use the spring-boot-starter-jdbc module to configure the DataSource bean.

   Following code snippet shows how to add JDBC dependencies:

```
<dependencies>
   <dependency>
  <groupId>org.hsqldb</groupId>
  <artifactId>hsqldb</artifactId>
  <scope>runtime</scope>
 </dependency>
  </dependencies>
```

3. **Create the database schema**. Developers need to create a schema to map for a new table. Assume that a table 'Users' is to be created. Add the following code in a file named schema.sql and save under the resources folder of the project.

   src/main/resources/schema.sql

```
CREATE TABLE Users(userId INTEGER NOT NULL,Name VARCHAR(100) NOT
NULL,Email VARCHAR(100) DEFAULT NULL,address VARCHAR(100) DEFAULT
NULL,PRIMARY KEY (userId));
```

   Similarly, create Data.sql in src/main/resources to create rows as follows:

```
INSERT INTO Users(userId, Name, Email, address) VALUES (1011, 'John',
'J@gmail.com', 'Florida');

INSERT INTO Users(userId, Name, Email, address) VALUES (1012, 'Tom',
'T@gmail.com', 'Chicago');

INSERT INTO Users(userId, Name, Email, address) VALUES (1013, 'Pete',
'P@gmail.com', 'Pittsburg');
```

4. **Create the Java model class.** The model class will define the entity and its attributes.

Use the following code snippet to create the Java model class named Developer:

```
package com.example.demo.model;
public class Developer {
	private Integer id;
	private String name;
	private String email;
	private String address;

	public Integer getId() {
		return id;
	}
	public void setId(Integer id) {
		this.id = id;
	}
	public String getName() {
		return name;
	}
	public void setName(String name) {
		this.name = name;
	}
	public String getEmail() {
		return email;
	}
	public void setEmail(String email) {
		this.email = email;
	}
	public String getAddress() {
		return address;
	}
	public void setAddress(String address) {
		this.address = address;
	}
	@Override
	public String toString() {
		return "Developer [id=" + id + ", name=" + name + ", email=" + email
+ ", address=" + address + "]";
	}
}
```

5. **Create a service class**. The service class and DeveloperService, uses JdbcTemplate to insert and retrieve data from the HSql database. The service class will contain the methods createUser (to add a new row to the table) and findAllUsers (to fetch all the rows from the table).

Use the following code snippet to create DeveloperService.java:

```
package com.example.demo.service;
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.PreparedStatementCreator;
import org.springframework.jdbc.support.GeneratedKeyHolder;
import org.springframework.jdbc.support.KeyHolder;
import org.springframework.stereotype.Component;
import org.springframework.transaction.annotation.Transactional;
import com.example.demo.model.Developer;
import com.example.demo.utils.DeveloperRowMapper;

@Component
public class DeveloperService {
	@Autowired
	private JdbcTemplate jdbcTemplate;
	@Transactional(readOnly = true)
	public List<Developer> findAll() {
		return jdbcTemplate.query("select * from Users", new
		 DeveloperRowMapper());
	}
	@Transactional(readOnly = true)
	public Developer findUserById(int id) {
		return jdbcTemplate.queryForObject("select * from Users where
		userId=?", new Object[] { id }, new DeveloperRowMapper());
	}
	public Developer create(final Developer developer) {
		final String sql = "insert into Users(userId, Name, Email, address)
					values(?,?,?,?)";
		KeyHolder holder = new GeneratedKeyHolder();
		jdbcTemplate.update(new PreparedStatementCreator() {
		@Override
			public PreparedStatement
	 createPreparedStatement(Connection connection) throws SQLException
{
			PreparedStatement ps =
	connection.prepareStatement(sql, Statement.RETURN_GENERATED_KEYS);
			ps.setInt(1, developer.getId());
			ps.setString(2, developer.getName());
			ps.setString(3, developer.getEmail());
			ps.setString(4, developer.getAddress());
```

```
                        return ps;
                    }
            }, holder);
            int newUserId = holder.getKey().intValue();
            developer.setId(newUserId);
            return developer;
            }
}
```

6. **Create a RowMapper class.** The RowMapper class helps to map a row of the data/relation with the instance of user-defined entity/model class. In the current application, it will help to map each row of the table against the properties defined in the Developer model class.

   Use the following code snippet to create the RowMapper class named DeveloperRowMapper:

```
package com.example.demo.utils;
import java.sql.ResultSet;
import java.sql.SQLException;
import org.springframework.jdbc.core.RowMapper;
import com.example.demo.model.Developer;
public class DeveloperRowMapper implements RowMapper<Developer> {
    @Override
    public Developer mapRow(ResultSet rs, int rowNum) throws SQLException {
                Developer user = new Developer();
                user.setId(rs.getInt("userId"));
                user.setName(rs.getString("Name"));
                user.setEmail(rs.getString("Email"));
                user.setAddress(rs.getString("address"));
                return user;
        }
}
```

7. **Create a REST controller for accessing data.** The controller class acts as an intermediary between the model and the service classes. It calls the methods of the service class and retrieves the data.
   Use the following code to create the REST controller:

```
package com.example.demo.controller;
import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;
import com.example.demo.model.Developer;
import com.example.demo.service.DeveloperService;
@RestController
```

```
public class DeveloperController {
	@Autowired
	private DeveloperService developerService;
	@RequestMapping("/")
	public Developer home(Developer developer) {
		developer = developerService.create(developer);
		return developer;
	}
	@RequestMapping("/developers")
	public List<Developer> findAllUsers() {
		return developerService.findAll();
	}
}
```

8. **Create the main application class.** Finally, the main application class contains the main method that will be called when the application is executed.
   Use the following code snippet to create the main application class named BootdemoApplication.java:

```
package com.example.demo;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.ComponentScan;
@SpringBootApplication
@ComponentScan(basePackages = "com.example.demo")
public class BootdemoApplication {

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

9. **Run Spring Boot with HSql**: To use HSql with Spring Boot, run the application using the option **Run As → Spring Boot App**.

   The HSql database used in the application is in-memory and embedded, therefore, one cannot view its contents directly. One can only view the data in JSON format through code, in the browser, as shown in Figure 7.8.

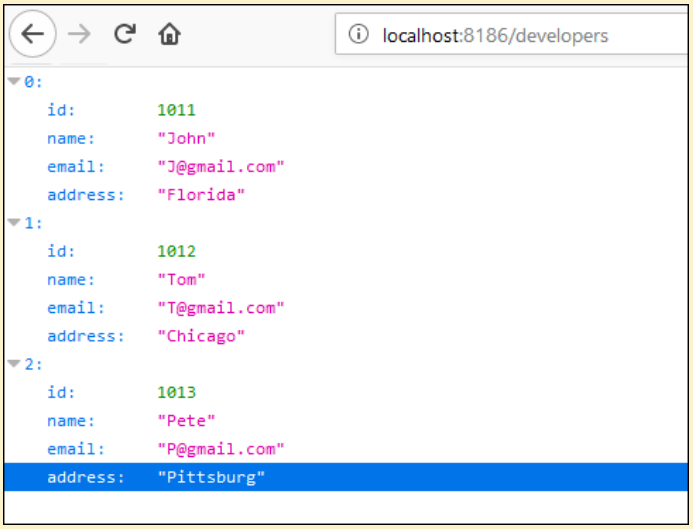
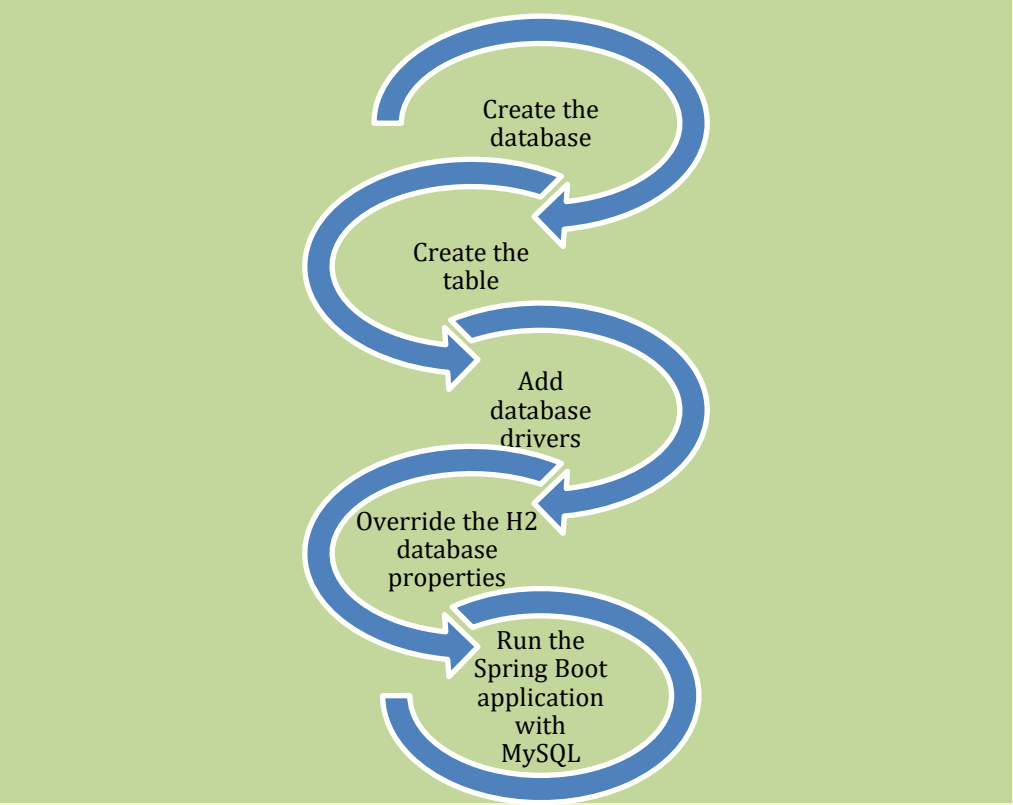


Figure 7.8: Output of BootdemoApplication

As already mentioned, developers can use any other relational database apart from HSql. As an example, one can use MySQL as a database and Java Persistence API (JPA) for Spring Boot applications. JPA helps manage relational data in Spring applications using Java Platform, Standard Edition, and Enterprise Edition.

Steps to configure any relational database in Spring Boot and store data in it are as follows:



These steps are explained in detail as follows:

1. **Create the database**: To use a new MySQL database, developers need to install MySQL and run the following command on the command prompt to login to this database:

   mysql –u root

   Run the following command to create a database:

   ```
   CREATE DATABASE springbootdb;
   ```

2. **Create the table**: After creating a database, developers need to create a table, say 'Users', using the following command:

```
CREATE TABLE Users (
userId INTEGER NOT NULL,
Name VARCHAR(100) NOT NULL,
Email VARCHAR(100) DEFAULT NULL,
address VARCHAR(100) DEFAULT NULL,
        PRIMARY KEY (userId));
```

Create mydata.sql in src/main/resources as follows:

```
INSERT INTO Users (userId, Name, Email, address) VALUES (1011, 'John',
'J@gmail.com', 'Florida');

INSERT INTO Users (userId, Name, Email, address) VALUES (1012, 'Tom',
'T@gmail.com', 'Chicago');

INSERT INTO USERS (userId, Name, Email, address) VALUES (1013, 'Pete',
'P@gmail.com', 'Pittsburg');
```

3. **Verify database drivers**: Make sure that the following code snippet does exit in the POM.xml file because while creating the project, the MySQL option had been selected.

```
<dependency>
<groupId>mysql</groupId>
<artifactId>mysql-connector-java</artifactId>
</dependency>
```

4. **Override the H2 database properties**: As mentioned, Spring Boot provides default support for the H2 database properties, but if developers configure properties for any other database, default properties are overridden. Following code snippet displays how to configure MySQL properties and configure Hibernate for JPA in the src/main/resources/application.properties file:

```
spring.datasource.url= jdbc:mysql://localhost:3306/springbootdb
spring.datasource.username=root
spring.datasource.password=
spring.jpa.hibernate.ddl-auto=create-drop
```

Run Spring Boot with MySQL: Ensure that MySQL is running and the database is existing before executing the application. Run the application and go to the following link to check the result:

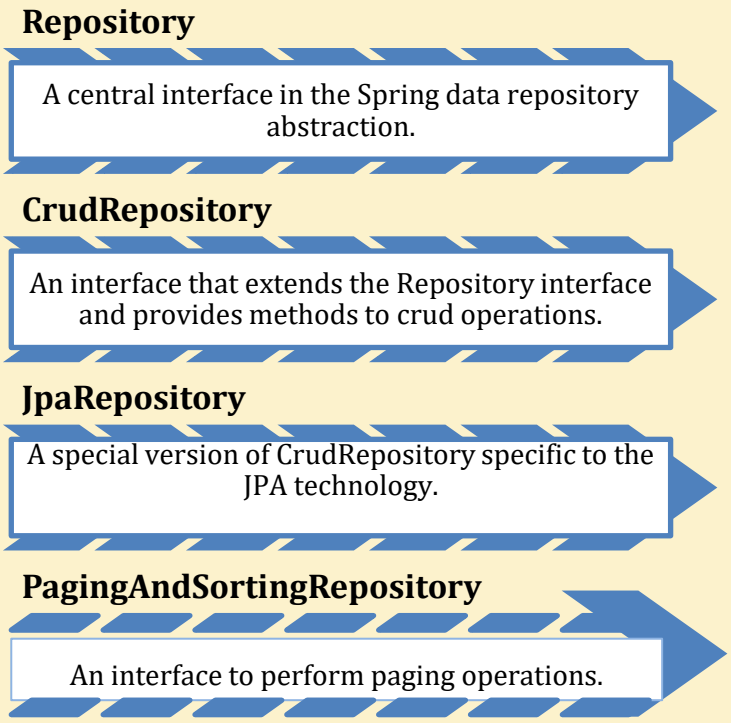
Figure 7.9: Application Output

http://localhost:8080/developers
Figure 7.9 displays the output of the application.

7.4 Data Access with Spring Boot

Spring Boot eliminates the need to write most of the data access operations, such as querying and pagination by generating everything dynamically at run-time. For run-time generation, Spring Boot creates the proxy instances of abstract repositories and performs the required operations. There are several sub-projects that Spring Boot provides, such as Spring Boot JPA, Spring Boot Solr, Spring Boot MongoDB, and Spring Boot REST.

Spring Boot provides abstract repositories, which are implemented by the spring container dynamically to perform the Create Retrieve Update Delete (CRUD) operations. Developers just need to provide abstract methods in any of the following interfaces:



7.5 Securing the App with Spring Boot

Spring Boot security, like Spring Security also supports major security operations, such as authentication and authorization for our Spring Boot Web applications. To introduce security to an application, for example, one can use the following code snippet and add dependency:

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

The spring-boot-starter-security dependency adds HTTP basic security setup for all endpoints. Now, developers need to add the code to design the login page to the application. Here is a code that creates the login page:
(/src/main/resources/templates/login.html)

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml" xmlns:th="http://www.thymeleaf.org"
     xmlns:sec="http://www.thymeleaf.org/thymeleaf-extras-springsecurity3">
<head>
   <title>Spring Security Example</title>
</head>
<body>
<div th:if="${param.error}">
   Invalid username and password.
</div>
<div th:if="${param.logout}">
   You have been logged out.
</div>
<form th:action="@{/login}" method="post">
   <div><label> User Name* : <input type="text" id="username" name="username"/>
</label></div>
   <div><label> Password* : <input type="password" id="password"
name="password"/> </label></div>
   <div><input type="submit" value="Sign In"/></div>
</form>
</body>
</html>
```

Create two configuration related class files as XML based configuration is not being used in this example.

Create Web Configuration file by adding the following code snippet:

```
package com.example.demo.config;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Configuration;
import
org.springframework.security.config.annotation.authentication.builders.Authenticatio
nManagerBuilder;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.config.annotation.web.builders.WebSecurity;
import
org.springframework.security.config.annotation.web.configuration.WebSecurityConfi
gurerAdapter;
import org.springframework.security.web.access.AccessDeniedHandler;
@Configuration
public class SpringSecurityConfig extends WebSecurityConfigurerAdapter {
        @Autowired
```

```
	private AccessDeniedHandler accessDeniedHandler;

	@Override
	protected void configure(HttpSecurity http) throws Exception {
		http.csrf().disable()
				.authorizeRequests()
				.antMatchers("/", "/developers").permitAll()
				.antMatchers("/user/**").hasAnyRole("USER")
				.anyRequest().authenticated()
				.and()
				.formLogin()
				.loginPage("/login.html")
				.permitAll()
				.and()

	.exceptionHandling().accessDeniedHandler(accessDeniedHandler);
	}
	@Autowired
	public void configureGlobal(AuthenticationManagerBuilder auth) throws
Exception {
		auth.inMemoryAuthentication()
	.withUser("developer").password("{noop}developer").roles("USER");
	}
}
```

Add the following denied handler which will redirect for authentication of configured URL, as shown in the following code snippet:

```
package com.example.demo.config;
import java.io.IOException;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.security.access.AccessDeniedException;
import org.springframework.security.core.Authentication;
import org.springframework.security.core.context.SecurityContextHolder;
import org.springframework.security.web.access.AccessDeniedHandler;
import org.springframework.stereotype.Component;
@Component
public class CustomAccessDeniedHandler implements AccessDeniedHandler {
	@Override
	public void handle(HttpServletRequest request, HttpServletResponse response,
AccessDeniedException accessDeniedException) throws IOException,
ServletException {
```

```
        final Logger logger =
                LoggerFactory.getLogger(CustomAccessDeniedHandler.class);
        Authentication authentication =
                SecurityContextHolder.getContext().getAuthentication();
        if (authentication != null) {
                        logger.info(authentication.getName() + "': -> URL: " +
                request.getRequestURI());
        }
        response.sendRedirect(request.getContextPath() + "/403");
        }
}
```

As per the code, the **localhost:8080/test** is restricted to free access, so the application redirects the URL for login page as shown in the Figure 7.10.

Enter 'developer' in both in **User Name** and **Password** text boxes, respectively and once authenticated, an output appears on browser, as shown in Figure 7.11.

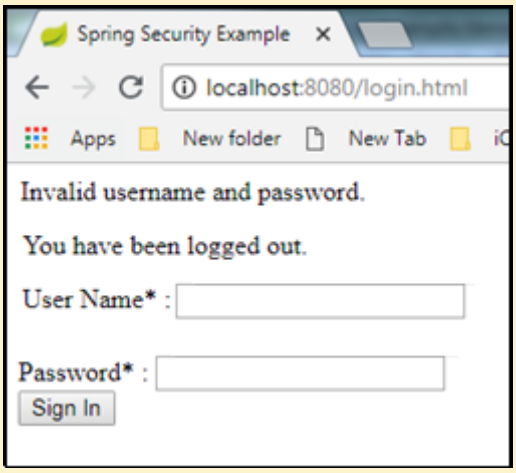


Figure 7.10: Redirected Page

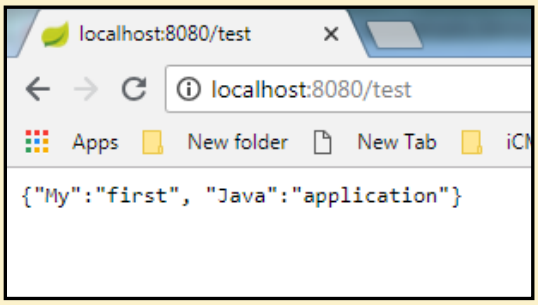


Figure 7.11: Application Output

To use Spring Boot security, developers can also configure their data source. Here is the code snippet that configures the data source:

```
/src/main/resources/application.properties:
spring.datasource.url=jdbc:mysql://localhost:3306/beyond-the-examples
spring.datasource.username=root
spring.datasource.password
spring.datasource.driverClassName=com.mysql.jdbc.Driver
spring.jpa.hibernate.dialect= org.hibernate.dialect.MySQLInnoDBDialect
spring.jpa.generate-ddl=false
```

Summary

- Spring Boot provides pre-defined code and annotation configuration that developers can use to build, package, and deploy new Spring applications with minimal configurations.
- Auto configuration is the process of detecting the availability of certain frameworks, such as Spring Batch, Spring Data JPA, Hibernate, and JDBC, and configuring that framework with defaults.
- Starter projects are a set of convenient dependency descriptors that developers can use in a Spring application.
- Spring Boot has more components, in addition to auto configuration and starter projects, which are, Spring Boot Core, Spring Boot Command Line Interface (CLI), and Spring Boot Actuator.
- Spring Initializr is a Web application that creates a project structure and developers can create their own instance of Spring Initializr by using the Jars as libraries in their application.
- Developers can use Spring Initializr in any of the three ways, namely, Web-based interface, Spring tool suite, and Spring Boot CLI.
- Spring Boot provides abstract repositories, which are implemented by the Spring container dynamically to perform the CRUD operations.

Check Your Progress

1. Which of the following is not a Spring Boot component?

A.	Spring Boot Core	**B.**	Spring Boot CLI
C.	Spring Boot Actuator	**D.**	Spring Boot API

2. Which of the following are components of the Spring Boot equation?

A.	Spring Framework	**B.**	XML Configuration
C.	Integrated Server	**D.**	All of these

3. Auto configuration is the process of detecting the availability of certain _____and configuring that framework with _____.

A.	Batch, groups	**B.**	Frameworks, defaults
C.	Device, application	**D.**	method, application

4. Spring Initializr is a _____ that doesn't generate any application code, but creates a project _____.

A.	Web application, structure	**B.**	method, class
C.	Class, method	**D.**	database, definition

5. Which of the following is not related to data access in Spring Boot?

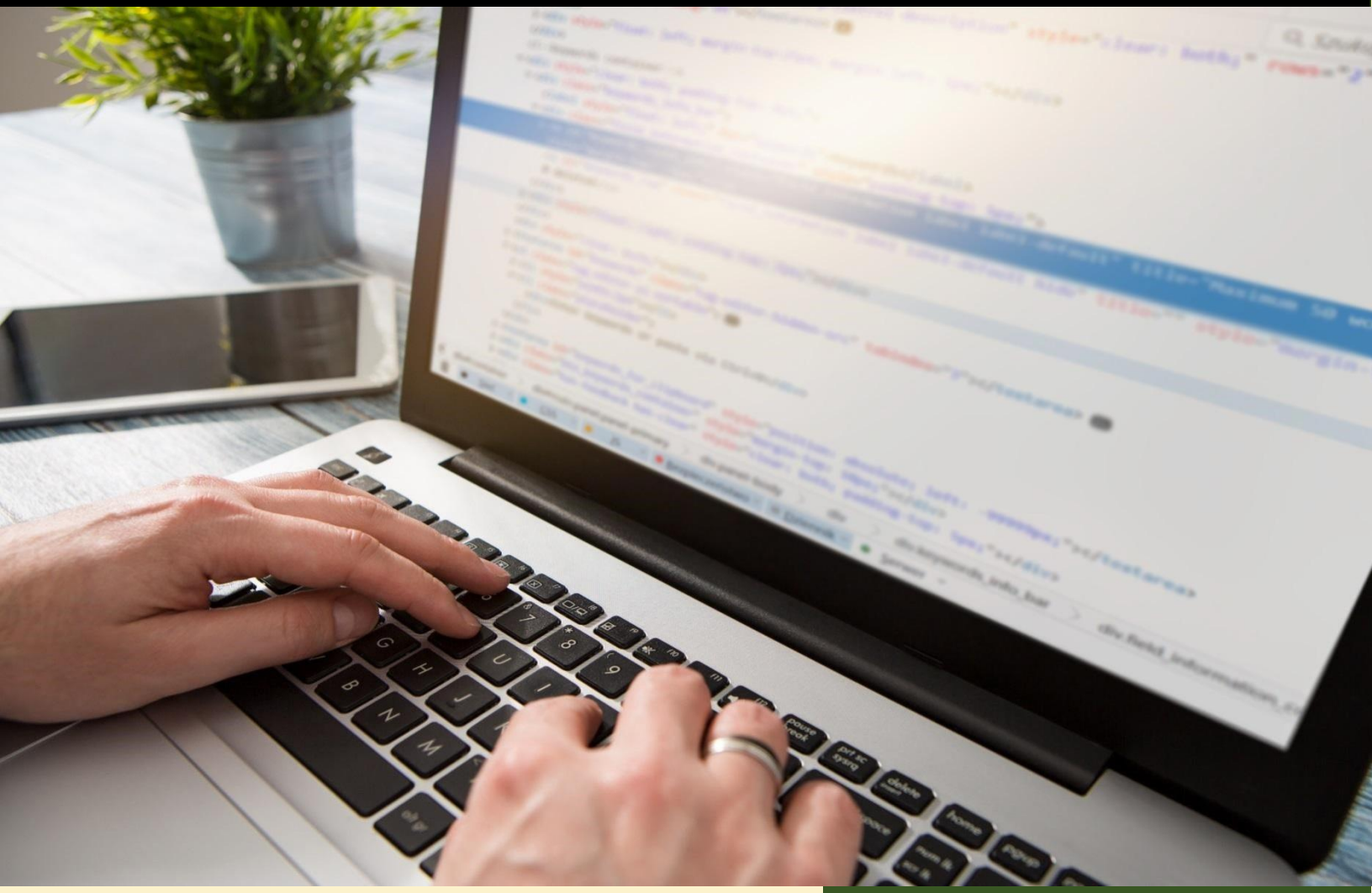
A.	Repository	**B.**	CrudRepository
C.	Validator interface	**D.**	JpaRepository

Answers

1.	D
2.	D
3.	B
4.	A
5.	C

Session 8

Spring Cloud and Spring Microservices



Objectives

Welcome to the session, Spring Cloud and Spring Microservices.

This session introduces the concept of Spring Cloud and Spring Microservices. In addition, this session discusses how to use Spring Cloud and Microservices.

In this session, students will learn to:
- Explain how to use Spring Cloud
- Explain how to use Spring Microservices
- Explain how to use Spring Microservices with Spring Cloud

8.1 What is Spring Cloud?

Spring Cloud is a tool-set that builds on the concepts of Spring Boot and enables developers to quickly build patterns used in distributed systems. A distributed system is a set of autonomous systems, which are part of different networks, but work together to perform a function and appear as a single system to the user. Some common examples of distributed systems are telephone and cellular networks, World-Wide-Web, network file systems, and multiplayer online games.

Spring Cloud provides tools for some of the common components of distributed systems, such as configuration management, distributed sessions, and service discovery. Spring cloud enables the developers to use applications and services to quickly create these patterns, which can be reused with different distributed systems.

8.1.1 Understanding Spring Cloud

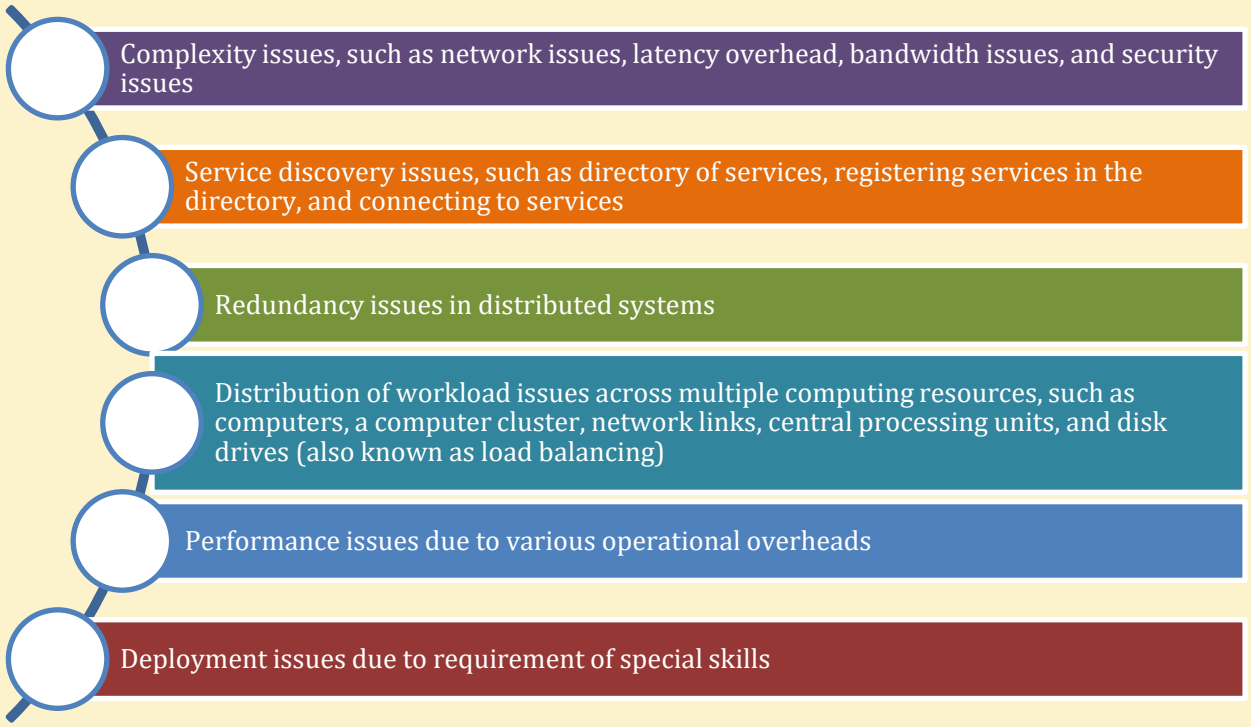
Spring Cloud provides multiple libraries that ease the process of developing, deploying, and operating distributed applications for the cloud. When developers add these libraries to the classpath, application behavior is enhanced and effort to manage hardware, installation, operation, and backups is reduced.

Unlike cloud applications, Spring Cloud applications can easily avail services without the need for code to access and configure service connections. In cloud applications, developers need to create a data source object based on that service. Spring Boot allows to build standalone applications as well as distributed applications, but they can have some issues. On the other hand, Spring Cloud enables us to connect various standalone applications and build a distributed system. Spring Cloud also enables applications to easily connect to services and retrieve information from multiple clouds such as Cloud Foundry which can be extended to other cloud platforms.

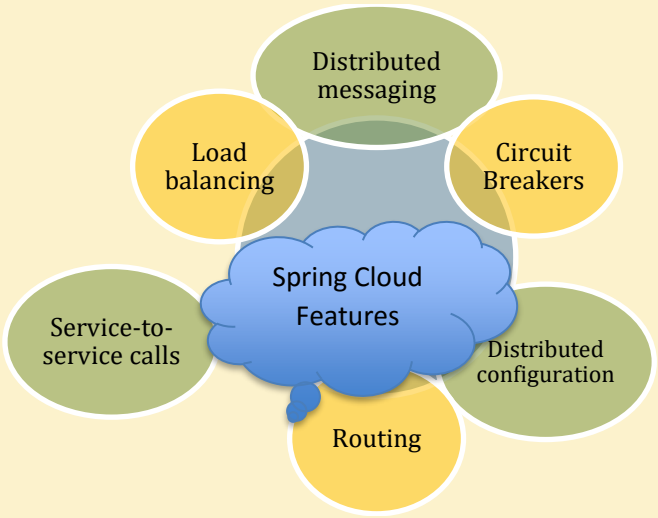
In a nutshell, Spring Cloud can be of assistance in the following ways:
- Spring Cloud enables applications to access cloud services. Consider that a relational database is bound to an application, for which developers will have to create a Data Source Object. This is where Spring Cloud comes in. It eliminates the work needed to access and configure service connectors and lets developers concentrate on the service. It also gives out the application instance information.
- Spring Cloud performs all the work in a cloud independent manner using Cloud Connectors. Thus, allowing full use of libraries.
- Spring Cloud, apart from providing many common services, also allows extension of its functionality to other services.

- Spring Cloud extends support to Spring applications including Spring Boot applications.

Spring Cloud overcomes the following Spring Boot application issues:



Spring Cloud includes following features:



8.1.2 Spring Cloud Releases

Spring Cloud is a project that consists of independent projects with names instead of versions. Each project can have several components called as sub-projects. Versions of the components (sub-projects) are defined in the spring-cloud-starter-parent Bill of Materials (BOM), which in turn defines different versions of its

sub-components. Any confusion with the sub-projects is avoided as the naming of project is used in place of the versions.

Every release name is known as a release train because it may have many other service releases. When there are major changes in any of the projects that need to be published, Spring Cloud team markets a new release with names ending with .SRX, where 'X' is a number. The names of release trains marketed for Spring Cloud are available in chronological order starting from A (alphabetical order). These names are adapted from various London Tube stations.

Spring Cloud releases and the supported Spring Boot versions are shown in Table 8.1.

Release Name	Supported Spring Boot Version
Finchley	2.0.x
Dalston and Edgware	1.5.x
Camden	1.4.x
Brixton	1.3.x and 1.4.x
Angel	1.2.x and 1.3.x

Table 8.1: Spring Cloud Releases

If more than one release supports the same version of Spring Boot, it means that only some of the libraries and applications built on one release will work with another release. For example, Brixton is supported on Spring Boot 1.3.x, but is incompatible with 1.2.x version. Angel is compatible with 1.2.x and 1.3.x; therefore, not all but some libraries are available, and most applications built on Angel run successfully on Brixton.

8.2 Demystifying Microservices

Microservices is architecture as well as an approach that can be used to combine number of smaller components for building up large systems/applications. A component is a Microservice. Microservices are independent from each other, but work together to accomplish a task.

Microservices' architecture-based Web application is very useful where concurrent user varies from one component to another component. In this scenario, Microservices architecture is useful because it allows component-wise scalability instead of application scalability. Online book shop is good example where number of users visiting the site to access book review, read expert comments, and access free chapter is higher than the users visiting the registration page of the online book shop. Similarly, number of visitors to the online shopping module compared to feedback module is higher.

So, the online book shop may have different modules, such as Book Catalog Service, Book Shopping Cart Service, User Feedback Service and so on, as separate Microservices

components. All services can be developed as independent Microservices which will work together to serve the user when he/she visits the online book shop.

Book Catalog services can be deployed in x number of containers while the Book Shopping Cart services could be deployed in y number of containers where y is less than x because every user who reviews a book won't purchase. On the contrary, all the modules of the online book shop can be developed as a single application.

In which case, if the user demand increased, then additional container/clusters would be required to deploy the entire application irrespective of which module receives more user visits.

Application is always bigger than a module and requires more resources to replicate a cluster while Microservices require fewer resources when it is replicated or reused. In Figure 8.1, for example, all microservices are independent entities and are used together to build an Online Utility Services Management System.

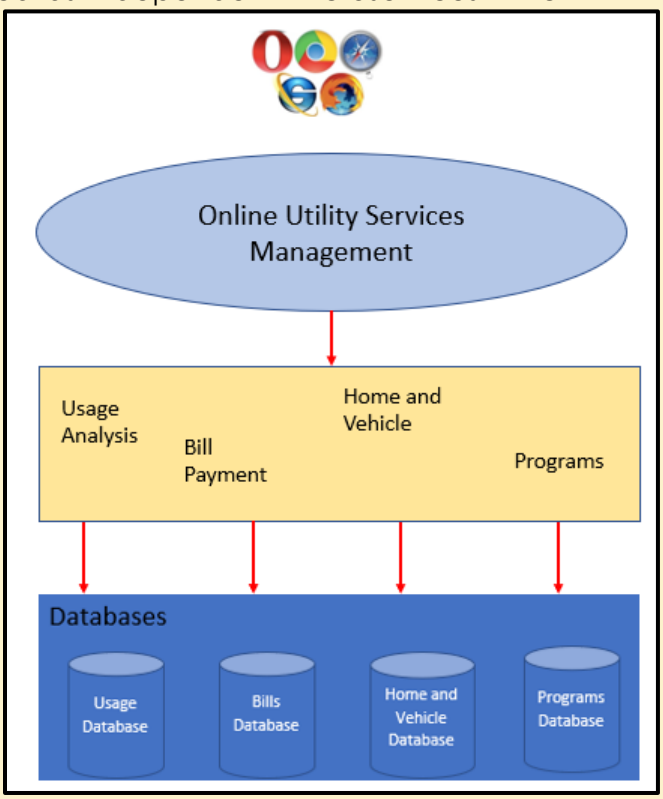


Figure 8.1: Microservices Use in an Application

8.2.1 Principles and Characteristics of Microservices

Microservices are written based on certain principles and they have few characteristics or features of their own.
The principles act as a fundamental concept that guide the behavior of a service. These principles have to be strictly followed while a microservice is designed or developed.
Characteristics of microservices define the distinctive features of the microservice which make it different from other Web services. While creating microservices for an application, developers must consider principles as shown in Figure 8.2. These principles basically define the functional and technical scope of a microservice.

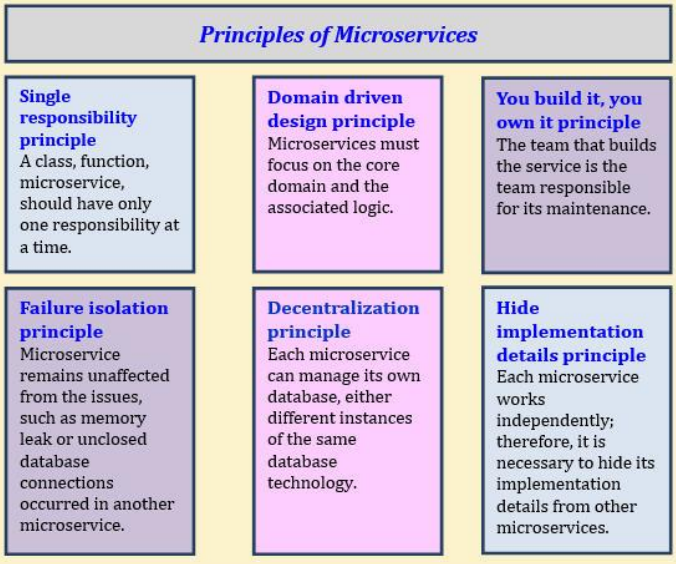


Figure 8.2: Principles of Microservices

Microservices have the following characteristics:

- **Single Responsibility**: A microservice works on a Single Responsibility principle. This means that a microservice will perform a single operation in an application.
- **No Sharing**: A microservice does not share any data or functions with other services. It manages its own data and functions.
- **Data Privacy**: A microservice controls and manages the data storage under it. It has its own data storage mechanism. The service, however, defines the kind of data store it will handle, such as NoSQL database, relational database, or flat files. This encapsulates the data and ensures that it remains consistent. Any changes are handled by the microservice.
- **Private Implementation**: The implementation of a microservice is not done through a shared library, but through a deploy-time dependency. So, if an application needs to execute a specific process, it will call the API exposed by the microservice.
- **Group deployment**: Microservices should be deployed in groups as they work best in groups. It should be developed in such a manner that multiple instances can be deployed. These multiple instances allow the service to continue running even if any one instance malfunctions. This helps in scaling the service horizontally in order to respond to and manage increased demand.

8.2.2 Benefits of Microservices

Microservices provide the following benefits:

- **Generate quality code**: Microservices are small components of a large application, which enables developers to write better codes with more precise results.
- **Localization of complexities**: Microservices are self-contained and independent applications as such the development team of each microservice are concerned with the complexities of their service. This compartmentalization of knowledge and localization of complexity helps in creating and managing large applications.
- **Cross-cutting business service**: Microservices eliminate the need to build several functions to be used multiple times within an application. A single business service can have multiple applications as microservices to improve business functionality.
- **Increased resiliency and scalability**: An application can support multiple microservices and these services are independent of each other, so in case one service fails, it does not affect the entire application. This fault isolation also allows the neighboring services to continue functioning and the user gets uninterrupted experience. Similarly, microservice assists in quickly finding out the bottlenecks in an application and also allows the scaling of individual microservices to resolve the bottlenecks, thus, providing better user experience.
- **Real-time data processing and customized output**: The publish-subscribe framework of the microservice architecture allows processing of data in real-time and also delivers direct output and insights. In addition, microservices enable developers to customize the presentation of data for different audiences.
- **Use of different technologies**: Microservices are smaller functions of a large application. This allows the use of the best programming language and technology

for developing these functions. There is no limitation to use only one technology for the whole application. In addition, microservices are independent applications, thus allowing developers to experiment with new technologies and programming languages on individual components.

- **Application reuse and protected outsourcing**: Microservices are modular in nature, thus allowing reuse of applications and smooth and faster deployment of additional applications for better business value. In addition, businesses which outsource their work can protect their intellectual property by using microservices. It allows the business to outsource only their non-core functions without disclosing their core functions and services.

8.2.3 Building Microservices with Spring Boot

There are various frameworks that support Spring to develop Microservices based application.

HATEOAS Enabled Spring Boot Microservice

HATEOAS stands for Hypermedia as the Engine of Application State. It is used to give extra information about a REST API and is considered as an extra level upon REST. A HATEOAS-driven site provides information to facilitate better navigation of the site's REST interfaces by providing hypermedia links with responses. These hypermedia links are further used to communicate with the servers. The latest version can be downloaded from the corresponding Maven repository link. For example,

https://mvnrepository.com/artifact/org.springframework.hateoas/spring-hateoas/0.25.0.RELEASE

Steps to create HATEOAS Enabled Spring Boot Microservice are as follows:

1. Create a new Maven based Spring Boot application through the wizard.
2. Add the dependencies as shown in following code snippet, in the pom.xml file:

```
<!-- Microservices related dependencies -->
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-hateoas</artifactId>
</dependency>
```

This specifies the HATEOAS dependencies.

3. Create a Customer object with the following code snippet:

```
package com.session.eight.model;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.Table;
@Entity
```

```
@Table(name = "customer")
public class Customer {
	@Id
	@Column
	private Integer id;

	@Column
	private String name;

	public String getName() {
		return name;
	}
	public void setName(String name) {
		this.name = name;
	}
	public Integer getId() {
		return id;
	}
	public void setId(Integer id) {
		this.id = id;
	}
}
```

Through this code, a Customer entity is created, with attributes name and id.

4. Create a Customer Repository which executes CRUD operation on database with the code shown in the following code snippet:

```
package com.session.eight.repository;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.Query;
import org.springframework.stereotype.Repository;
import com.session.eight.model.Customer;

@Repository
public interface CustomerRepository extends JpaRepository<Customer, Integer> {
	@Query("SELECT c FROM Customer c WHERE c.id = ?1 ")
	Customer findBy(Integer id);
}
```

5. Create a REST Controller class to handle REST request as shown in the following code snippet:

```
package com.session.eight.controller;
import java.util.List;
import java.util.stream.Collectors;
import org.springframework.beans.factory.annotation.Autowired;
```

```
import org.springframework.hateoas.Link;
import org.springframework.hateoas.Resources;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;
import org.springframework.web.servlet.support.ServletUriComponentsBuilder;

import com.session.eight.model.Customer;
import com.session.eight.repository.CustomerRepository;
import com.session.eight.utility.CustomerResource;

@RestController
@RequestMapping(value = "/customer", produces = "application/hal+json")
public class CustomerController {

	@Autowired
	private CustomerRepositorycustomerRepository;

	@GetMapping
	public ResponseEntity<Resources<CustomerResource>>all() {
		List<CustomerResource>customerResources
=customerRepository.findAll().stream().map(CustomerResource::new).collect(Collect
ors.toList());
		Resources<CustomerResource> resources = new
Resources<>(customerResources);
		final String uriString =
ServletUriComponentsBuilder.fromCurrentRequest().build().toUriString();
		resources.add(new Link(uriString, "self"));
		return ResponseEntity.ok(resources);

	}

	@GetMapping("/{id}")
	public ResponseEntity<Customer>get(@PathVariable Integer id) {
		Customer customer = customerRepository.findBy(id);
		return new ResponseEntity(customer, HttpStatus.OK);
	}
}
```

The controller class maps the requests received with the action specified in the class body. It implements a basic REST service with each endpoint returning a ResponseEntity. The ResponseEntity contains one or more CustomerResource objects. Here is where a HATEOAS service can be distinguished from standard REST service.

A standard REST service would have returned ResponseEntity containing Customer instead of CustomerResource object. In the current application, the code returns ResponseEntity<CustomerResource> (HATEOAS) instead of ResponseEntity<Customer> (REST).

The HATEOAS framework is being used in this controller class, as shown in the following code snippet:

```
import org.springframework.hateoas.Link;
import org.springframework.hateoas.Resources;
```

6. A utility class, CustomerResource, is required to generate desired output. Create the CustomerResource utility class using the code shown in the following code snippet:

```
package com.session.eight.utility;
import org.springframework.hateoas.ResourceSupport;
import static org.springframework.hateoas.mvc.ControllerLinkBuilder.linkTo;
import static org.springframework.hateoas.mvc.ControllerLinkBuilder.methodOn;
import com.session.eight.controller.CustomerController;
import com.session.eight.model.Customer;

public class CustomerResource extends ResourceSupport {
        private final Customer customer;
        public CustomerResource(final Customer customer) {
                this.customer = customer;
           Integer id = customer.getId();
           add(linkTo(CustomerController.class).withRel("customer"));
           add(linkTo(methodOn(CustomerController.class).get(id)).withSelfRel());
        }
        public Customer getCustomer() {
                return customer;
        }
}
```

The HATEOAS REST service will return output in JSON format. To understand this better, consider an example. Given a data record as follows:

Name	Alison Jones
Id	212

It can be rendered in basic JSON format as follows:

{ "name" : "Alison Jones", "id":212 }

In the same manner, Customer data in the output may look as follows:

{"id":1,"name":"James Morgan"}, {"id":2,"name":"Elizabeth Howard"}

Finally, the last step will be to execute the application.

7. Ensure that MySQL is running and a database with the mentioned table is created with at least two rows. Run the application as a Spring Boot App and open the following URL in the Address bar of the Web browser:

   http://localhost:8080/customer

The output is displayed as shown in Figure 8.3. This HATEOAS based response has the customer's name, as well as a self-linking URL to locate the customer.

- **rel** stands for relationship. In this example, it is a self-referencing hyperlink. In complex system rel can include other relationships.
- **href** gives the URL that defines the resource.

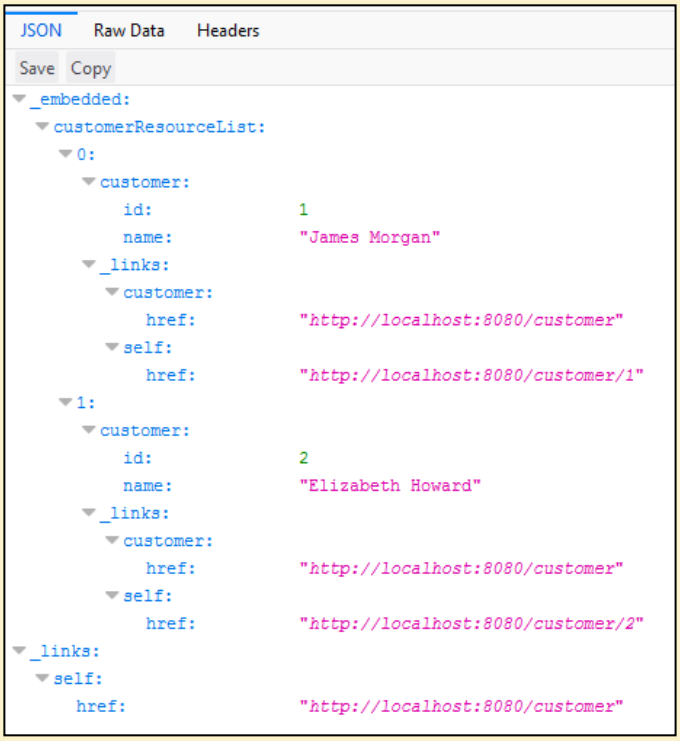


Figure 8.3: Output of HATEOAS Example

8.2.4 Reactive Spring Boot Microservice

Reactive programming style is generally event-driven and asynchronous. In addition, the most important technique in reactive programming is that it supports non-blocking executions of threads. Apart for this, they need only small number of threads to scale vertically within JVM rather than scaling horizontally through clustering.

The API for reactive programming libraries is of the following types:

- **Callback Based**: In this type, the callbacks are attached to an event source, which are invoked when the events pass through dataflow chain.
- **Declarative**: This library uses functional composition using combinations, such as filters, map, fold, and so on.

Reactive programming can be implemented in various use cases such as:

- **External service calls**: Today, majority of backend services use the REST Web services, because IO processes block operations. This creates waiting time for one call to complete one request before processing the next request.
- **Highly concurrent messaging**: Reactive programming and patterns are highly suitable for message processing.

Reactive Spring is based on project reactor which uses Spring platform and Spring cloud framework for developing non-blocking applications. The reactor has following three important interfaces:

- **Publisher**: This is the source of data
- **Subscriber**: This receives data asynchronously
- **Processor**: This is the publisher

The reactive core is completely supported by Spring 5 or later and Spring Boot 2 or later versions.

8.2.5 Implementing Security

When using microservice architecture for an application, security is an important aspect. The architecture of microservices is by nature highly distributed and is characterized by high network traffic. The business data can be accessed through REST APIs, which can be stolen if it is not adequately protected. These data thefts can be avoided by using HTTPS protocol for which the API endpoints should also be well secured.

The security measures should be implemented before a microservice API is called otherwise the service will not perform well and also expose the business data. In order to secure the endpoint, it is recommended that the microservice should have an identity and set of permissions attached to it. This can be verified by the API endpoint security component. Spring offers interesting features and frameworks for securing microservices; one of the commonly used is OAuth2.0.

8.2.6 Enabling Cross Origin for Microservices Interaction

The most important security concept of Web browsers is the same-origin policy, which prevents the JavaScript code from making different origin requests other than the domain being served. This same-origin policy is effective in stopping resources from different origins but at the same time also prevents authentic server and client interaction from known and trusted origins.

Therefore, to relax the same-origin policy, the Cross-Origin Resource Sharing (CORS) technique is deployed. CORS allows a JavaScript code on a Web page to access a REST API called from a different domain/origin.

When CORS is enabled, it first performs a security check whenever the browser requests a resource from a different origin. For example, if developers have an application on domainX.com and they request a microservice from domainY.com, a CORS check will be done. Spring Boot provides a simple approach to enable cross-origin requests.

Following code snippet shows how to enable a microservice to enable cross-origin requests:

```
@RestController
class GreetingController{
  @CrossOrigin
  @RequestMapping("/")
  Greet greet() {
    return new Greet("Hello World!");
  }
}
```

8.3 Scaling Microservices with Spring Cloud Using Spring Cloud Config

Spring Cloud Config is a Spring Boot application which provides server-side and client-side support for externalized configuration, primarily in a distributed system. The Cloud Config Server provides a central location to manage all the external properties of an application, across all environments. It works well for Spring applications because the core concept of the server and client-side are identical to Spring Environment and PropertySource abstractions. Therefore, when the application moves through the deployment process from development to testing phase, the configuration can be managed between these development environments. This makes the application ready to use and migrate into Spring configuration.

Some of the Spring Cloud Config Server features are as follows:

- Spring Cloud Config Server can be easily embedded in a Spring Boot Application using @EnableConfigServer.
- Spring Cloud Config Server can encrypt and decrypt property values, both symmetrically and asymmetrically.
- Spring Cloud Config Server uses HTTP and resource-based API for external configurations.

In addition, Config Client has the following features for Spring applications:

- Binding the Config Server and initializing the Spring Environment with remote property sources
- Encrypting and decrypting property values

Configuring the Spring Cloud Config Server

A Spring Cloud Config Server can be configured in following two ways:

- **Local File System**: In this method, the properties which have to be externalized are stored in the local file system of the Spring Cloud Config Server.
- **GIT Repo**: In this method, the properties which have to be externalized are stored in the GIT Repo.

Enabling Cloud Config Server

Spring Cloud's @EnableConfigServer method is used to create a Config Server as shown in the following code snippet:

```
@EnableConfigServer
@SpringBootApplication
public class ConfigServiceApplication {
   public static void main(String[] args) {
SpringApplication.run(ConfigServiceApplication.class, args);
   }
}
```

8.3.1 Feign as Declarative REST Client

Spring Cloud Feign is basically a Spring Cloud Netflix library, which makes it easier to configure Web service clients. Feign works on a declarative principle or in other words is a declarative Web service client. Feign is deployed by creating an interface and then annotating it. First declarative REST service interfaces are written at the client side and then these interfaces are used to program the client. The advantage of using Feign is that developers do not have to worry about the implementation of this interface, as it is dynamically done by Spring at runtime. Feign, therefore as a client, works as an important tool for communicating with other microservices through REST API. Developers can use the following methods to include Feign in their project:

- Use the starter with group org.springframework.cloud
- Use artifact id spring-cloud-starter-netflix

Create a Maven based Spring Boot project (say session-8-feign) and perform following steps to create its components.

Let's see an example in which a Training Service will be customized to make it Feign-enabled, as shown in Figure 8.4.

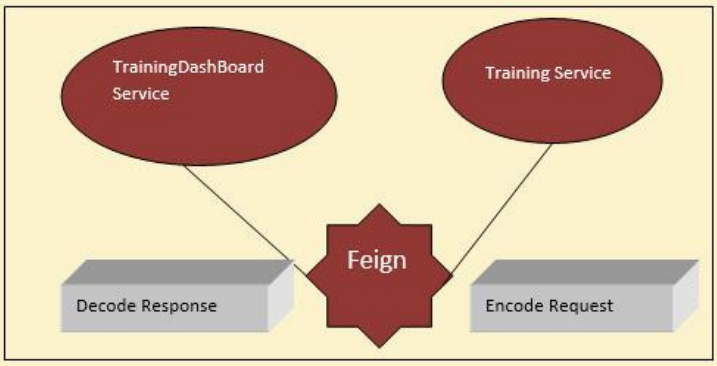


Figure 8.4: Including Feign in a Project

Feign framework creates a wrapper on top of REST based HTTP request so that developer/tester do not need to know about REST. Feign client can be used to consume REST services either in any consumer application or perform testing of the REST services.

Step 1: Add the feign dependency into pom.xml of the application (say session-8-feign), as shown in the following code snippet:

```
<dependency>
	<groupId>org.springframework.cloud</groupId>
	<artifactId>spring-cloud-starter-netflix</artifactId>
</dependency>
```

Step 2: In this step, we will create an interface to declare the service which we want to call. It is important to note that, Training Service Rest URL is same as Service Request mapping. Feign will call this URL when we call the Training service.

Feign will dynamically generate the implementation of the interface we have created; this interface has to be assigned a name which will become the {Service-Id} of TrainingService. All these will inform Feign which service to call beforehand.

Eureka is a server used to locate services for balancing load and making sure that all service requests are directed to available instances. Now, Feign will contact the Eureka server with the Service Id, then will resolve the actual hostname/IP of the TrainingService, and will call the URL given in Request Mapping. It is important to note that whenever developers use @PathVariable for a Feign Client, always put value property else it will show this error –
java.lang.IllegalStateException:PathVariable annotation was empty on param 0

```
package com.session.eight.controller;
import java.util.Collection;
import org.springframework.cloud.netflix.feign.FeignClient;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import com.session.eight.model.Trainer;
@FeignClient(value="session-8-service")
public interface TrainingServiceProxy {
	@RequestMapping("/trainer/find/{id}")
	public Trainer findById(@PathVariable(value = "id") Integer id);
	@RequestMapping("/trainer/findall")
	public Collection<Trainer> findAll();
}
```

Step 3: This step involves creating FeignTrainingController where the interface is autowired, so that Spring can inject actual implementation during runtime, as shown in following code snippet. Then, that implementation is invoked to call the TrainingService REST API.

```
package com.session.eight.controller;
import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.cloud.context.config.annotation.RefreshScope;
import org.springframework.web.bind.annotation.PathVariable;
```

```
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;
import com.session.eight.model.Trainer;
@RefreshScope
@RestController
public class FeignTrainingController {
	@Autowired
	TrainingServiceProxy serviceProxy;
	@RequestMapping("/dashboard/feign/{myself}")
	public Trainer findme(@PathVariable Long myself) {
		return serviceProxy.findById(myself);
	}
	@RequestMapping("/dashboard/feign/peers")
	public List<Trainer> findPeers() {
		return serviceProxy.findAll();
	}
}
```

Step 4: In this step, we need to inform our project that we will be using Feign client, so scan its annotation. For this, we need to add the @EnableFeignClients annotation on top of the application class as shown in the following code snippet:

```
package com.session.eight;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.boot.web.client.RestTemplateBuilder;
import org.springframework.cloud.client.circuitbreaker.EnableCircuitBreaker;
import org.springframework.cloud.netflix.feign.EnableFeignClients;
import org.springframework.context.annotation.Bean;
import org.springframework.web.client.RestTemplate;
@EnableFeignClients
@SpringBootApplication
public class Session8FeignApplication {
	public static void main(String[] args) {
		SpringApplication.run(Session8FeignApplication.class, args);
	}
	@Bean
	public RestTemplate restTemplate(RestTemplateBuilder builder) {
		return builder.build();
	}
}
```

Step 5: Create a Trainer model as follows:

```
package com.session.eight.model;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.Table;
@Entity
@Table(name = "trainer")
public class Trainer {
	@Id
	@Column
	private Integer id;
	@Column
	private String name;
	@Column
	private String designation;
	@Column
	private String domain;
	@Column
	private String organization;
	public Integer getId() {
		return id;
	}
	public void setId(Integer id) {
		this.id = id;
	}
	public String getName() {
		return name;
	}
	public void setName(String name) {
		this.name = name;
	}
	public String getDesignation() {
		return designation;
	}
	public void setDesignation(String designation) {
		this.designation = designation;
	}
	public String getDomain() {
		return domain;
	}
	public void setDomain(String domain) {
		this.domain = domain;
	}
	public String getOrganization() {
		return organization;
```

```
        }
        public void setOrganization(String organization) {
                this.organization = organization;
        }
}
```

After this step, developers can use the Feign client. However, the Eureka server and client need to be configured before executing this application.

8.3.2 Eureka for Registration and Discovery

Eureka or Eureka Server is a REST-based, AWS Discovery Service. It is used in AWS cloud and includes a server and a client. Its main purpose is to locate services for balancing load and making sure that all service requests are directed to available instances.

Eureka includes Eureka Client, which is a Java based client component.

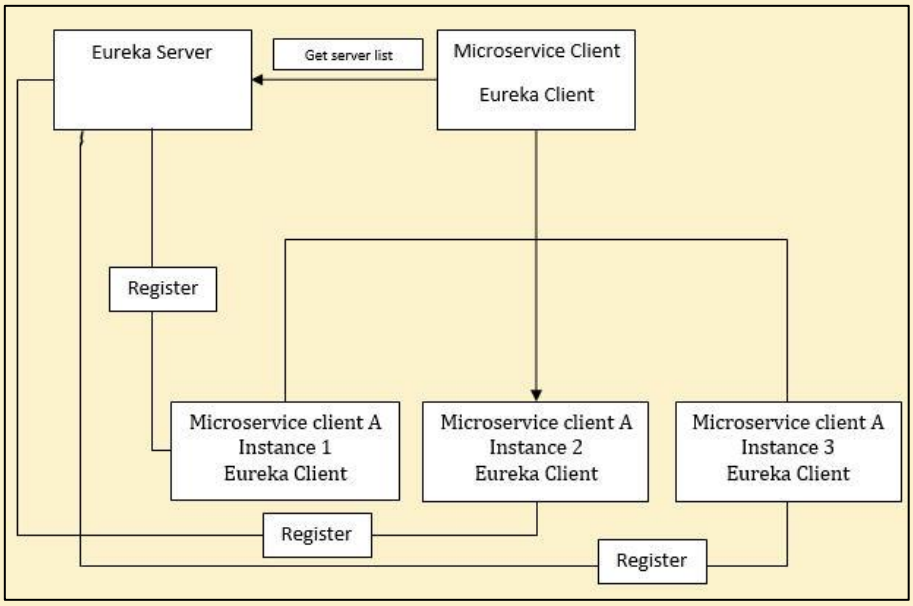


Figure 8.5: Workflow of Eureka

Eureka Client is basically a bundle of Java libraries, framework, software, such as load balancer and is used for interfacing with the Eureka Server. It also has an in-built load balancer application that balances the number of incoming service requests using the basic round-robin technique. This client ensures easier and quick interaction with the services. The Eureka service workflow is shown in Figure 8.5.

Figure 8.5 displays how Eureka client is deployed on three instances (Instance 1, Instance 2, and Instance 3) or container and each client (instance of client) registers itself to the server which helps discoverer to identify server as well as route requests as per load balancing rule.

Consider an example.

Step 1: Create a Maven based Spring Boot application project. (For example, session-8-eureka)

Step 2: Modify pom.xml file by following snippet:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
	xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
	<modelVersion>4.0.0</modelVersion>

	<groupId>com.example.demo</groupId>
	<artifactId>EurekaServerTest</artifactId>
	<version>0.0.1-SNAPSHOT</version>
	<packaging>jar</packaging>
	<name>session-8-eureka</name>
	<description>Demo project for Spring Boot</description>
	<parent>
		<groupId>org.springframework.boot</groupId>
		<artifactId>spring-boot-starter-parent</artifactId>
		<version>2.0.4.RELEASE</version>
		<relativePath/> <!-- lookup parent from repository -->
	</parent>
	<properties>
		<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
		<project.reporting.outputEncoding>UTF-
8</project.reporting.outputEncoding>
		<java.version>1.8</java.version>
		<spring-cloud.version>Finchley.SR1</spring-cloud.version>
	</properties>
	<dependencies>
		<dependency>
			<groupId>org.springframework.cloud</groupId>
			<artifactId>spring-cloud-starter-netflix-eureka-server</artifactId>
		</dependency>

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

```
                        <dependency>
                                <groupId>org.springframework.cloud</groupId>
                                <artifactId>spring-cloud-dependencies</artifactId>
                                <version>${spring-cloud.version}</version>
                                <type>pom</type>
                                <scope>import</scope>
                        </dependency>
                </dependencies>
        </dependencyManagement>
        <build>
                <plugins>
                        <plugin>
                                <groupId>org.springframework.boot</groupId>
                                <artifactId>spring-boot-maven-plugin</artifactId>
                        </plugin>
                </plugins>
        </build>
</project>
```

Note: Based on actual library downloads of Spring boot and cloud releases, the version numbering may change in the dependencies in pom.xml. This applies to all the pom.xml files for this entire application.

Step 3: Modify main Application Java file (in this project, name would be Session8EurekaApplication.java) by following snippet:

```
package com.session.eight;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.netflix.eureka.server.EnableEurekaServer;
@EnableEurekaServer
@SpringBootApplication
public class Session8EurekaApplication {
  public static void main(String[] args) {
                SpringApplication.run(Session8EurekaApplication.class, args);
        }
}
```

Step 4: Modify application.properties file by following snippet:

```
server.port=8761
# ${PORT:8761} # Indicate the default PORT where this service will be started
spring.application.name=session-8-eureka
eureka.instance.hostname=localhost
eureka.client.registerWithEureka:false
# telling the server not to register itself in the service
eureka.client.fetchRegistry=false
```

```
eureka.client.serviceUrl.defaultZone=http://${eureka.instance.hostname}:${server.port
}/eureka
spring.jpa.hibernate.ddl-auto=validate
spring.jpa.generate-ddl=false
spring.main.allow-bean-definition-overriding=true
```

Now, the server can be used with the Training project. The steps are as follows:

Step1: Create a Maven based Spring Boot project named 'session-8-service'.

Step 2: Modify pom.xml by following snippet:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.session</groupId>
  <artifactId>service</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <packaging>jar</packaging>
  <name>session-8-service</name>
  <description>Demo project for Spring Boot</description>
  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.1.0.BUILD-SNAPSHOT</version>
    <relativePath /> <!-- lookup parent from repository -->
  </parent>
  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
    <java.version>1.8</java.version>
    <spring-cloud.version>Dalston.SR1</spring-cloud.version>
  </properties>
  <dependencies>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-actuator</artifactId>
    </dependency>
    <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-starter-config</artifactId>
    </dependency>
    <dependency>
      <groupId>org.springframework.boot</groupId>
```

```
    <artifactId>spring-boot-starter-jersey</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-eureka</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
  </dependency>
</dependencies>
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-dependencies</artifactId>
      <version>${spring-cloud.version}</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
<build>
  <plugins>
    <plugin>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId>
    </plugin>
  </plugins>
</build>
<repositories>
  <repository>
    <id>spring-snapshots</id>
    <name>Spring Snapshots</name>
    <url>https://repo.spring.io/snapshot</url>
    <snapshots>
      <enabled>true</enabled>
    </snapshots>
  </repository>
  <repository>
    <id>spring-milestones</id>
    <name>Spring Milestones</name>
```

```
    <url>https://repo.spring.io/milestone</url>
    <snapshots>
      <enabled>false</enabled>
    </snapshots>
  </repository>
 </repositories>
 <pluginRepositories>
  <pluginRepository>
    <id>spring-snapshots</id>
    <name>Spring Snapshots</name>
    <url>https://repo.spring.io/snapshot</url>
    <snapshots>
      <enabled>true</enabled>
    </snapshots>
  </pluginRepository>
  <pluginRepository>
    <id>spring-milestones</id>
    <name>Spring Milestones</name>
    <url>https://repo.spring.io/milestone</url>
    <snapshots>
      <enabled>false</enabled>
    </snapshots>
  </pluginRepository>
 </pluginRepositories>
</project>
```

Step 3: Modify main Java Application file (in this case name would be 'Session8ServiceApplication.java') by following snippet:

```
package com.session.eight;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.client.discovery.EnableDiscoveryClient;
@SpringBootApplication
@EnableDiscoveryClient
public class Session8ServiceApplication {
        public static void main(String[] args) {
                SpringApplication.run(Session8ServiceApplication.class, args);
        }
}
```

The @EnableDiscoveryClient annotation looks for the implementations of a Discovery Service. It is defined in spring-cloud-commons.

Note: Now on, it is not necessary to specify @EnableDiscoveryClient. Adding a DiscoveryClient implementation on the classpath is sufficient to inform the Spring Boot application to register with the service discovery server.

Step 4: Create a domain class for Trainer entity using following code:

```
package com.session.eight.model;
public class Trainer {
	private Integer id;
	private String name;
	private String designation;
	private String domain;
	private String organization;
	public Integer getId() {
		return id;
	}
	public void setId(Integer id) {
		this.id = id;
	}
	public String getName() {
		return name;
	}
	public void setName(String name) {
		this.name = name;
	}
	public String getDesignation() {
		return designation;
	}
	public void setDesignation(String designation) {
		this.designation = designation;
	}
	public String getDomain() {
		return domain;
	}
	public void setDomain(String domain) {
		this.domain = domain;
	}
	public String getOrganization() {
		return organization;
	}
	public void setOrganization(String organization) {
		this.organization = organization;
	}
}
```

Step 5: Create a service class which will serve as business logic using following code:

```
package com.session.eight.service;
import java.util.Collection;
import java.util.Map;
```

```
import java.util.stream.Collectors;
import java.util.stream.Stream;
import org.springframework.stereotype.Service;
import com.session.eight.model.Trainer;
@Service
public class TrainerService {
	private static Map<Integer, Trainer> repository = null;
	static {
		Stream<String> stream = Stream.of("1, James Morgan, Java,Junior
Trainer, Aptech", "2,Elizabeth Howard, Database, Senior Trainer, Aptech");

		repository = stream.map(str -> {
			String[] info = str.split(",");
			return create(new Integer(info[0]), info[1], info[2], info[3], info[4]);
		}).collect(Collectors.toMap(Trainer::getId, trainer -> trainer));
	}
	private static Trainer create(Integer id, String name, String practiceArea, String
designation, String organization) {
		Trainer trainer = new Trainer();
		trainer.setId(id);
		trainer.setName(name);
		trainer.setDomain(practiceArea);
		trainer.setDesignation(designation);
		trainer.setOrganization(organization);
		return trainer;
	}
	public Trainer findById(Integer id) {
		return repository.get(id);
	}
	public Collection<Trainer> findAll() {
		return repository.values();
	}
}
```

Step 6: Create a controller class named TrainerController using following code:

```
package com.session.eight.controller;
import java.util.Collection;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.cloud.context.config.annotation.RefreshScope;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;
import com.session.eight.model.Trainer;
import com.session.eight.service.TrainerService;
@RefreshScope
@RestController
```

```
public class TrainerController {
	@Autowired
	TrainerService trainerService;
	@RequestMapping("/trainer/find/{id}")
	public Trainer findById(@PathVariable Integer id) {
		System.out.println("Service->TrainerController::findById");
		return trainerService.findById(id);
	}
	@RequestMapping("/trainer/findall")
	public Collection<Trainer> findAll() {
		System.out.println("Service->TrainerController::findAll");
		return trainerService.findAll();
	}
}
```

This class maps various requests against the respective method. Then, create a HomeController class which will handle default request using following code:

```
package com.session.eight.controller;
import org.springframework.web.bind.annotation.CrossOrigin;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;
@RestController
public class HomeController {
	@CrossOrigin
	@RequestMapping("/")
	String greet() {
		return "Hello World!";
	}
}
```

Finally, the Feign client created earlier can be tested. All the microservices created so far have to be started, in the following order (each as a Spring Boot Application):

1. session-8-configserver
2. session-8-eureka
3. session-8-service
4. session-8-feign

Then, type the following URL in the Address bar of the Web browser and press Enter:
http://localhost:8091/dashboard/feign/peers

The output is displayed as shown in Figure 8.6.

Figure 8.6: Output of the Feign Client Showing All Records

If the URL http://localhost:8091/dashboard/feign/1 is typed in the Address bar, the output will be as shown in Figure 8.7.

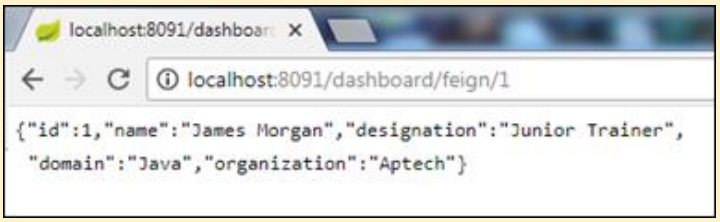


Figure 8.7: Output of the Feign Client Showing Single Record

8.3.3 Zuul Proxy for the API Gateway

The basic purpose of microservices design pattern is to develop an independent service which can be deployed and scaled independently. In a real-world scenario, a complex business domain can include multiple microservices. For example, a particular business has 50 microservices for which a User Interface (UI) has to be implemented. This UI will help to manage these services, and display information about them.

So, from the developer's perspective, to collect information of all 50 microservices, all fifty REST APIs will have to be called. Consequently, the client has to know about all the REST APIs and URLs in order to call them. This process increases the operational overhead and is not recommended for implementation. An alternative mechanism can be to have a UI that includes all the information about the microservices ports/server details to fulfill the services. This alternative mechanism is provided by Zuul. Zuul is an edge service or a simple gateway service that manages all incoming requests. In cases, where the UI needs to be enabled to receive all incoming requests and to delegate these requests to internal microservices, a new set of Zuul-enabled microservices have to be created which sits on top all other microservices. It acts as a client-facing service or edge service, and also a proxy service for the internal microservices. The service API of Zuul-enabled microservices should be exposed to the UI/client. The client calls this service as a proxy for the actual internal microservices, this service then delegates the request to the relevant service.

Zuul Components

Zuul consists of four types of filters. These filters allow interception of traffic at different times of request processing for a particular transaction. Also, any number of filters can be added for a particular URL pattern. These filters are:

- **Prefilters**: These filters are invoked before the request is routed.
- **Post filters**: These filters are invoked after the request has been routed to the relevant microservice.

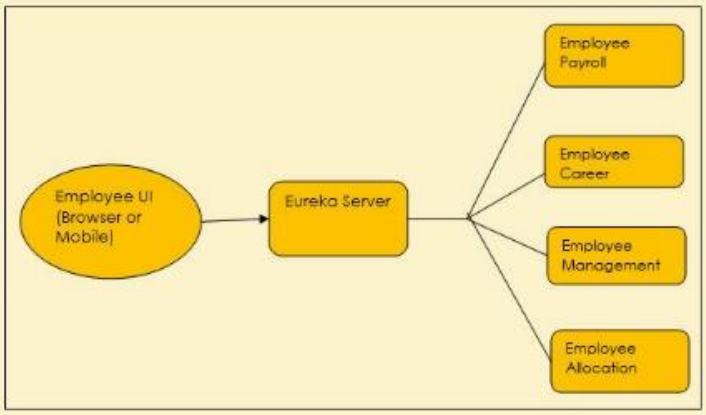


Figure 8.8: Non-Zuul Enabled Service

- **Route filters**: These filters are used to route a request to an appropriate microservice.
- **Error filters**: These filters are invoked when an error occurs during request handling.

Figures 8.8 and 8.9 illustrate the difference between Zuul-enabled and Non-Zuul enabled services.

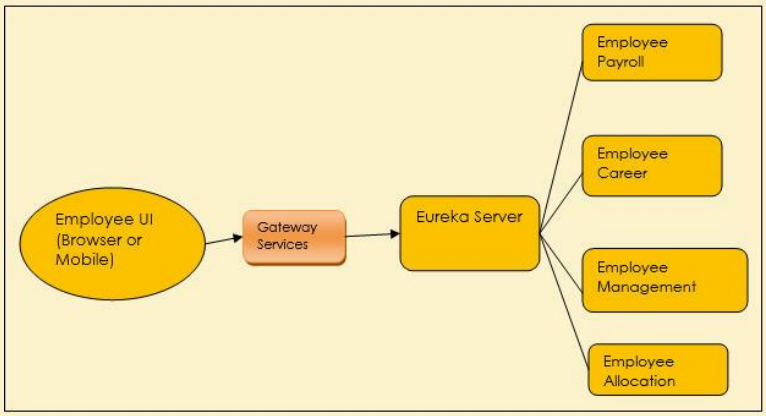


Figure 8.9: Zuul-Enabled Services

8.4 Logging and Monitoring Microservices

The biggest challenge with microservices after their deployment is the logging and monitoring of individual microservices. It is difficult to trace end-to-end transactions just by correlating logs generated by different microservices.

Monitoring microservices is essential to identify and troubleshoot problems. Logging provides the details of the transactions that have occurred and this information is essential to find the source of an error and then debug it.

Monitoring is essential to keep track of the performance of microservices and identify issues that decrease performance.

Therefore, logging and monitoring are critical control systems of microservices.

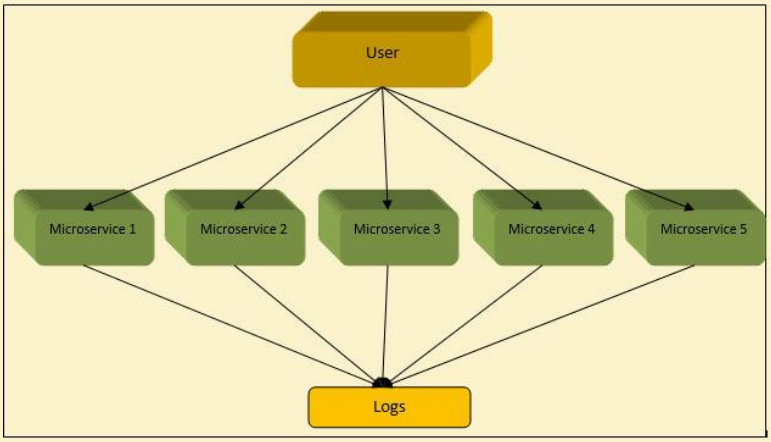


Figure 8.10: Logging in Microservices

Figure 8.10 illustrates how logs are generated in an application containing microservices.

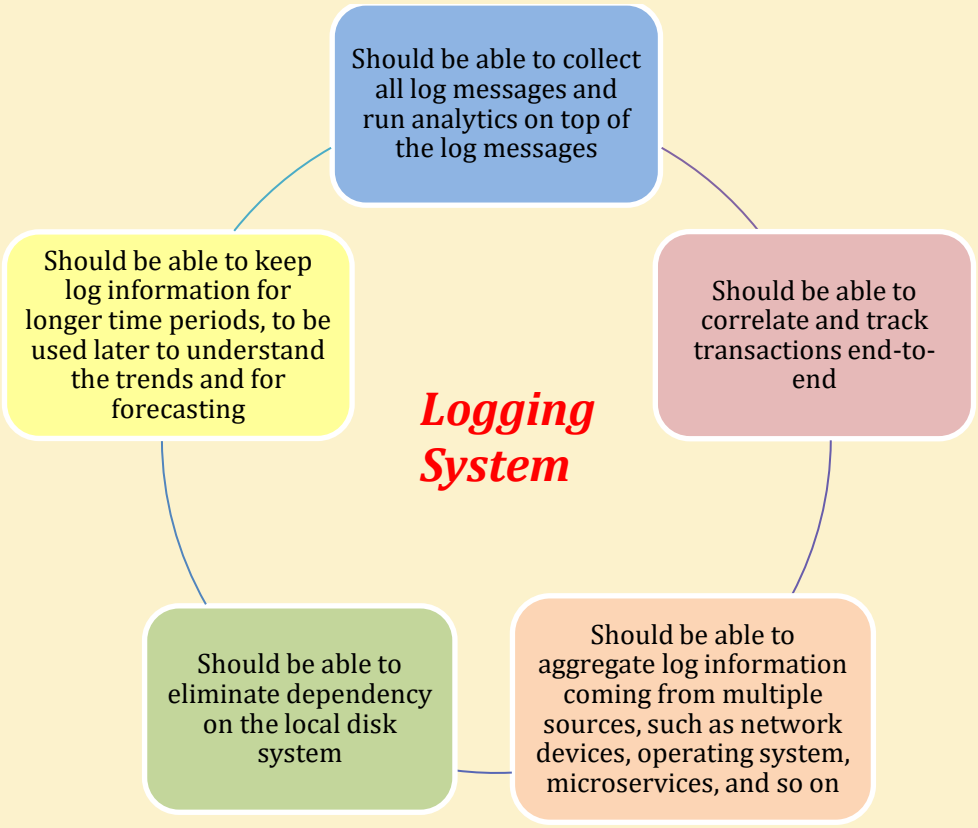
8.4.1 Understanding Log Management

Logs are streams of events coming from a running process. These logs provide an insight into the performance and problems of microservices by capturing valuable information. Traditionally, applications send the log entries to file system or to the console. However, with cloud deployment the applications are no longer confined to a machine. This means that one cannot rely completely on the logs written on the disk. The traditional logging

system also has a major issue of logging files filling up the disk space and consequently, slowing down the entire application. It also affects the scalability of the application.

8.4.2 Need for Centralized Logging

To address the challenges of traditional logging solutions and to extend the capabilities further, a logging solution is expected to do the following:



The solution to all these problems is to have a centralized logging system that centrally stores the logs and analyzes these messages, irrespective of the log source. The fundamental principle on which the new logging solution is based is to remove log storage and processing from services environment. For this, Big Data solutions are best suited for storing, analyzing, and processing large numbers of log messages effectively than storing and processing these log messages in microservice execution environment. In the centralized logging solution, the log messages are sent from the execution environment to a central big data store.

8.4.3 Monitoring Microservices

Monitoring microservices is an integral part of the entire microservice execution environment. Without monitoring, users may face problems for managing complex, large-scale microservices. The traditional monolithic applications were easy to handle as the deployments were limited in terms of instances, services, machines, and so on.

However, enormous numbers of microservices which run on distributed systems are not at all easy to manage and monitor. A centralized logging system solves only a part of the problem, as it is important for users to understand runtime deployment of microservices, arrangement, linking, and behavior of the systems.

Basically, monitoring of microservices involve collection of performance metric and their validation against pre-defined or baseline values. For example, if there is a breach at the service-level, the monitoring tools will immediately generate a breach alert and send it to the administrator.

An efficient monitoring system should work on the following principles:

- Monitor the containers and what's inside them.
- Generate alert on service performance.
- Monitor services that are elastic and available at multiple locations.
- Monitor the APIs.
- Monitoring should be in line with the organizational structure.

8.4.4 Monitoring Tools

There are many monitoring tools available, the choice of which depends on the kind of ecosystem that needs to be monitored. In most cases, more than one monitoring tool is deployed to monitor large microservices.

Here's a list of some of most commonly used monitoring tools:

- New Relic, AppDynamic, and Dynatrace are microservice friendly monitoring tools; these are largely effective in a single console.
- Ruxit, Dataloop, and Datadog offer monitoring tools for microservices in a distributed system.
- Cloud offers its own monitoring tools, but in case of complex microservices these tools may not be enough and may require additional tools. For instance, Google Cloud Platform uses Cloud Monitoring while AWS uses CloudWatch for monitoring its microservices.
- Spring Boot Actuator is one of the good mechanisms to collect microservices metrics, counters, and gauges.
- Monitoring through Logging is also a popular approach.
- Sensu is a popular choice for monitoring microservice from open source community.

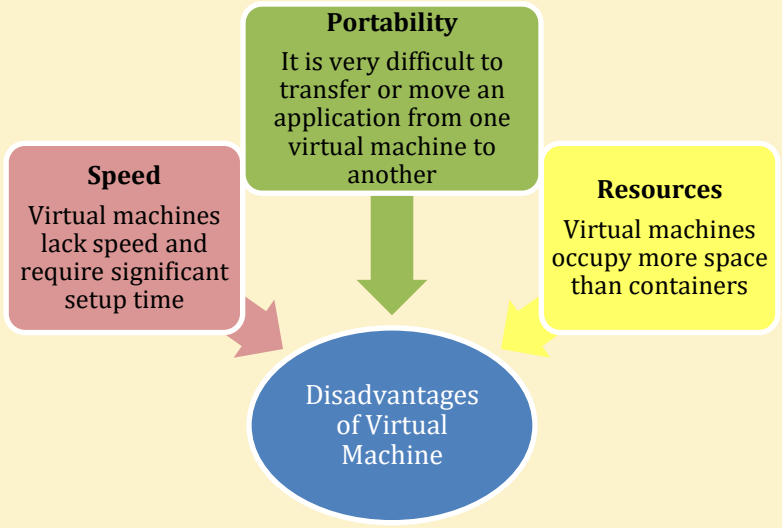
Apart from these tools, many new-generation monitoring tools/applications learn the application's behavior on their own and also set automatic threshold values. This frees up application administrators from doing this task.

It has been observed that automated baselines are sometimes more accurate than human-specified baseline values.

8.5 Containerizing Microservices with Docker

Containerization is commonly explained as 'next-generation virtualization' or 'virtualization of virtualization'. It enables developers to execute an application in a virtual environment, by compiling and storing all the application files and libraries, together at one location, as a package, in a container. This ensures that applications are easily and quickly deployed irrespective of the deployment environment. In addition, containers enable developers to have more control over system resources, which ensures operational efficiency.

This container can be plugged directly into the operating system kernel and does not require creating of a new virtual machine every time a new instance is created or when another application uses the same operating system. Containerization's popularity can be largely attributed to the open source software Docker. The specialty of Docker as compared to other container technologies is that it offers separate workflows for Windows, Linux, and Unix. In addition, Docker smoothly bundles an application in isolation, thus making it easy to move to any operating system or machine. Containerization gained popularity against the traditional virtual machines because the following shortcomings:



8.5.1 What are Containers?

Containers are defined as standardized units to build and combine software components. Containers can bundle together all the binaries and libraries required for running an application on any platform or in any environment. In addition, containers maintain their own IP address, file system, internal processes, network interfaces, dependencies, and other configurations. Containers also isolate a software or application from other applications running on the same infrastructure, which ensures less or no friction between multiple applications. Containers can, therefore, be defined as isolated private virtual spaces or virtual engines, which allow the processes to run on top of the operating system in an isolated environment.

8.5.2 Benefits of Containers

Containers have been available for quite a while, but their increase in demand can be attributed to wide scale adoption of cloud computing and shortcomings of traditional virtual machines. Some benefits of containers are as follows:

- **Security**: Container architecture offers enhanced data security. In case of damage to a container, it limits the damage and prevents it from spreading to the rest of the server.
- **Compatibility**: Containers, such as Dockers allow developers to create completely portable applications that can run on any platform. As such, developers have the freedom to select the language of their choice for the application to be developed. In addition, containers are self-contained and self-sufficient with their own libraries and binaries.
- **Speed**: Containers offer high-level of compatibility with all platforms, this ultimately increases the speed factor. In addition, containers also use fewer resources from the host operating server.
- **Resource utilization**: Improved speed makes servers consume less resources, even when running multiple applications.

8.5.3 Microservices and Containers

Containers are frequently used to run microservices for the simple reason that they serve as lightweight envelopes, which provide software portability. The containers required for microservices can be created or destroyed as per load, which enables microservices to be highly available and scalable. A container will contain the required code to run and execute a microservice, thus allowing the developers to create isolated, efficient, and decoupled execution engines for each service and application. Containers are therefore, a good fit for dynamic microservices. For any project which needs to run in Docker container a developer must add a plugin in pom.xml file as shown here:

```
<plugin>
<groupId>com.spotify</groupId>
<artifactId>docker-maven-plugin</artifactId>
<version>1.1.0</version>
<configuration>
<dockerHost>http://127.0.0.1:2375</dockerHost>
<verbose>true</verbose>
<imageName>${docker.image.prefix}/${project.artifactId}</imageName>
<dockerDirectory>src/main/docker</dockerDirectory>
<resources>
<resource>
<targetPath>/</targetPath>
<directory>${project.build.directory}</directory>
<include>${project.build.finalName}.jar</include>
</resource>
</resources>
```

```
</configuration>
</plugin>
```

The complete configuration specified in the pom.xml file should be as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
<modelVersion>4.0.0</modelVersion>
<groupId>guru.springframework</groupId>
<artifactId>sfg-thymeleaf-course</artifactId>
<version>0.0.1-SNAPSHOT</version>
<packaging>jar</packaging>
<name>sfg-thymeleaf-course</name>
<description>Thymeleaf Course</description>
<parent>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-parent</artifactId>
<version>2.0.3.RELEASE</version>
<relativePath /><!-- lookup parent from repository -->
</parent>
<properties>
<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
<java.version>1.8</java.version>
</properties>
<dependencies>
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-thymeleaf</artifactId>
</dependency>
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-devtools</artifactId>
</dependency>
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-security</artifactId>
</dependency>
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-test</artifactId>
```

```
<scope>test</scope>
</dependency>
</dependencies>
<build>
<plugins>
<plugin>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-maven-plugin</artifactId>
</plugin>
<plugin>
<groupId>com.spotify</groupId>
<artifactId>docker-maven-plugin</artifactId>
<version>1.1.0</version>
<configuration>
<dockerHost>http://127.0.0.1:2375</dockerHost>
<verbose>true</verbose>
<imageName>${docker.image.prefix}/${project.artifactId}</imageName>
<dockerDirectory>src/main/docker</dockerDirectory>
<resources>
<resource>
<targetPath>/</targetPath>
<directory>${project.build.directory}</directory>
<include>${project.build.finalName}.jar</include>
</resource>
</resources>
</configuration>
</plugin>
</plugins>
</build>
</project>
```

Following command can be used to build in STS environment:

```
clean package docker:build
```

Figure 8.11 displays the command in STS Command Window.

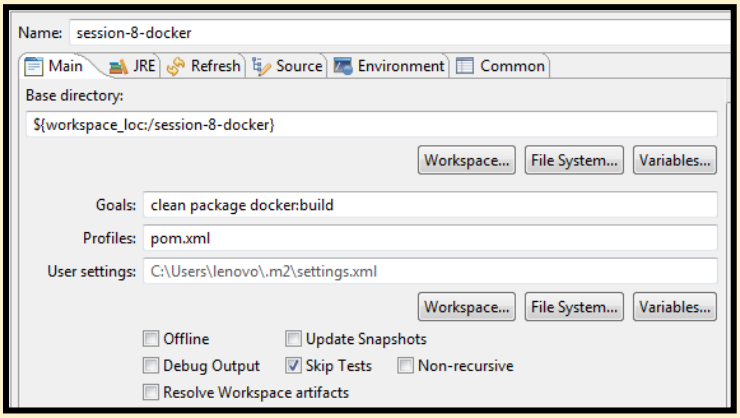


Figure 8.11: STS Command Window

Now, the project is ready to run on Docker platform. Download any commonly used Docker platform.

Since Docker runs on Linux based OS so if developers want to run on Windows then install any VM on their Windows Machine and then Install Docker, otherwise they can install Docker in Linux based OS (say Ubuntu). Refer to the following link to install on Windows:

https://store.docker.com/search?offering=enterprise&type=edition&operating_system=windows

Install Java on Docker platform. Now, the project jar file is ready to run in Docker environment and use following command to run the application:

```
docker run -p8080:8080 -dsession-8-docker
```

Summary

- Spring Cloud is a tool-set that is used to create patterns in distributed applications and provides several libraries to develop, deploy, and operate distributed applications for the cloud.
- Microservices is an architecture that allows developers to combine several small components to build large or enterprise-level applications or systems.
- The principles of microservices are:
  - Single responsibility and no sharing of data or functions between microservices
  - Microservices manage and control their own data storage
  - Group deployment
- HATEOAS stands for Hypermedia as the Engine of Application State and is used to give extra information about a REST API.
- Eureka server is a load-balancing REST-based AWS Discovery service that directs service requests to available instances.
- Zuul is an edge service or a simple gateway service that manages all incoming requests and uses filters to allow interception and identification of incoming requests.
- Monitoring microservices is essential to track their performance and identify issues that decrease their performance.
- Containerization is the process of executing an application in a virtual environment by creating a package of its files and libraries and storing them in a container.

Check Your Progress

1. Which of the following is a gateway service?

A.	Eureka	**B.**	Zuul
C.	Docker	**D.**	LoCS

2. Eureka is based on which of the following services?

A.	AWS-based Discovery	**B.**	Java-based AWS Discovery
C.	HTTP-based Discovery	**D.**	REST-based AWS Discovery

3. Which of the following is known as next-generation virtualization?

A.	Containerization	**B.**	Cloud Config
C.	Eureka	**D.**	Zuul

4. Which of the following are not valid Zuul filters?

A.	Error	**B.**	Route
C.	URL	**D.**	Address

5. Which of the following is a Spring Cloud Netflix library?

A.	Net REST	**B.**	Feign
C.	Artifact	**D.**	SC Cloud

Answers

1.	B
2.	D
3.	A
4.	C and D
5.	B