SPRING FRAMEWORK COOKBOOK

Hot Recipes for the Spring Framework

JAVA CODE GEEKS

Java Code Geeks
JAVA 2 JAVA DEVELOPERS RESOURCE CENTER

Spring Framework Cookbook

Contents

1 Spring Framework Best Practices **1**

1.1 Define singleton beans with names same as their class or interface names . . . 1
1.2 Place Spring bean configuration files under a folder instead of root folder . . . 1
1.3 Give common prefixes or suffixes to Spring bean configuration files . . . 2
1.4 Avoid using import elements within Spring XML configuration files as much as possible . . . 2
1.5 Stay away from auto wiring in XML based bean configurations . . . 2
1.6 Always externalize bean property values with property placeholders . . . 3
1.7 Select default version-less XSD when importing namespace definitions . . . 3
1.8 Always place classpath prefix in resource paths . . . 4
1.9 Create a setter method even though you use field level auto wiring . . . 4
1.10 Create a separate service layer even though service methods barely delegate their responsibilities to corresponding DAO methods . . . 4
1.11 Use stereotype annotations as much as possible when employing annotation driven bean configuration . . . 5
1.12 Group handler methods according to related scenarios in different Controller beans . . . 6
1.13 Place annotations over concrete classes and their methods instead of their interfaces . . . 6
1.14 Prefer throwing runtime exceptions instead of checked exceptions from service layer . . . 6
1.15 Manage transactions only in the service layer . . . 7
1.16 Mark transactions as readOnly=true when service methods only contain queries . . . 7
1.17 Be aware of false positives in transactional ORM integration tests . . . 8
1.18 Do not use DriverManagerDataSource . . . 8
1.19 Either use NamedParameterJdbcTemplate or JdbcTemplate for your JDBC operations . . . 9
1.20 Use SessionFactory and EntityManager directly in your DAO beans . . . 9
1.21 Summary . . . 10

2 Spring 4 Autowire Example **11**

2.1 Introduction . . . 11
2.2 Usage of Autowire . . . 11
2.3 Step by Step Implementation . . . 11
2.3.1 Create the Application . . . 11
2.3.2 Configure POM.xml (maven) . . . 12

2.3.3 Create Services . . . 13
2.3.4 Configure beans (applicationContext.xml) . . . 14
2.3.5 Create the class that will use (injection) the service . . . 14
2.3.6 Test it Out! . . . 15
2.4 Download the Eclipse project of this tutorial: . . . 15

3 How to write Transactional Unit Tests with Spring 16
3.1 Create a new Maven Project . . . 16
3.2 Add necessary dependencies in your project . . . 19
3.3 Create log4j.xml file in your project . . . 24
3.4 Prepare DDL and DML scripts to initialize database . . . 24
3.5 Write Domain Class, Service and DAO Beans . . . 25
3.6 Configure Spring ApplicationContext . . . 27
3.7 Write a transactional integration unit test . . . 28
3.8 Run the tests and observe the results . . . 29
3.9 Summary . . . 31
3.10 Download the Source Code . . . 31

4 Spring Framework JMSTemplate Example 32
4.1 Dependencies . . . 32
4.2 Sending and Receiving Messages without JmsTemplate . . . 33
4.3 Configuring JmsTemplate . . . 35
4.4 Using JMSTemplate to produce messages . . . 36
4.5 Using JMSTemplate to consume messages . . . 37
4.6 Complete JmsTemplate example to send/receive messages . . . 38
4.7 JmsTemplate with Default destination . . . 39
4.8 JmsTemplate with MessageConverter . . . 42
4.9 Configuring MessageConverter . . . 44
4.10 Download the Eclipse Project . . . 45

5 How to Start Developing Layered Web Applications with Spring 46
5.1 Create a new Maven WebApp project . . . 46
5.2 Add necessary dependencies in your project . . . 51
5.3 Create log4j.xml . . . 56
5.4 Prepare DDL and DML scripts to initialize database . . . 57
5.4.1 schema.sql . . . 57
5.4.2 data.sql . . . 57
5.5 Write Domain Class, Service and DAO Classes . . . 57
5.5.1 Person.java . . . 57
5.5.2 PersonDao.java . . . 58

5.5.3 JdbcPersonDao.java . . . 58
5.5.4 PersonService.java . . . 60
5.5.5 PersonServiceImpl.java . . . 60
5.6 Write Controller Classes and JSPs to handle UI logic . . . 61
5.6.1 PersonListController and personList.jsp . . . 61
5.6.2 PersonCreateController and personCreate.jsp . . . 63
5.6.3 PersonUpdateController and personUpdate.jsp . . . 64
5.6.4 PersonDeleteController and personDelete.jsp . . . 66
5.7 Configure your web application to bootstrap with Spring . . . 67
5.7.1 WebAppConfig.java . . . 67
5.7.2 WebAppInitializer.java . . . 68
5.8 Configure your IDE to run Tomcat instance . . . 69
5.9 Run Tomcat instance and access your webapp through your browser . . . 74
5.10 Summary . . . 75
5.11 Download the Source Code . . . 75

6 Angularjs and Spring Integration Tutorial 76
6.1 What is Spring? . . . 76
6.2 What Is Angular? . . . 76
6.3 Create a New Project . . . 76
6.3.1 Maven dependencies . . . 77
6.3.2 Web app java-based configuration . . . 79
6.3.3 SpringMVC controller and jsp . . . 80
6.3.4 Angularjs controllers and js files . . . 81
6.3.5 Build and run the application on tomcat . . . 82
6.4 Download the source code . . . 83

7 Spring MVC Application with Spring Security Example 84
7.1 Introduction to Spring Security . . . 84
7.2 Project Setup . . . 84
7.3 Project Implementation . . . 87
7.4 Download the Source Code . . . 93

8 Spring MVC Hibernate Tutorial 94
8.1 Introduction . . . 94
8.2 Environment . . . 94
8.3 Spring MVC Framework . . . 94
8.4 Hibernate For Model . . . 95
8.5 Example . . . 95
8.5.1 Maven Project and POM dependencies . . . 95

8.5.2 Configure Hibernate . . . 100
8.5.3 Domain Entity Class . . . 101
8.5.4 Service Layer . . . 103
8.5.5 DAO Layer . . . 105
8.5.6 Configure Spring MVC . . . 107
8.5.7 Initializer Class . . . 108
8.5.8 Application Controller . . . 108
8.5.9 Views . . . 111
8.5.10 Deploy and running the app . . . 113
8.6 Download . . . 114
8.7 Related Articles . . . 114

9 Spring rest template example 115
9.1 Download the Source Code . . . 118

10 Spring data tutorial for beginners 119
10.1 Output: . . . 125
10.2 Download the Source Code . . . 126

11 Spring Batch Tasklet Example 127
11.1 Introduction . . . 127
11.2 Spring Batch Framework: Key Concepts . . . 127
11.2.1 Jobs . . . 127
11.2.2 Steps . . . 128
11.2.2.1 ItemReader . . . 129
11.2.2.2 ItemProcessor . . . 129
11.2.2.3 ItemWriter . . . 129
11.2.2.4 Chunk Processing . . . 129
11.2.2.5 TaskletStep Processing . . . 130
11.2.3 Tasklet Example . . . 131
11.2.3.1 Tools used . . . 131
11.2.3.2 Create a Maven Project . . . 131
11.2.3.3 Add Dependencies . . . 135
11.2.3.4 Add db2* jars . . . 136
11.2.3.5 HSQL Table Creation . . . 136
11.2.3.6 Supply Sample Data . . . 136
11.2.3.7 Data Model . . . 137
11.2.3.8 RowMapper . . . 138
11.2.3.9 Tasklet . . . 138
11.2.3.10 Job Configuration . . . 139

11.2.3.11 Context Configuration . . . 141
11.2.3.12 Properties File . . . 142
11.2.3.13 Run the Application . . . 142
11.2.3.14 Output . . . 143
11.2.4 Download Example . . . 143

12 Spring Boot Tutorial for beginners 144
12.1 Introduction . . . 144
12.2 Environment . . . 144
12.3 Sample Application using Spring Boot . . . 144
12.3.1 Create and configure a Gradle project in Eclipse IDE . . . 144
12.3.2 build.gradle . . . 151
12.3.2.1 Modify build.gradle . . . 151
12.3.2.2 Walk through build.gradle . . . 152
12.3.2.3 Run initial build . . . 153
12.3.3 Create SampleApplication.java . . . 153
12.3.4 Create SampleController.java . . . 158
12.3.5 SampleApplication.java . . . 163
12.3.5.1 Modify SampleApplication.java . . . 163
12.3.6 Run SampleApplication . . . 163
12.4 References . . . 164
12.5 Conclusion . . . 164
12.6 Download the Eclipse project . . . 165

13 Spring Session Tutorial 166
13.1 Introduction . . . 166
13.2 Project Set-Up . . . 166
13.3 Implementation . . . 168
13.3.1 Sticky Session . . . 168
13.3.2 Single Sign On . . . 172
13.4 Download The Source Code . . . 177

14 Spring Web Flow Tutoriall 178
14.1 Introduction . . . 178
14.2 Project Set-Up . . . 178
14.3 Implementation . . . 179
14.4 Download The Source Code . . . 184

Preface

The Spring Framework is an open-source application framework and inversion of control container for the Java platform. The framework's core features can be used by any Java application, but there are extensions for building web applications on top of the Java EE platform. Although the framework does not impose any specific programming model, it has become popular in the Java community as an alternative to, replacement for, or even addition to the Enterprise JavaBeans (EJB) model. (Source: https://en.wikipedia.org/wiki/Spring_Framework).

Spring helps development teams everywhere build simple, portable, fast and flexible JVM-based systems and applications. The project's mission is to help developers build a better Enterprise. (Source: https://spring.io/)

In this ebook, we provide a compilation of Spring Framework tutorials that will help you kick-start your own programming projects. We cover a wide range of topics, from basic usage and best practices, to specific projects like Boot and Batch. With our straightforward tutorials, you will be able to get your own projects up and running in minimum time.

About the Author

JCGs (Java Code Geeks) is an independent online community focused on creating the ultimate Java to Java developers resource center; targeted at the technical architect, technical team lead (senior developer), project manager and junior developers alike.

JCGs serve the Java, SOA, Agile and Telecom communities with daily news written by domain experts, articles, tutorials, reviews, announcements, code snippets and open source projects.

You can find them online at https://www.javacodegeeks.com/

Chapter 1

Spring Framework Best Practices

Spring Application Framework has been in action for quite a long time, and programmers have developed several conventions, usage patterns, and idioms during that time period. In this example, we will try to explain some of them and give examples to illustrate how to apply them in your projects.

Let's begin.

1.1 Define singleton beans with names same as their class or interface names

Most of the bean definitions in Spring ApplicationContext are singleton scope, and again they are mostly sole bean definitions of their classes in the application. Developers therefore, give them names same as with their class or interface names in order to easily match with bean definitions with their classes. That way, it becomes easier to go from beans to their classes or vice versa.

```
public class SecurityServiceImpl implements SecurityService {

        @Override
        public String getCurrentUser() {
                //...
        }

}
```

```
<bean id="securityService" class="com.example.service.SecurityServiceImpl">
                ...
        </bean>
```

1.2 Place Spring bean configuration files under a folder instead of root folder

If you place xml configuration files under root class path, and create a jar then, Spring might fail to discover those xml bean configuration files within jar file, if they are loaded with wildcards like below.

```
<web-app>
                <context-param>
                        <param-name>contextConfigLocation</param-name>
                        <param-value>classpath*:/beans-*.xml</param-value>
                </context-param>
        </web-app>
```

This problem is related with a limitation of Java IO API, and it is better to create a folder such as /beans or /appcontext and place xml configuration files beneath it. That way, it becomes safe to employ wildcards while loading them from jar archives.

1.3 Give common prefixes or suffixes to Spring bean configuration files

If you give common prefixes or suffixes to xml bean configuration files in the application, like beans-service.xml, beans-dao.xml, beans-security.xml, beans-config.xml and so on, then it becomes easier to load those xml configuration files while creating Spring Container using wildcards as follows.

```
<web-app>
         <context-param>
                <param-name>contextConfigLocation</param-name>
                <param-value>classpath*:/appcontext/beans-*.xml</param-value>
         </context-param>
</web-app>
```

1.4 Avoid using import elements within Spring XML configuration files as much as possible

Spring XML based configuration offers element to include bean definitions within another xml file. However, you should use element wisely. If you use it within several different places in your xml configuration files, it becomes difficult to grasp big picture of the system configuration and get confused about bean definition overrides as well. Instead, either prefer loading xml configuration files by making use of wild cards as explained in the previous tip, or create a separate xml configuration file, whose sole purpose is just to contain elements.

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

        <import resource="classpath:/appcontext/beans-controller.xml"/>
        <import resource="classpath:/appcontext/beans-service.xml"/>
        <import resource="classpath:/appcontext/beans-dao.xml"/>

</beans>
```

1.5 Stay away from auto wiring in XML based bean configurations

Mixing auto wiring with explicit setter or constructor injection in xml bean definitions might cause confusion and make it harder to grasp the big picture in the application. Therefore, either make use of auto wiring in all of your bean definitions throughout the application, or stick with the explicit dependency injection definitions.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="https://www.springframework.org/schema/beans"
        xmlns:xsi="https://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="https://www.springframework.org/schema/beans https://www. ↩
            springframework.org/schema/beans/spring-beans.xsd" default-autowire="byType">

        ...

</beans>
```

1.6 Always externalize bean property values with property placeholders

Instead of placing hard coded values in bean definitions, place property placeholder variables in place of actual values. That way, it will be easier to customize system configuration according to the target runtime environment without requiring any modifications in the bean configurations.

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

        <bean id="dataSource" class="org.springframework.jdbc.datasource. ↩
            DriverManagerDataSource">
                <property name="driverClassName" value="${dataSource.driverClassName}"/>
                <property name="url" value="${dataSource.url}"/>
                <property name="username" value="${dataSource.username}"/>
                <property name="password" value="${dataSource.password}"/>
        </bean>

        <context:property-placeholder location="classpath:application.properties"/>

</beans>
```

1.7 Select default version-less XSD when importing namespace definitions

Namespaces are introduced into Spring in order to simplify complex bean configurations, and enable Spring features in a more natural way. You need to add namespace XSD into xml configuration files in order to make use of namespace elements available in Spring modules as follows.

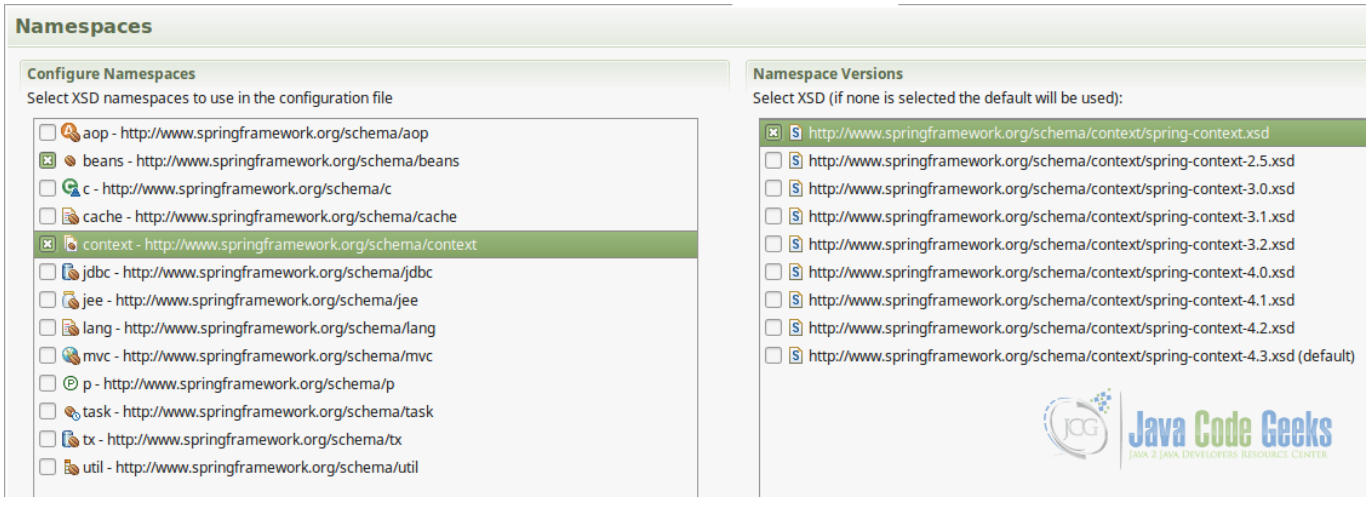

Figure 1.1: spring namespace xsd versions

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="https://www.springframework.org/schema/beans"
        xmlns:xsi="https://www.w3.org/2001/XMLSchema-instance"
```

```
        xmlns:context="https://www.springframework.org/schema/context"
        xsi:schemaLocation="https://www.springframework.org/schema/beans https://www. ↩
            springframework.org/schema/beans/spring-beans.xsd
                https://www.springframework.org/schema/context https://www.springframework. ↩
                    org/schema/context/spring-context.xsd">

        ...

</beans>
```

Spring introduces new namespace elements in each new version, and if you place Spring version number in namespace XSD, you will be missing new features introduced in the upcoming Spring releases. If you exclude version number in the XSD, its current version is enabled, and whenever you upgrade Spring version in the project, latest namespace elements of Spring modules will be available without any other extra effort.

1.8 Always place classpath prefix in resource paths

Unless you place resource type prefix in your resource paths, type of the Spring ApplicationContext determines the location from where those resource paths will be resolved.

```
<context:property-placeholder location="application.properties"/>
```

For example, in the above configuration application.properties file, placed into classpath will be looked up from classpath when ApplicationContext is created during Spring integration tests, and it will be loaded without any problem. However, when it comes to load it during bootstrap of the web application, Spring WebApplicationContext will attempt to resolve it from context root instead of classpath, and therefore will fail. Hence, it is almost always better to place your resources somewhere under classpath and place classpath: prefix in front of their paths.

```
<context:property-placeholder location="classpath:application.properties"/>
```

1.9 Create a setter method even though you use field level auto wiring

Spring supports field level injection in addition to setter and constructor injection methods. However, you will need those setters when you attempt to unit test those classes. Hence, it is still important to create setter methods even though you place @Autowired on top your attributes.

```
@Service
public class SecurityServiceImpl implements SecurityService {

        @Autowired
        private SecurityDao securityDao;

        public void setSecurityDao(SecurityDao securityDao) {
                this.securityDao = securityDao;
        }
}
```

1.10 Create a separate service layer even though service methods barely delegate their responsibilities to corresponding DAO methods

Creating a separate service layer and service classes almost always pays off in the long term even though service methods merely delegate their responsibilities to their DAO counterparts.

In the beginning, your system might look like so simple and a separate service layer might look useless.

However, it is still useful to create a separate service layer as many of Spring features like transaction management, method level security, method level caching or service method parameter validations best suit to that service layer. If you start with a separate service layer from the beginning, it will be simply a matter of applying related annotations to enable those features in the application.

```
@Service
public class SecurityServiceImpl implements SecurityService {

        @Autowired
        private SecurityDao securityDao;

        public void setSecurityDao(SecurityDao securityDao) {
                this.securityDao = securityDao;
        }

        @Transactional(readOnly=true)
        @Override
        public User findUserByUsername(String username) {
                return securityDao.findUserByUsername();
        }

}
```

1.11 Use stereotype annotations as much as possible when employing annotation driven bean configuration

Spring annotation based configuration offers several annotations, like @Controller, @Service , @Repository and so on. They all inherit from @Component annotation as well. Although it is possible to create beans with only using @Component annotation, you will be missing some functionality which becomes available on your beans when they are defined with appropriate stereo type annotations.

For example, @Repository annotation helps handling of Hibernate or JPA specific exceptions and converting them into Spring specific DataAccessExceptions. @Controller annotation signals to DispatcherServlet that, it contains handler methods with @RequestMapping annotation. Although @Service annotation doesn't make all of the public methods transactional in a service bean - like session beans in EJBs, it is just a matter of defining an annotation which brings those @Service and @Transactional annotations together, or write an aspect to achieve similar behavior.

```
@Controller
public class SecurityController {

        private SecurityService securityService;

        @Autowired
        public void setSecurityService(SecurityService securityService) {
                this.securityService = securityService;
        }

        //...
}

@Service
public class SecurityServiceImpl implements SecurityService {

        @Autowired
        private SecurityDao securityDao;

        //...
```

```
}

@Repository
public class HibernateSecurityDao implements SecurityDao {

        private SessionFactory sessionFactory;

        @Autowired
        public void setSessionFactory(SessionFactory sessionFactory) {
                this.sessionFactory = sessionFactory;
        }

        //...
}
```

1.12 Group handler methods according to related scenarios in different Controller beans

Spring MVC allows you to write multiple handler methods within a single Controller bean. However, this approach might lead Controller classes to get cluttered unless you become careful. For example, you will need to add methods that initialize model objects using @ModelAttribute annotation, or add exception handler methods with @ExceptionHandler annotation, or init methods to initialize WebDataBinder with @InitBinder annotation, and things will get collide with each other for different scenarios over time. Instead, you should create several Controller classes for each group of related scenarios in the application, so that any of those initializing or error handling methods related with the same scenarios goes into same Controller classes. This will result in more manageable and understandable Controller beans in the end.

Do not place business logic in Controller beans. Role of Controller beans is to handle web request, extract user submitted data, convert it into an appropriate form for service layer, invoke service layer for business execution, then get the result from service layer and build up response for the user to be shown. Do not let the business logic leak into the Controller beans. The only logic allowed in the Controller beans should be UI logic, which is mainly related with managing state for the UI, nothing else.

1.13 Place annotations over concrete classes and their methods instead of their interfaces

You should place Spring annotations only over classes, their fields or methods, not on interfaces or methods declared within them, as Java doesn't allow annotations placed on interfaces to be inherited by the implementing classes.

1.14 Prefer throwing runtime exceptions instead of checked exceptions from service layer

Default rollback behavior for @Transactional annotation is to commit when a checked exception is thrown from within a transactional method, instead of rollback, as opposed to its counterpart, unchecked exceptions, which cause rollback by default. However, most of the time developers need rollback behavior for checked exceptions as well. Therefore, they override default rollback behavior, whenever they throw checked exceptions and want to cause rollback. Instead of repeating this step each time for your transactional service methods, it will be much safer to throw unchecked exceptions from within those service methods.

```
@Service
public class SecurityServiceImpl implements SecurityService {

        @Autowired
        private SecurityDao securityDao;
```

```
        @Override
        public User findUserByUsername(String username) {
                User user = securityDao.findUserByUsername();
                if(user == null) throw new UserNotFoundException("User not found :" +  ↩
                    username);
                return user;
        }

        //...
}
```

1.15 Manage transactions only in the service layer

The place to demarcate transactions in a Spring enabled application is service layer, nowhere else. You should only mark @Service beans as @Transactional or their public methods.

```
@Service
public class SecurityServiceImpl implements SecurityService {

        @Autowired
        private SecurityDao securityDao;

        @Override
        @Transactional(readOnly=true)
        public User findUserByUsername(String username) {
                //...
        }

        @Override
        @Transactional
        public void createUser(User user) {
                //...
        }

        @Override
        @Transactional
        public void updateUser(User user) {
                //...
        }

        @Override
        @Transactional
        public void deleteUser(User user) {
                //...
        }
}
```

You can still place @Transactional with propagation=Propagation.MANDATORY over DAO classes so that they wouldn't be accessed without an active transaction at all.

1.16 Mark transactions as readOnly=true when service methods only contain queries

In order to be able to use Hibernate contextual session capability, you need to start a transaction even for select operations. Therefore, you even mark your finder methods with @Transactional annotation in service beans. However, at the end of the finder method, transaction is committed, and Hibernate session flush will be triggered via that commit. Hibernate flush is an expensive operation, which traverses all those entities existing in the Hibernate Session, and try to detect dirty entities within it.

Such a dirty checking step obviously becomes unnecessary when we only perform select queries. Turning flush mode to manual prevents automatic flushing at the end of the transaction commit, and this will bring us a slight performance improvement, in addition to preventing unintended data modifications in the application.

```
@Service
public class SecurityServiceImpl implements SecurityService {

        @Autowired
        private SecurityDao securityDao;

        @Override
        @Transactional(readOnly=true)
        public User findUserByUsername(String username) {
                //...
        }

        //...
}
```

1.17 Be aware of false positives in transactional ORM integration tests

Spring TextContext Framework helps us to create transactional integration tests so that it becomes easier to test data access operations. It rollbacks the transaction created at the end of the test method in order not to cause side effects to other tests to be run next. If you are using JPA or Hibernate in your data access operations, JPA/Hibernate won't flush as the transaction rolls back, and SQL statements won't hit the database therefore. Hence, you won't be aware of any problems like constraint violations caused by those data access operations as no SQL is executed actually.

In order to overcome this problem, you need to inject SessionFactory or EntityManager, and perform flush before assert statements in the test methods.

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration("classpath:/appcontext/beans-*.xml")
public class SecurityIntegrationTests {
        @Autowired
        private SessionFactory sessionFactory;

        @Autowired
        private SecurityDao securityDao;

        @Test
        @Transactional
        public void shouldCreateNewUser() {
                User user = new User();
                user.setUsername("john");
                user.setPassword("secret");

                securityDao.createUser(user);

                sessionFactory.getCurrentSession().flush();
        }
}
```

1.18 Do not use DriverManagerDataSource

DriverManagerDataSource class is mostly used one to exemplify dataSource bean configurations throughout Spring related examples. However, DriverManagerDataSource causes a new physical connection to be opened each time you ask for an SQL

Connection from it, as it doesn't have a pooling mechanism. It is suitable only for development or testing environments. You should not use it in production environment. Instead you should either access dataSource bean configured within your application server via JNDI, or include an open source connection pooling library, like C3PO, Apache Commons DBCP or Hikari, and get connections through that connection pool.

```
<jee:jndi-lookup jndi-name="java:comp/env/jdbc/myDS" id="dataSource"/>
```

1.19 Either use NamedParameterJdbcTemplate or JdbcTemplate for your JDBC operations

Spring Data Access module provides two high level helper classes, JdbcTemplate and NamedParameterJdbcTemplate. You should use either one of them to perform any of your JDBC operations, instead of getting dataSource bean and opening up JDBC connections manually. Those template method based classes handle most of the repetitious code blocks internally, and relieves us from managing JDBC connections by ourselves. They also simplify combining ORM operations with native JDBC ones in the same transaction.

```
@Repository
public class JdbcSecurityDao implements SecurityDao {

        private JdbcTemplate jdbcTemplate;

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

        <bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">
                <property name="dataSource" ref="dataSource"/>
        </bean>
```

1.20 Use SessionFactory and EntityManager directly in your DAO beans

Before introduction of contextual session capability of Hibernate, Spring had provided HibernateTemplate helper class, similar to JdbcTemplate to simplify ORM operations inside DAO classes. The other class provided by Spring was HibernateDaoSupport for DAO classes to extend from for similar purposes. However, with the introduction of contextual session capability, working with Hibernate has been greatly simplified, and reduced to injecting SessionFactory into DAO beans, and calling getCurrentSession() to access transactional current Session to perform persistence operations. Therefore, prefer that type of usage within your DAO beans instead of cluttering them with an additional helper or base class.

```
@Repository
public class HibernateSecurityDao implements SecurityDao {

        private SessionFactory sessionFactory;

        @Autowired
        public void setSessionFactory(SessionFactory sessionFactory) {
                this.sessionFactory = sessionFactory;
        }

        public User findUserByUsername(String username) {
                return sessionFactory.getCurrentSession().createQuery("from User u where u. ↩
                    username = :username")
                                .setParameter("username", username).uniqueResult();
        }
}
```

1.21 Summary

In this article, I tried to list some common Spring usage practices and idioms developed over the years. Spring is a quite big project, and of course, best practices are not limited with only those explained above. I list the most popular and common ones which are also applied by myself, and sure there are tons of others as well. Nevertheless, they should help you start employing Spring features in a much more appropriate way within your projects.

Chapter 2

Spring 4 Autowire Example

2.1 Introduction

Autowiring is method of creating an instance of an object and "by concept" injecting that instance on a specific class that uses it. By that, therefore creates a "wiring" of an instance to a class that will use it's attributes. In Spring, when the application server initialize the context, it creates a stack/heaps of objects in it's JVM container. This objects are available for consumption at any given time as long as the application is running (runtime). Now once these objects are ready, they can now be injected to different classes that belongs on the same application context. In a standard Java application, we can use the `ClassPathXmlApplicationContext` class to create instances of the beans on the IoC container (JVM), making them available to be injected Or wired on any Java objects that needs it.

2.2 Usage of Autowire

In this example, I will show you how a bean is wired by create the classes (beans) inside the `applicationContext.xml` and using `ClassPathXmlApplicationContext` to create the object instance that will be used by our AppMain.java class.

2.3 Step by Step Implementation

2.3.1 Create the Application

Create the Java project. I would suggest using Maven so that we can easily get the dependency if needed.

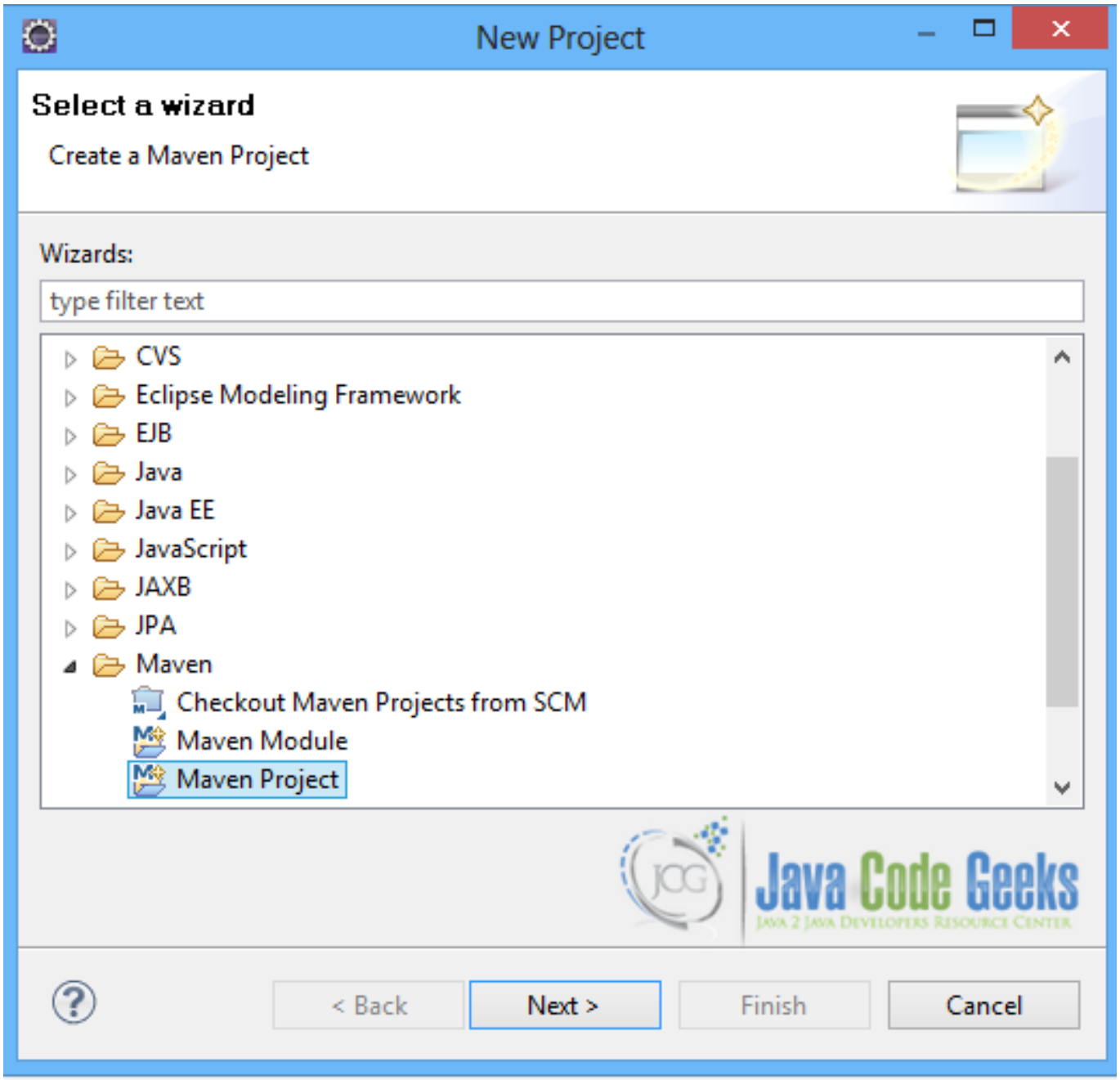

Figure 2.1: New Maven Project

2.3.2 Configure POM.xml (maven)

We need to add the spring dependency on our project. Add spring core and framework.

pom.xml

```
<project xmlns="https://maven.apache.org/POM/4.0.0" xmlns:xsi="https://www.w3.org/2001/ ↩
    XMLSchema-instance"
        xsi:schemaLocation="https://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd ↩
            /maven-4.0.0.xsd">
        <modelVersion>4.0.0</modelVersion>
        <groupId>com.jgc.areyes1.sample</groupId>
```

```
        <artifactId>spring-autowire-example</artifactId>
        <version>0.0.1-SNAPSHOT</version>


        <dependencies>
                <dependency>
                        <groupId>org.springframework</groupId>
                        <artifactId>spring-core</artifactId>
                        <version>${spring.version}</version>
                </dependency>
                <dependency>
                        <groupId>org.springframework</groupId>
                        <artifactId>spring-context</artifactId>
                        <version>${spring.version}</version>
                </dependency>
        </dependencies>

        <properties>
                <spring.version>4.1.6.RELEASE</spring.version>
        </properties>
</project>
```

2.3.3 Create Services

We then create the service that we will eventually define on the `applicationContext.xml`.

UserAccountService.java

```
package com.javacodegeeks.areyes1.beans;

public class UserAccountService {

        public UserAccountService() {
                this.name = "Alvin Reyes";
                this.description = "Account is activated with enough funds for equity  ↩
                    trading";
                this.details = "PHP10000.00";
        }

        private String name;
        private String description;
        private String details;
        public String getName() {
                return name;
        }
        public void setName(String name) {
                this.name = name;
        }
        public String getDescription() {
                return description;
        }
        public void setDescription(String description) {
                this.description = description;
        }
        public String getDetails() {
                return details;
        }
        public void setDetails(String details) {
                this.details = details;
        }
```

```
}
```

2.3.4 Configure beans (applicationContext.xml)

We then create the `applicationContext.xml` inside the resources folder. This is for it exist on the classpath

applicationContext.xml

```
<beans xmlns="https://www.springframework.org/schema/beans"
        xmlns:xsi="https://www.w3.org/2001/XMLSchema-instance" xmlns:p="https://www. ↩
            springframework.org/schema/p"
        xmlns:aop="https://www.springframework.org/schema/aop" xmlns:context="https://www. ↩
            springframework.org/schema/context"
        xmlns:jee="https://www.springframework.org/schema/jee" xmlns:tx="https://www. ↩
            springframework.org/schema/tx"
        xmlns:task="https://www.springframework.org/schema/task"
        xsi:schemaLocation="https://www.springframework.org/schema/aop https://www. ↩
            springframework.org/schema/aop/spring-aop-3.2.xsd https://www.springframework. ↩
            org/schema/beans https://www.springframework.org/schema/beans/spring-beans-3.2. ↩
            xsd https://www.springframework.org/schema/context https://www.springframework. ↩
            org/schema/context/spring-context-3.2.xsd https://www.springframework.org/schema ↩
            /jee https://www.springframework.org/schema/jee/spring-jee-3.2.xsd https://www. ↩
            springframework.org/schema/tx https://www.springframework.org/schema/tx/spring- ↩
            tx-3.2.xsd https://www.springframework.org/schema/task https://www. ↩
            springframework.org/schema/task/spring-task-3.2.xsd">

        <context:annotation-config />
        <!-- <bean class="org.springframework.beans.factory.annotation. ↩
            AutowiredAnnotationBeanPostProcessor" /> -->



        <bean id="userAccountService" autowire="byName"  class="com.javacodegeeks.areyes1. ↩
            beans.UserAccountService">
        </bean>


</beans>
```

2.3.5 Create the class that will use (injection) the service

Create the class that will call the beans. As you can see, we called the bean by name to get the instance of that object (cat and dog).

App.java

```
package com.javacodegeeks.areyes1.main;

import org.springframework.context.support.ClassPathXmlApplicationContext;

import com.javacodegeeks.areyes1.beans.UserAccountService;

public class AppMain {


        private UserAccountService userAccountService;

        public AppMain() {
                ClassPathXmlApplicationContext context =  new  ↩
                    ClassPathXmlApplicationContext("classpath*:/applicationContext.xml");
```

```
                UserAccountService userAccountService = (UserAccountService)context.getBean ↩
                    ("userAccountService");
                System.out.println(userAccountService.getName());
                System.out.println(userAccountService.getDetails());
                System.out.println(userAccountService.getDescription());

                context.close();
        }


        public static void main(String[] args ) {
                new AppMain();
        }
}
```

2.3.6 Test it Out!

Run the AppMain.java and see it for yourself! You should see the following result

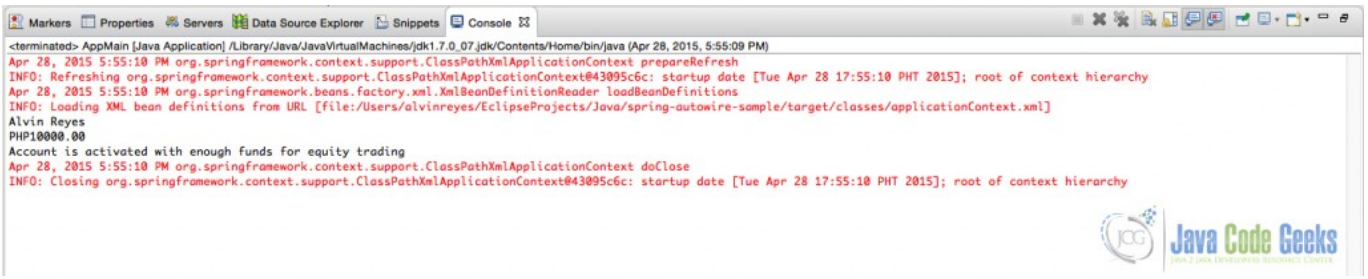

Figure 2.2: Results of running AppMain.java

2.4 Download the Eclipse project of this tutorial:

Download

You can download the full source code of this example here : **spring-autowire-sample**

Chapter 3

How to write Transactional Unit Tests with Spring

Spring is a great framework to develop enterprise Java web applications. It provides tons of features for us. One of them is its TestContext Framework, which helps us to implement integration unit tests easily in our enterprise applications.

Integration unit tests may cover several layers and include ApplicationContext loading, transactional persistence operations, security checks and so on. In this example, we will show you how to write transactional integration unit tests in your enterprise application so that you can be sure that your data access logic or persistence operations work as expected within an active transaction context.

Our preferred development environment is Spring Tool Suite 3.8.2 based on Eclipse 4.6.1 version. However, as we are going to create the example as maven project, you can easily work within your own IDE as well. We are also using Spring Application Framework 4.3.1.RELEASE along with JDK 1.8_u112, and H2 database version 1.4.192.

Let's begin.

3.1 Create a new Maven Project

Write click on Package Explorer and select New>Maven Project to create an new maven project by skipping archetype selection. This will create a simple maven project.

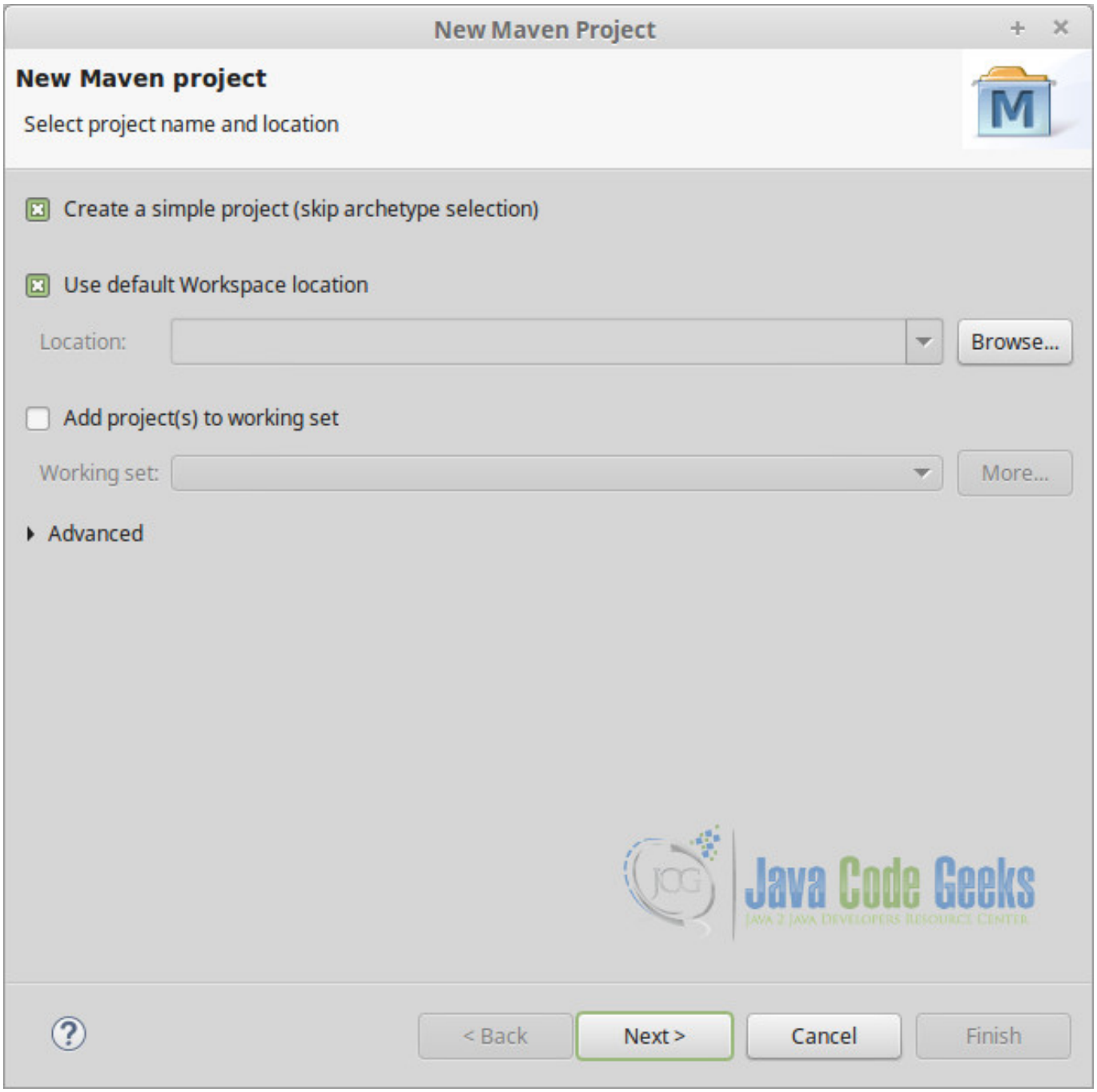

Figure 3.1: Create New Maven Project

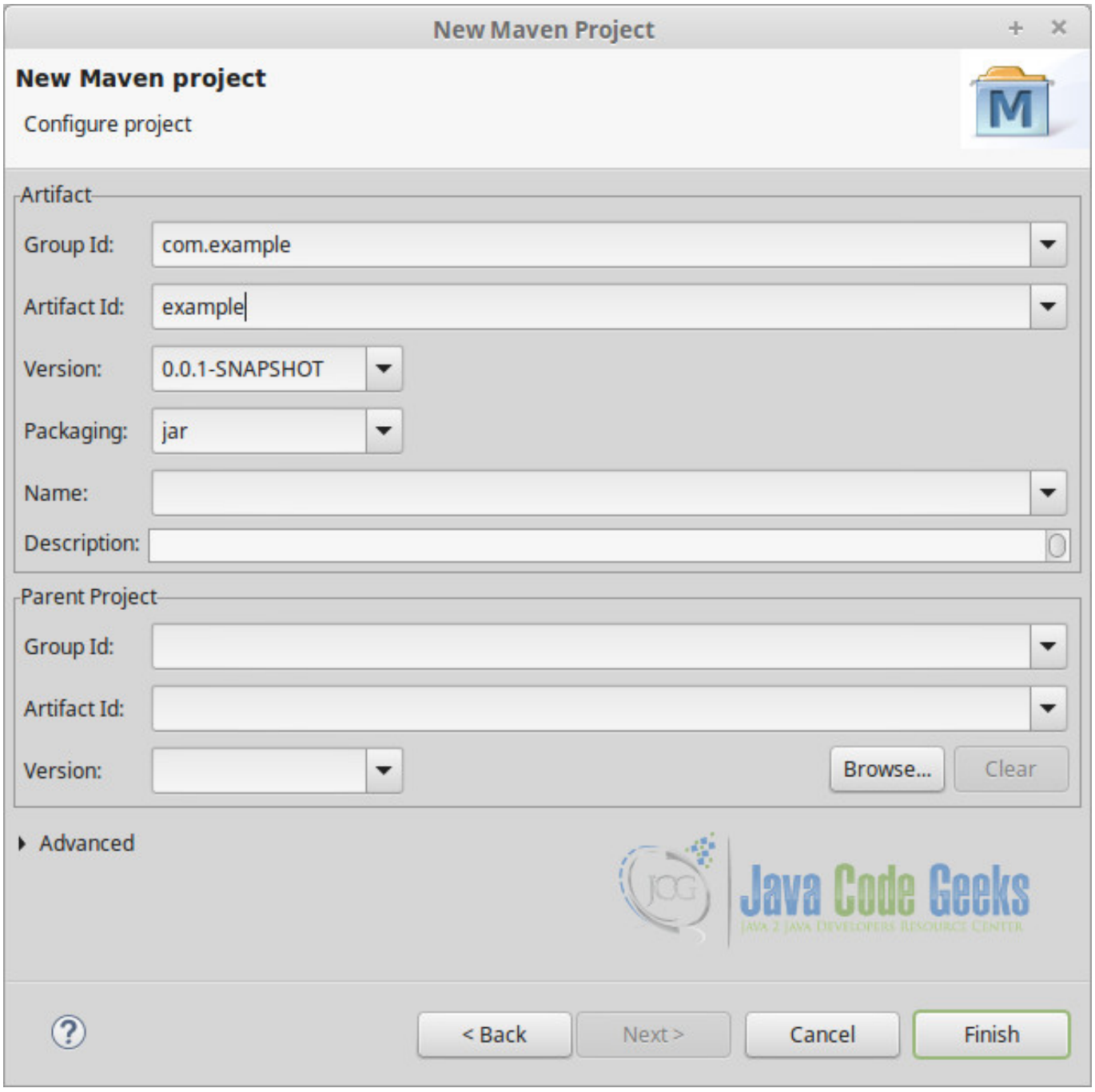

Figure 3.2: Configure Maven Project

Click pom.xml in the project root folder in order to open up pom.xml editor, and add maven.compiler.source and maven.compiler.target properties with value 1.8 into it.

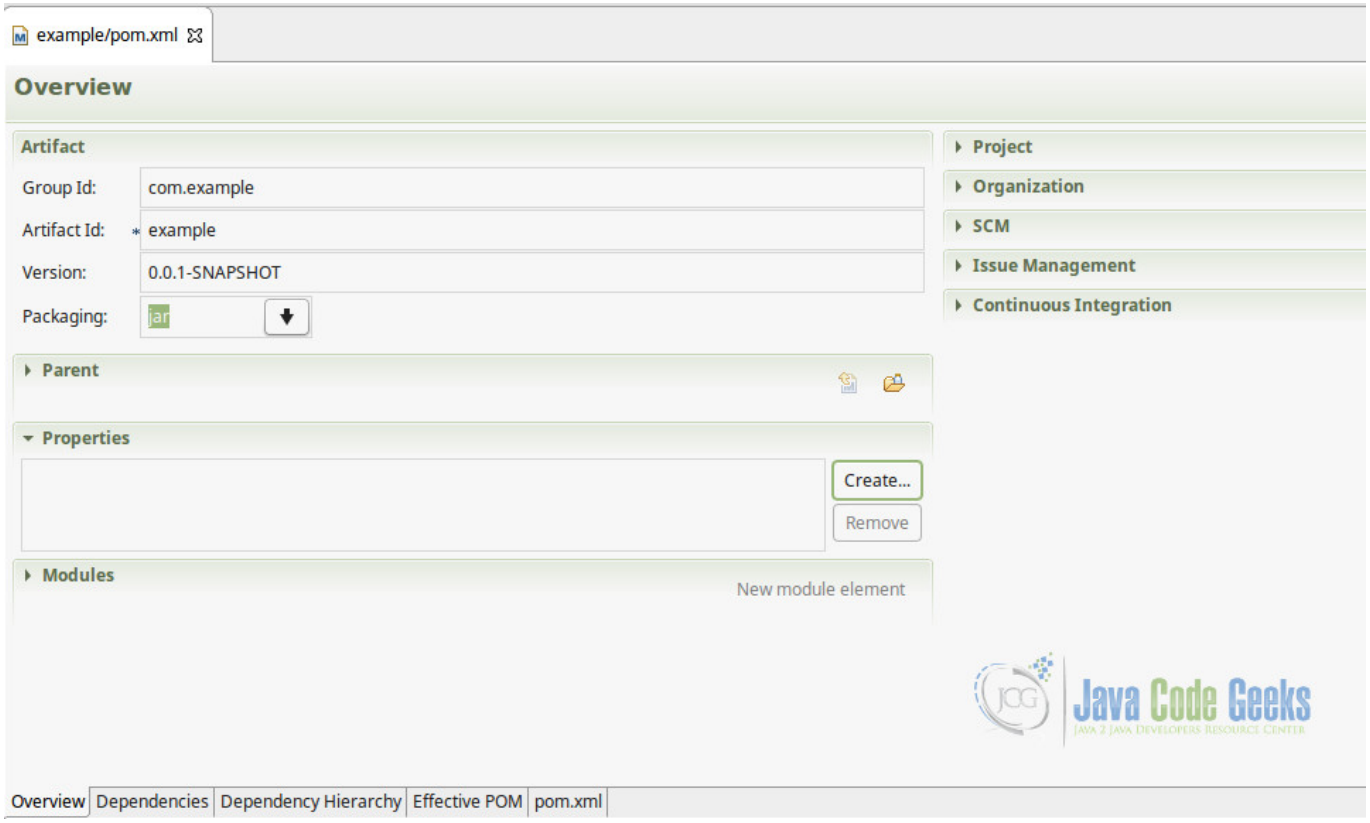

Figure 3.3: Click pom.xml to edit

3.2 Add necessary dependencies in your project

Add following dependencies into your pom.xml. You can make use of pom.xml editor you opened up in the previous step.

```
<dependency>
                <groupId>com.h2database</groupId>
                <artifactId>h2</artifactId>
                <version>1.4.192</version>
        </dependency>
        <dependency>
                <groupId>org.springframework</groupId>
                <artifactId>spring-test</artifactId>
                <version>4.3.1.RELEASE</version>
        </dependency>
        <dependency>
                <groupId>org.springframework</groupId>
                <artifactId>spring-context</artifactId>
                <version>4.3.1.RELEASE</version>
        </dependency>
        <dependency>
                <groupId>org.springframework</groupId>
                <artifactId>spring-jdbc</artifactId>
                <version>4.3.1.RELEASE</version>
        </dependency>
        <dependency>
                <groupId>log4j</groupId>
                <artifactId>log4j</artifactId>
                <version>1.2.17</version>
```

```
        </dependency>
        <dependency>
                <groupId>junit</groupId>
                <artifactId>junit</artifactId>
                <version>4.12</version>
                <scope>test</scope>
        </dependency>
```

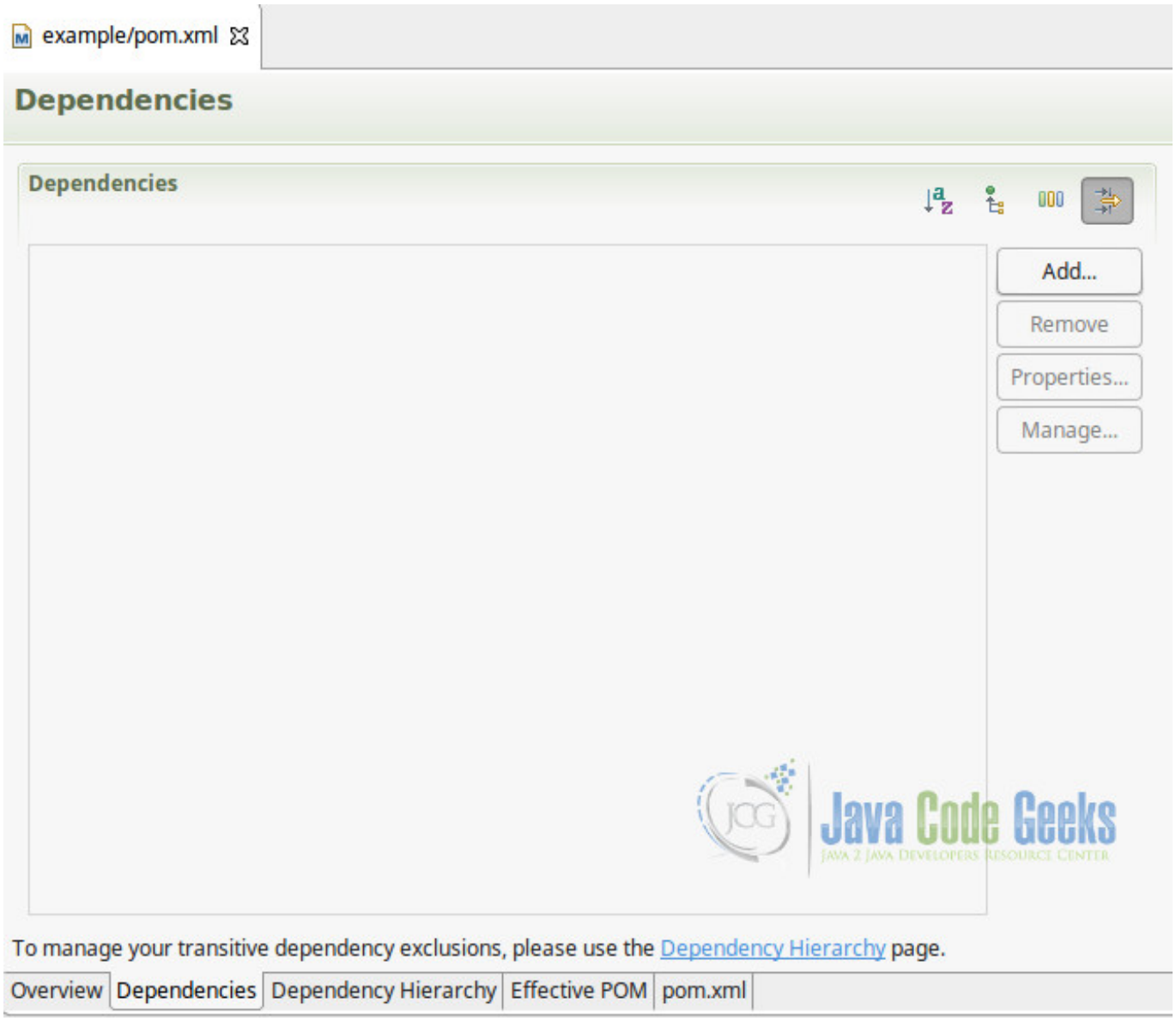

Figure 3.4: Add Maven Dependencies

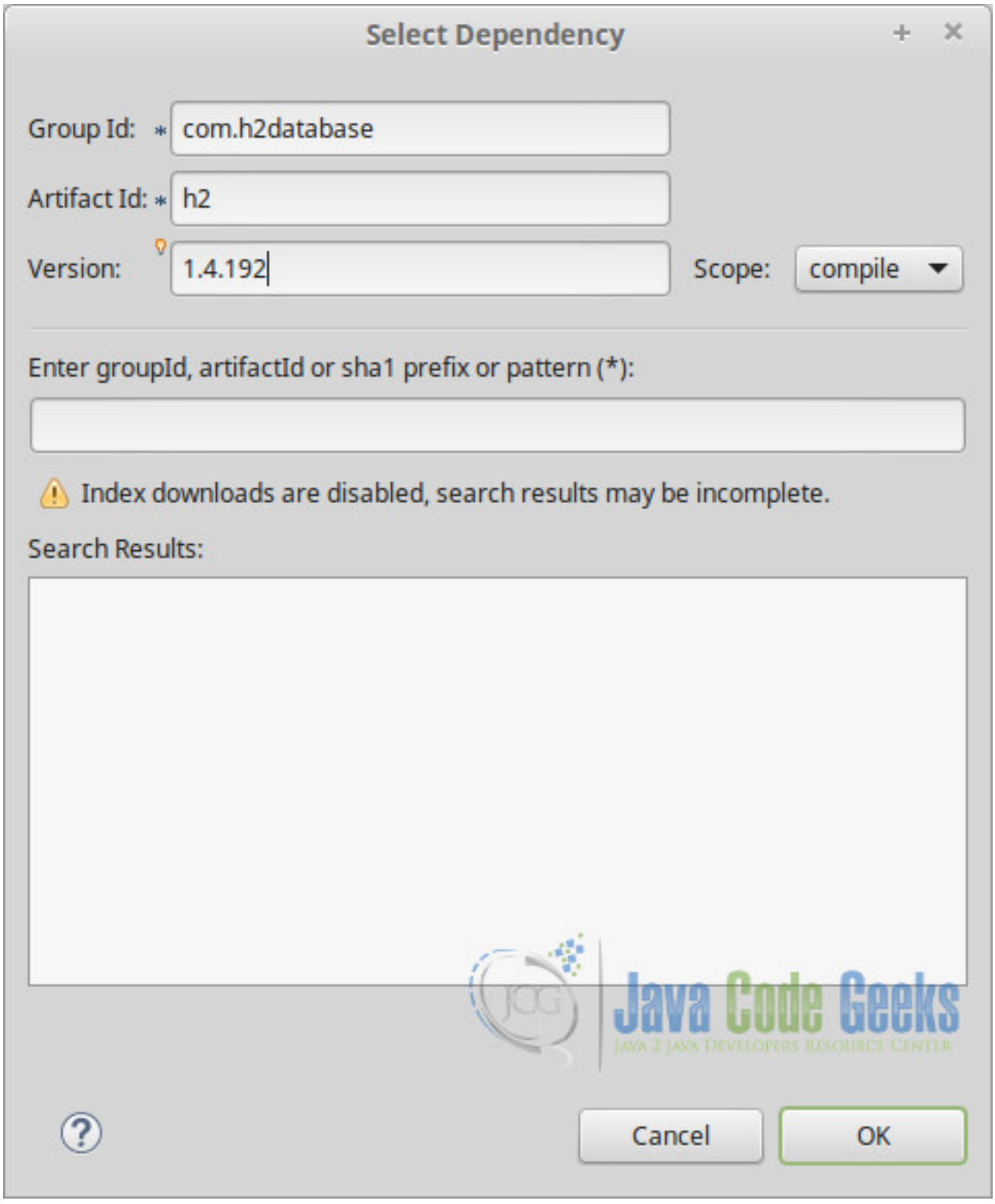

Figure 3.5: Add a New Dependency

You can either add those dependencies via add Dependency dialog, or switch into source view of pom.xml and copy all of them into section. After this step, added dependencies should have been listed as follows.

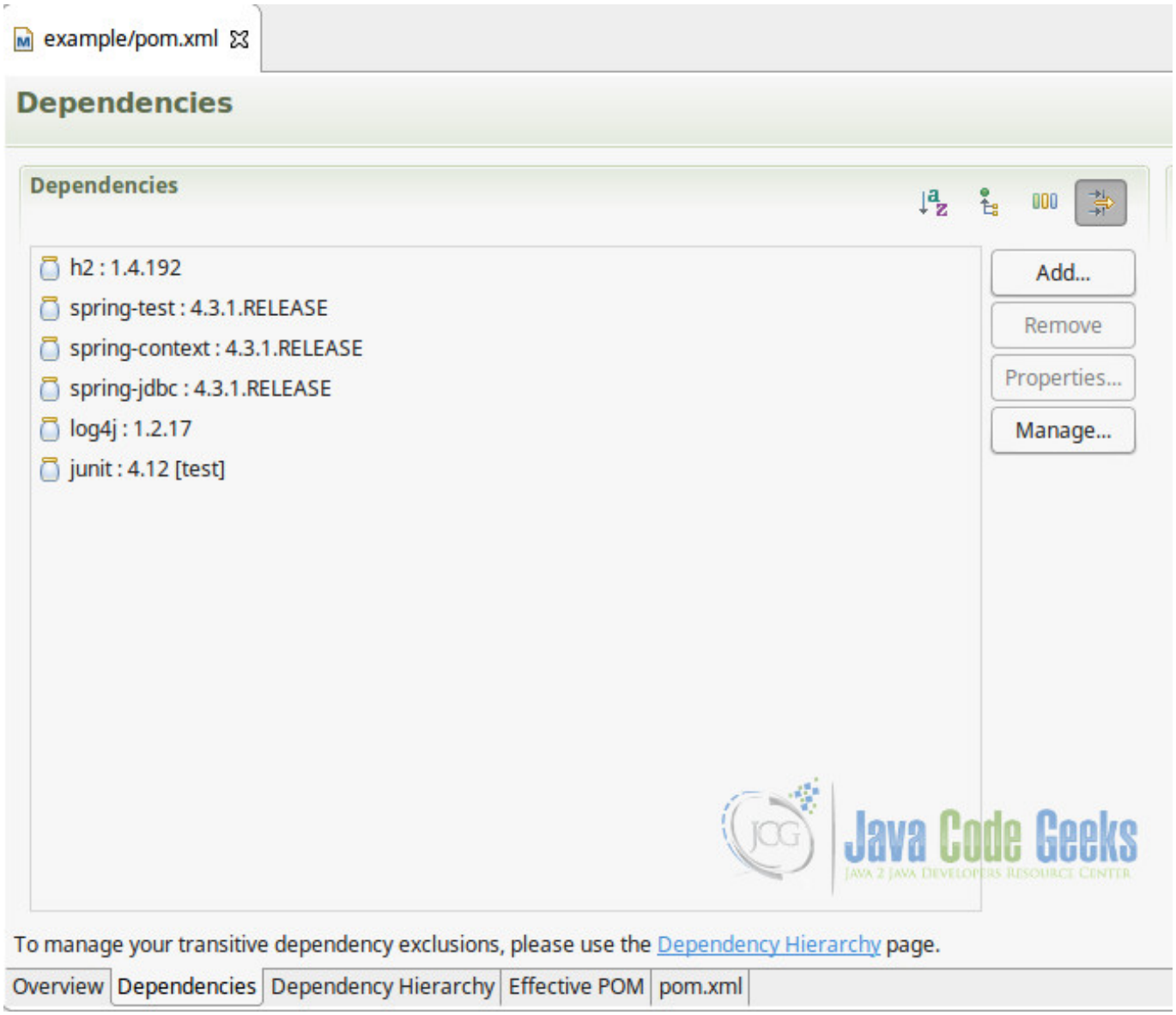

Figure 3.6: Maven Added Dependencies

Finally perform a project update by right clicking the project and then clicking “Update Project” through Maven>Update Project. . .

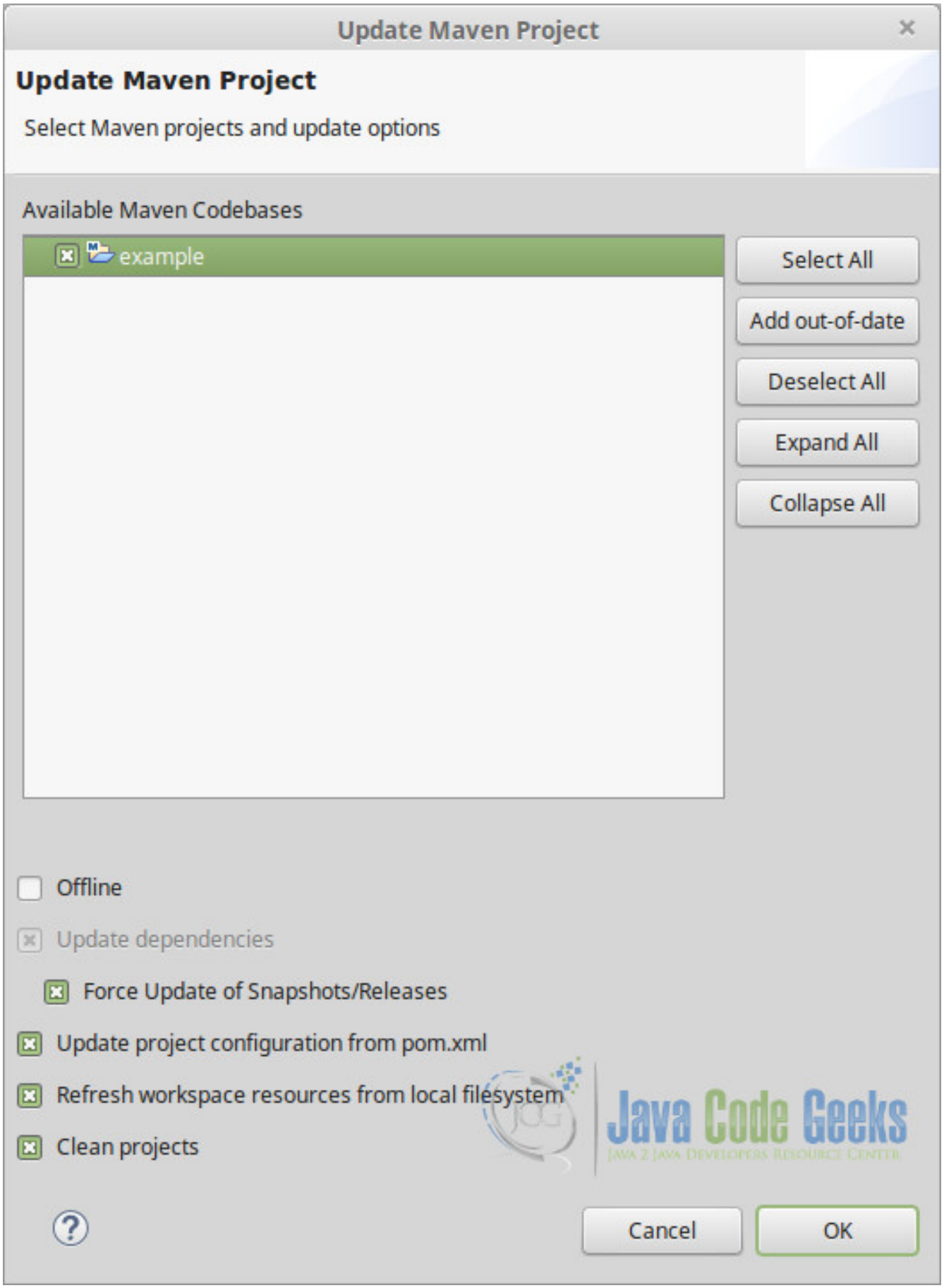

Figure 3.7: Update Maven Project

At this point, you are ready to work within the project.

3.3 Create log4j.xml file in your project

The first step is to create log4j.xml file under src/main/resources folder with the following content. It will help us to see log messages produced by Spring during execution of test methods and trace what is going on during those executions.

log4j.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE log4j:configuration PUBLIC "-//LOG4J" "log4j.dtd">
<log4j:configuration xmlns:log4j="https://jakarta.apache.org/log4j/">

        <appender name="CONSOLE" class="org.apache.log4j.ConsoleAppender">
                <layout class="org.apache.log4j.EnhancedPatternLayout">
                        <param name="ConversionPattern"
                                value="%d{HH:mm:ss,SSS} - %p - %C{1.}.%M(%L): %m%n" />
                </layout>
        </appender>

        <logger name="org.springframework">
                <level value="DEBUG" />
        </logger>

        <root>
                <level value="INFO" />

                <appender-ref ref="CONSOLE" />

        </root>

</log4j:configuration>
```

3.4 Prepare DDL and DML scripts to initialize database

Create schema.sql and data.sql files within src/main/resources with the following contents.

schema.sql

```
CREATE SEQUENCE PUBLIC.T_PERSON_SEQUENCE START WITH 1;

CREATE CACHED TABLE PUBLIC.T_PERSON(
    ID BIGINT NOT NULL,
    FIRST_NAME VARCHAR(255),
    LAST_NAME VARCHAR(255)
);
ALTER TABLE PUBLIC.T_PERSON ADD CONSTRAINT PUBLIC.CONSTRAINT_PERSON_PK PRIMARY KEY(ID);
```

data.sql

```
INSERT INTO T_PERSON (ID,FIRST_NAME,LAST_NAME) VALUES (T_PERSON_SEQUENCE.NEXTVAL, 'John',' ↩
    Doe');
INSERT INTO T_PERSON (ID,FIRST_NAME,LAST_NAME) VALUES (T_PERSON_SEQUENCE.NEXTVAL, 'Joe',' ↩
    Doe');
```

3.5 Write Domain Class, Service and DAO Beans

We are going to create a simple domain class with name Person as follows. It has only three attributes, id, firstName and lastName, and accessor methods for them.

Person.java

```
package com.example.model;

public class Person {
        private Long id;
        private String firstName;
        private String lastName;
        public Long getId() {
                return id;
        }
        public void setId(Long id) {
                this.id = id;
        }
        public String getFirstName() {
                return firstName;
        }
        public void setFirstName(String firstName) {
                this.firstName = firstName;
        }
        public String getLastName() {
                return lastName;
        }
        public void setLastName(String lastName) {
                this.lastName = lastName;
        }
}
```

We also create Service and DAO classes as follows, in order to perform simple persistence operations with our domain model. package com.example.dao;

PersonDao.java

```
import com.example.model.Person;

public interface PersonDao {
        public Person findById(Long id);
        public void create(Person person);
        public void update(Person person);
        public void delete(Long id);
}
```

PersonDao is a simple interface which defines basic persistence operations over Person instances like findById, create a new Person, update or delete an existing one.

JdbcPersonDao.java

```
package com.example.dao;

import java.sql.ResultSet;
import java.sql.SQLException;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.RowMapper;
import org.springframework.stereotype.Repository;
```

```
import com.example.model.Person;

@Repository
public class JdbcPersonDao implements PersonDao {
        private JdbcTemplate jdbcTemplate;

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

        @Override
        public Person findById(Long id) {
                return jdbcTemplate.queryForObject("select first_name, last_name from  ↩
                    t_person where id = ?",
                                new RowMapper() {

                                        @Override
                                        public Person mapRow(ResultSet rs, int rowNum)  ↩
                                            throws SQLException {
                                                Person person = new Person();
                                                person.setId(id);
                                                person.setFirstName(rs.getString(" ↩
                                                    first_name"));
                                                person.setLastName(rs.getString("last_name" ↩
                                                    ));
                                                return person;
                                        }
                                }, id);
        }

        @Override
        public void create(Person person) {
                jdbcTemplate.update("insert into t_person(id,first_name,last_name) values( ↩
                    t_person_sequence.nextval,?,?)",
                                person.getFirstName(), person.getLastName());
        }

        @Override
        public void update(Person person) {
                jdbcTemplate.update("update t_person set first_name = ?, last_name = ?  ↩
                    where id = ?", person.getFirstName(),
                                person.getLastName(), person.getId());
        }

        @Override
        public void delete(Long id) {
                jdbcTemplate.update("delete from t_person where id = ?", id);
        }
}
```

JdbcPersonDao is an implementation of PersonDao interface which employs NamedParameterJdbcTemplate bean of Spring in order to implement persistence operations via JDBC API.

PersonService.java

```
package com.example.service;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;
```

```
import com.example.dao.PersonDao;
import com.example.model.Person;

@Service
@Transactional
public class PersonService {
        private PersonDao personDao;

        @Autowired
        public void setPersonDao(PersonDao personDao) {
                this.personDao = personDao;
        }

        public Person findById(Long id) {
                return personDao.findById(id);
        }

        public void create(Person person) {
                personDao.create(person);
        }

        public void update(Person person) {
                personDao.update(person);
        }

        public void delete(Long id) {
                personDao.delete(id);
        }
}
```

PersonService is a transactional service which uses PersonDao bean in order to perform persistence operations. Its role is simply delegating to its DAO bean apart from being transactional in this context.

3.6 Configure Spring ApplicationContext

Write click onver src/main/resources and create a new Spring Bean Definition File through “New>Spring Bean Configuration File”. Make sure you select context, tx and jdbc namespaces as you create the configuration file.

spring-beans.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="https://www.springframework.org/schema/beans"
        xmlns:xsi="https://www.w3.org/2001/XMLSchema-instance"
        xmlns:jdbc="https://www.springframework.org/schema/jdbc"
        xmlns:tx="https://www.springframework.org/schema/tx"
        xmlns:context="https://www.springframework.org/schema/context"
        xsi:schemaLocation="https://www.springframework.org/schema/jdbc https://www. ↩
            springframework.org/schema/jdbc/spring-jdbc-4.3.xsd
                https://www.springframework.org/schema/beans https://www.springframework. ↩
                    org/schema/beans/spring-beans.xsd
                https://www.springframework.org/schema/context https://www.springframework. ↩
                    org/schema/context/spring-context-4.3.xsd
                https://www.springframework.org/schema/tx https://www.springframework.org/ ↩
                    schema/tx/spring-tx-4.3.xsd">

        <context:component-scan base-package="com.example"/>

        <tx:annotation-driven/>
```

```
        <bean id="transactionManager" class="org.springframework.jdbc.datasource. ↩
            DataSourceTransactionManager">
                <property name="dataSource" ref="dataSource"/>
        </bean>

        <jdbc:embedded-database type="H2" id="dataSource">
                <jdbc:script location="classpath:/schema.sql"/>
                <jdbc:script location="classpath:/data.sql"/>
        </jdbc:embedded-database>

        <bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">
                <property name="dataSource" ref="dataSource"/>
        </bean>
</beans>
```

3.7 Write a transactional integration unit test

PersonServiceIntegrationTests.java

```
package com.example;

import org.junit.Assert;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;
import org.springframework.transaction.annotation.Transactional;

import com.example.model.Person;
import com.example.service.PersonService;

@RunWith(SpringJUnit4ClassRunner.class)
@Transactional
@ContextConfiguration("classpath:/spring-beans.xml")
public class PersonServiceIntegrationTests {
        @Autowired
        private PersonService personService;

        @Autowired
        private JdbcTemplate jdbcTemplate;


        @Test
        public void shouldCreateNewPerson() {
                Person person = new Person();
                person.setFirstName("Kenan");
                person.setLastName("Sevindik");

                long countBeforeInsert = jdbcTemplate.queryForObject("select count(*) from  ↩
                    t_person", Long.class);
                Assert.assertEquals(2, countBeforeInsert);

                personService.create(person);

                long countAfterInsert = jdbcTemplate.queryForObject("select count(*) from  ↩
                    t_person", Long.class);
                Assert.assertEquals(3, countAfterInsert);
```

```
        }

        @Test
        public void shouldDeleteNewPerson() {
                long countBeforeDelete = jdbcTemplate.queryForObject("select count(*) from ↩
                    t_person", Long.class);
                Assert.assertEquals(2, countBeforeDelete);

                personService.delete(1L);

                long countAfterDelete = jdbcTemplate.queryForObject("select count(*) from ↩
                    t_person", Long.class);
                Assert.assertEquals(1, countAfterDelete);
        }

        @Test
        public void shouldFindPersonsById() {
                Person person = personService.findById(1L);

                Assert.assertNotNull(person);
                Assert.assertEquals("John", person.getFirstName());
                Assert.assertEquals("Doe", person.getLastName());
        }
}
```

Above test methods test creation of a new Person instance, deletion of an existing one and finding by its id. @RunWith annotation belongs to Junit, and is used to tell IDE which Runner class, SpringJUnit4ClassRunner.class in this case, to use to run test methods defined in the class. SpringJUnit4ClassRunner creates an ApplicationContext by loading Spring bean configuration files listed in @ContextConfiguration("classpath:/spring-beans.xml") annotation. It is possible to use Java Configuration classes as well, however, I preferred to follow classical XML way in this example. After creation of ApplicationContext, dependencies specified in the test class are autowired for use within test methods. @Transactional annotation tells SpringJUnit4ClassRunner that all test methods defined in this class must be run within an active transaction context.

Therefore, SpringJUnit4ClassRunner starts a new transaction at the beginning of each test method execution, and then rolls back it at the end. The reason to rollback instead of commit is that those changes performed on the database within each test method should not adversely affect execution of other integration tests. However, any service method call which expects an active transaction to work during its execution is satisfied with that active transaction spanning the test method. It is possible to see how transaction is created and then rolled back from the log messages shown below.

3.8 Run the tests and observe the results

Right click over the test class, and run it with JUnit. You should have seen all JUnit tests passed as follows.

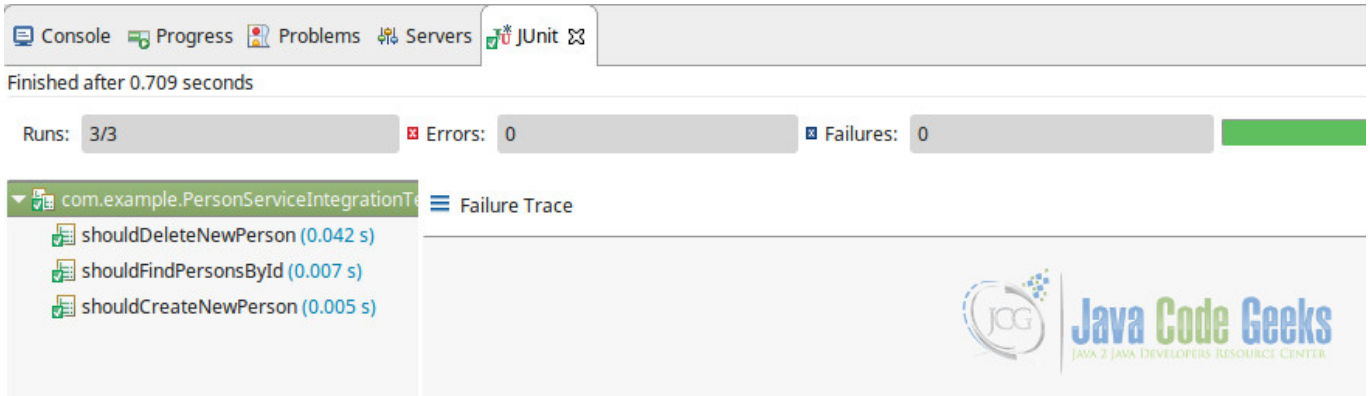

Figure 3.8: JUnit Test Results

When you click over the console tab, you should have seen log messages similar to the following.

```
17:51:24,230 - DEBUG - o.s.t.c.t.TransactionalTestExecutionListener.beforeTestMethod(183):  ↩
    Explicit transaction definition [PROPAGATION_REQUIRED,ISOLATION_DEFAULT; ''] found for  ↩
    test context [DefaultTestContext@3e6fa38a testClass = PersonServiceIntegrationTests,  ↩
    testInstance = com.example.PersonServiceIntegrationTests@6a4f787b, testMethod =  ↩
    shouldCreateNewPerson@PersonServiceIntegrationTests, testException = [null],  ↩
    mergedContextConfiguration = [MergedContextConfiguration@66a3ffec testClass =  ↩
    PersonServiceIntegrationTests, locations = '{classpath:/spring-beans.xml}', classes = ' ↩
    {}', contextInitializerClasses = '[]', activeProfiles = '{}', propertySourceLocations =  ↩
    '{}', propertySourceProperties = '{}', contextCustomizers = set[[empty]], contextLoader  ↩
    = 'org.springframework.test.context.support.DelegatingSmartContextLoader', parent = [ ↩
    null]]]
17:51:24,230 - DEBUG - o.s.t.c.t.TransactionalTestExecutionListener. ↩
    retrieveConfigurationAttributes(476): Retrieved @TransactionConfiguration [null] for  ↩
    test class [com.example.PersonServiceIntegrationTests].
17:51:24,230 - DEBUG - o.s.t.c.t.TransactionalTestExecutionListener. ↩
    retrieveConfigurationAttributes(483): Using TransactionConfigurationAttributes [ ↩
    TransactionConfigurationAttributes@5167f57d transactionManagerName = '', defaultRollback ↩
     = true] for test class [com.example.PersonServiceIntegrationTests].
17:51:24,230 - DEBUG - o.s.t.c.c.DefaultCacheAwareContextLoaderDelegate.loadContext(129):  ↩
    Retrieved ApplicationContext from cache with key [[MergedContextConfiguration@66a3ffec  ↩
    testClass = PersonServiceIntegrationTests, locations = '{classpath:/spring-beans.xml}',  ↩
    classes = '{}', contextInitializerClasses = '[]', activeProfiles = '{}',  ↩
    propertySourceLocations = '{}', propertySourceProperties = '{}', contextCustomizers =  ↩
    set[[empty]], contextLoader = 'org.springframework.test.context.support. ↩
    DelegatingSmartContextLoader', parent = [null]]]
17:51:24,230 - DEBUG - o.s.t.c.c.DefaultContextCache.logStatistics(290): Spring test  ↩
    ApplicationContext cache statistics: [DefaultContextCache@2fb0623e size = 1, maxSize =  ↩
    32, parentContextCount = 0, hitCount = 1, missCount = 1]
17:51:24,231 - DEBUG - o.s.b.f.s.AbstractBeanFactory.doGetBean(251): Returning cached  ↩
    instance of singleton bean 'transactionManager'
17:51:24,231 - DEBUG - o.s.t.c.t.TransactionalTestExecutionListener.isRollback(426): No  ↩
    method-level @Rollback override: using default rollback [true] for test context [ ↩
    DefaultTestContext@3e6fa38a testClass = PersonServiceIntegrationTests, testInstance =  ↩
    com.example.PersonServiceIntegrationTests@6a4f787b, testMethod =  ↩
    shouldCreateNewPerson@PersonServiceIntegrationTests, testException = [null],  ↩
    mergedContextConfiguration = [MergedContextConfiguration@66a3ffec testClass =  ↩
    PersonServiceIntegrationTests, locations = '{classpath:/spring-beans.xml}', classes = ' ↩
    {}', contextInitializerClasses = '[]', activeProfiles = '{}', propertySourceLocations =  ↩
    '{}', propertySourceProperties = '{}', contextCustomizers = set[[empty]], contextLoader  ↩
    = 'org.springframework.test.context.support.DelegatingSmartContextLoader', parent = [ ↩
    null]]].
17:51:24,232 - DEBUG - o.s.t.s.AbstractPlatformTransactionManager.getTransaction(367):  ↩
    Creating new transaction with name [com.example.PersonServiceIntegrationTests. ↩
    shouldCreateNewPerson]: PROPAGATION_REQUIRED,ISOLATION_DEFAULT; ''
17:51:24,233 - DEBUG - o.s.j.d.SimpleDriverDataSource.getConnectionFromDriver(138):  ↩
    Creating new JDBC Driver Connection to [jdbc:h2:mem:dataSource;DB_CLOSE_DELAY=-1; ↩
    DB_CLOSE_ON_EXIT=false]
17:51:24,233 - DEBUG - o.s.j.d.DataSourceTransactionManager.doBegin(206): Acquired  ↩
    Connection [conn1: url=jdbc:h2:mem:dataSource user=SA] for JDBC transaction
17:51:24,234 - DEBUG - o.s.j.d.DataSourceTransactionManager.doBegin(223): Switching JDBC  ↩
    Connection [conn1: url=jdbc:h2:mem:dataSource user=SA] to manual commit
17:51:24,234 - INFO - o.s.t.c.t.TransactionContext.startTransaction(101): Began transaction ↩
     (1) for test context [DefaultTestContext@3e6fa38a testClass =  ↩
    PersonServiceIntegrationTests, testInstance = com.example. ↩
    PersonServiceIntegrationTests@6a4f787b, testMethod =  ↩
    shouldCreateNewPerson@PersonServiceIntegrationTests, testException = [null],  ↩
    mergedContextConfiguration = [MergedContextConfiguration@66a3ffec testClass =  ↩
    PersonServiceIntegrationTests, locations = '{classpath:/spring-beans.xml}', classes = ' ↩
    {}', contextInitializerClasses = '[]', activeProfiles = '{}', propertySourceLocations =  ↩
    '{}', propertySourceProperties = '{}', contextCustomizers = set[[empty]], contextLoader  ↩
```

```
    = 'org.springframework.test.context.support.DelegatingSmartContextLoader', parent = [ ↩
    null]]]; transaction manager [org.springframework.jdbc.datasource. ↩
    DataSourceTransactionManager@2eea88a1]; rollback [true]
17:51:24,236 - DEBUG - o.s.j.c.JdbcTemplate.query(451): Executing SQL query [select count ↩
    (*) from t_person]
17:51:24,253 - DEBUG - o.s.b.f.s.AbstractBeanFactory.doGetBean(251): Returning cached  ↩
    instance of singleton bean 'transactionManager'
17:51:24,253 - DEBUG - o.s.t.s.AbstractPlatformTransactionManager.handleExistingTransaction ↩
    (476): Participating in existing transaction
17:51:24,273 - DEBUG - o.s.j.c.JdbcTemplate.update(869): Executing prepared SQL update
17:51:24,274 - DEBUG - o.s.j.c.JdbcTemplate.execute(616): Executing prepared SQL statement  ↩
    [insert into t_person(id,first_name,last_name) values(t_person_sequence.nextval,?,?)]
17:51:24,279 - DEBUG - o.s.j.c.JdbcTemplate$2.doInPreparedStatement(879): SQL update  ↩
    affected 1 rows
17:51:24,279 - DEBUG - o.s.j.c.JdbcTemplate.query(451): Executing SQL query [select count ↩
    (*) from t_person]
17:51:24,281 - DEBUG - o.s.t.s.AbstractPlatformTransactionManager.processRollback(851):  ↩
    Initiating transaction rollback
17:51:24,281 - DEBUG - o.s.j.d.DataSourceTransactionManager.doRollback(284): Rolling back  ↩
    JDBC transaction on Connection [conn1: url=jdbc:h2:mem:dataSource user=SA]
17:51:24,283 - DEBUG - o.s.j.d.DataSourceTransactionManager.doCleanupAfterCompletion(327):  ↩
    Releasing JDBC Connection [conn1: url=jdbc:h2:mem:dataSource user=SA] after transaction
17:51:24,283 - DEBUG - o.s.j.d.DataSourceUtils.doReleaseConnection(327): Returning JDBC  ↩
    Connection to DataSource
17:51:24,283 - INFO - o.s.t.c.t.TransactionContext.endTransaction(136): Rolled back  ↩
    transaction for test context [DefaultTestContext@3e6fa38a testClass =  ↩
    PersonServiceIntegrationTests, testInstance = com.example. ↩
    PersonServiceIntegrationTests@6a4f787b, testMethod =  ↩
    shouldCreateNewPerson@PersonServiceIntegrationTests, testException = [null],  ↩
    mergedContextConfiguration = [MergedContextConfiguration@66a3ffec testClass =  ↩
    PersonServiceIntegrationTests, locations = '{classpath:/spring-beans.xml}', classes = ' ↩
    {}', contextInitializerClasses = '[]', activeProfiles = '{}', propertySourceLocations =  ↩
    '{}', propertySourceProperties = '{}', contextCustomizers = set[[empty]], contextLoader  ↩
    = 'org.springframework.test.context.support.DelegatingSmartContextLoader', parent = [ ↩
    null]]].
```

Sometimes, you may need test executions to commit, instead of rollback so that you can connect to database, and observe the changes performed there, or you may employ integration unit tests for populating database with sample data. You can place either @Rollback(false) or @Commit annotations either on method or class level so that transaction commits instead of rollback.

3.9 Summary

In this example, we created a maven project, implemented several classes to perform persistence operations using JDBC API within it, and wrote an integration unit test in order to check whether those classes perform necessary persistence related operations as expected within an active transaction.

3.10 Download the Source Code

Download

You can download the full source code of this example here: **HowToWriteTransactionalTestsInSpring**

Chapter 4

Spring Framework JMSTemplate Example

In order to send or receive messages through JMS, we need a connection to JMS provider, obtain session, create destination creation, the JMS API involved becomes too verbose and repetitive. `JmsTemplate` is a helper class that simplifies receiving and sending of messages through JMS and gets rid of the boilerplate code.

`JmsTemplate` simplifies the development efforts on constructing the message to send or processing messages that are received through synchronous JMS access code.

Let's start with a simple example and then re-factor it to use `JmsTemplate`

4.1 Dependencies

In order to send and receive JMS messages to and from a JMS message broker, we need to include the message service library. In this example we are using activeMq so our pom.xml will have dependencies related to spring as well as activeMQ.

pom.xml:

```
<project xmlns="https://maven.apache.org/POM/4.0.0" xmlns:xsi="https://www.w3.org/2001/ ↩
    XMLSchema-instance"
        xsi:schemaLocation="https://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd ↩
            /maven-4.0.0.xsd">
        <modelVersion>4.0.0</modelVersion>
        <groupId>com.javacodegeeks.camel</groupId>
        <artifactId>springQuartzScheduler</artifactId>
        <version>0.0.1-SNAPSHOT</version>
        <dependencies>
                <dependency>
                        <groupId>org.springframework</groupId>
                        <artifactId>spring-core</artifactId>
                        <version>4.1.5.RELEASE</version>
                </dependency>
                <dependency>
                        <groupId>org.springframework</groupId>
                        <artifactId>spring-context</artifactId>
                        <version>4.1.5.RELEASE</version>
                </dependency>
                <dependency>
                        <groupId>org.springframework</groupId>
                        <artifactId>spring-jms</artifactId>
                        <version>4.1.5.RELEASE</version>
                </dependency>
                <dependency>
                        <groupId>org.apache.activemq</groupId>
                        <artifactId>activemq-all</artifactId>
```

```
                        <version>5.12.0</version>
                </dependency>
        </dependencies>

</project>
```

4.2 Sending and Receiving Messages without JmsTemplate

We will first start with an example of producer and consumer that works without the use of JMS Template.

We first need to start the broker. We are using ActiveMQ which acts as the JMS Provider.

BrokerLauncher:

```
package com.javacodegeeks.spring.jms;

import java.net.URI;
import java.net.URISyntaxException;

import org.apache.activemq.broker.BrokerFactory;
import org.apache.activemq.broker.BrokerService;

public class BrokerLauncher {
        public static void main(String[] args) throws URISyntaxException, Exception {
                BrokerService broker = BrokerFactory.createBroker(new URI(
                                "broker:(tcp://localhost:61616)"));
                broker.start();
        }
}
```

Output:

```
INFO | JMX consoles can connect to service:jmx:rmi:///jndi/rmi://localhost:1099/jmxrmi
 INFO | PListStore:[C:\javacodegeeks_ws\springJmsTemplateExample\activemq-data\localhost\ ↩
    tmp_storage] started
 INFO | Using Persistence Adapter: KahaDBPersistenceAdapter[C:\javacodegeeks_ws\ ↩
    springJmsTemplateExample\activemq-data\localhost\KahaDB]
 INFO | Apache ActiveMQ 5.12.0 (localhost, ID:INMAA1-L1005-59525-1448470360347-0:1) is  ↩
    starting
 INFO | Listening for connections at: tcp://127.0.0.1:61616
 INFO | Connector tcp://127.0.0.1:61616 started
 INFO | Apache ActiveMQ 5.12.0 (localhost, ID:INMAA1-L1005-59525-1448470360347-0:1) started
 INFO | For help or more information please see: https://activemq.apache.org
 WARN | Store limit is 102400 mb (current store usage is 0 mb). The data directory: C:\ ↩
    javacodegeeks_ws\springJmsTemplateExample\activemq-data\localhost\KahaDB only has 29337 ↩
     mb of usable space - resetting to maximum available disk space: 29337 mb
 WARN | Temporary Store limit is 51200 mb, whilst the temporary data directory: C:\ ↩
    javacodegeeks_ws\springJmsTemplateExample\activemq-data\localhost\tmp_storage only has  ↩
    29337 mb of usable space - resetting to maximum available 29337 mb.
```

Here is the producer bean. You can see we need to create connection factory, get the connection, session, create destination etc.

JmsProducer:

```
package com.javacodegeeks.spring.jms;

import java.net.URISyntaxException;

import javax.jms.Connection;
import javax.jms.ConnectionFactory;
```

```
import javax.jms.Message;
import javax.jms.MessageProducer;
import javax.jms.Queue;
import javax.jms.Session;

import org.apache.activemq.ActiveMQConnectionFactory;

public class JmsProducer {
        public static void main(String[] args) throws URISyntaxException, Exception {
                Connection connection = null;
                try {
                        // Producer
                        ConnectionFactory connectionFactory = new ActiveMQConnectionFactory ↩
                            (
                                        "tcp://localhost:61616");
                        connection = connectionFactory.createConnection();
                        Session session = connection.createSession(false,
                                        Session.AUTO_ACKNOWLEDGE);
                        Queue queue = session.createQueue("customerQueue");
                        MessageProducer producer = session.createProducer(queue);
                        String payload = "SomeTask";
                        Message msg = session.createTextMessage(payload);
                        System.out.println("Sending text '" + payload + "'");
                        producer.send(msg);
                        session.close();
                } finally {
                        if (connection != null) {
                                connection.close();
                        }
                }
        }
}
```

Output:

```
Sending text 'SomeTask'
```

Consumer also needs a connection factory, connection, session and destination objects just like its counterpart.

JmsConsumer:

```
package com.javacodegeeks.spring.jms;

import java.net.URISyntaxException;

import javax.jms.Connection;
import javax.jms.ConnectionFactory;
import javax.jms.MessageConsumer;
import javax.jms.Queue;
import javax.jms.Session;
import javax.jms.TextMessage;

import org.apache.activemq.ActiveMQConnectionFactory;

public class JmsConsumer {
        public static void main(String[] args) throws URISyntaxException, Exception {
                Connection connection = null;
                ConnectionFactory connectionFactory = new ActiveMQConnectionFactory(
                                "tcp://localhost:61616");
                connection = connectionFactory.createConnection();
                connection.start();
                Session session = connection.createSession(false,
```

```
                                Session.AUTO_ACKNOWLEDGE);
                try {
                        Queue queue = session.createQueue("customerQueue");

                        // Consumer
                        MessageConsumer consumer = session.createConsumer(queue);
                        TextMessage textMsg = (TextMessage) consumer.receive();
                        System.out.println(textMsg);
                        System.out.println("Received: " + textMsg.getText());
                } finally {
                        if (session != null) {
                                session.close();
                        }
                        if (connection != null) {
                                connection.close();
                        }
                }
        }
}
```

Output:

```
ActiveMQTextMessage {commandId = 5, responseRequired = true, messageId = ID:INMAA1-L1005 ↩
    -59616-1448470447765-1:1:1:1:1, originalDestination = null, originalTransactionId = null ↩
    , producerId = ID:INMAA1-L1005-59616-1448470447765-1:1:1:1, destination = queue:// ↩
    customerQueue, transactionId = null, expiration = 0, timestamp = 1448470448008, arrival ↩
    = 0, brokerInTime = 1448470448010, brokerOutTime = 1448470613044, correlationId = null, ↩
    replyTo = null, persistent = true, type = null, priority = 4, groupID = null, ↩
    groupSequence = 0, targetConsumerId = null, compressed = false, userID = null, content = ↩
     org.apache.activemq.util.ByteSequence@d7b1517, marshalledProperties = null, ↩
    dataStructure = null, redeliveryCounter = 1, size = 0, properties = null, ↩
    readOnlyProperties = true, readOnlyBody = true, droppable = false, ↩
    jmsXGroupFirstForConsumer = false, text = SomeTask}
Received: SomeTask
```

4.3 Configuring JmsTemplate

`JmsTemplate` takes care of creating a connection, obtaining a session, and the actual sending and receiving of messages. Let's configure `JmsTemplate`.

To use JmsTemplate, we'll need to declare it as a bean in the Spring configuration XML.

```
<bean id="jmsTemplate" class="org.springframework.jms.core.JmsTemplate">
                <property name="connectionFactory" ref="connectionFactory" />
                <property name="receiveTimeout" value="10000" />
</bean>
```

`JmsTemplate` is only a helper class so it still needs to know how to get connections to the message broker.

`ConnectionFactory` bean is configured and `JmsTemplate` refers to the configured connection factory bean.

```
<bean id="connectionFactory" class="org.apache.activemq.ActiveMQConnectionFactory">
                <property name="brokerURL" value="tcp://localhost:61616" />
</bean>
```

applicationContext.xml:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="https://www.springframework.org/schema/beans"
        xmlns:xsi="https://www.w3.org/2001/XMLSchema-instance"
```

```
        xsi:schemaLocation="https://www.springframework.org/schema/beans https://www. ↩
            springframework.org/schema/beans/spring-beans.xsd">

        <bean id="connectionFactory" class="org.apache.activemq.ActiveMQConnectionFactory">
                <property name="brokerURL" value="tcp://localhost:61616" />
        </bean>
        <bean id="messageDestination" class="org.apache.activemq.command.ActiveMQQueue">
                <constructor-arg value="messageQueue1" />
        </bean>
        <bean id="jmsTemplate" class="org.springframework.jms.core.JmsTemplate">
                <property name="connectionFactory" ref="connectionFactory" />
                <property name="receiveTimeout" value="10000" />
        </bean>

        <bean id="springJmsProducer" class="com.javacodegeeks.spring.jms.SpringJmsProducer" ↩
            >
                <property name="destination" ref="messageDestination" />
                <property name="jmsTemplate" ref="jmsTemplate" />
        </bean>

        <bean id="springJmsConsumer" class="com.javacodegeeks.spring.jms.SpringJmsConsumer" ↩
            >
                <property name="destination" ref="messageDestination" />
                <property name="jmsTemplate" ref="jmsTemplate" />
        </bean>
</beans>
```

If you have noticed in the spring XML file above, we have also configured the producer and consumer bean. Both consumer and produce beans need `JmsTemplate` bean and the destination. JMS Destination is the queue the message will be sent to.

```
<bean id="messageDestination" class="org.apache.activemq.command.ActiveMQQueue">
                <constructor-arg value="messageQueue1" />
</bean>
```

The destination bean is injected through setter injection to both the producer and consumer beans.

4.4 Using JMSTemplate to produce messages

Let's now look into the producer bean's `sendMessage(msg)` method. It in turn calls `JmsTemplate.send()` method. The first parameter to the `send()` method is the name of the JMS Destination that the message will be sent to and the second parameter is an implementation of `MessageCreator` which contains the callback method `createMessage()` that `JmsTemplate` will use to construct the message that will be sent. Since `JmsTemplate` has access to the JMS provider's connection factory, it takes care of obtaining a JMS connection and session and will send the message on behalf of the sender.

SpringJmsProducer:

```
package com.javacodegeeks.spring.jms;

import javax.jms.Destination;
import javax.jms.JMSException;
import javax.jms.Message;
import javax.jms.Session;

import org.springframework.jms.core.JmsTemplate;
import org.springframework.jms.core.MessageCreator;

public class SpringJmsProducer {
        private JmsTemplate jmsTemplate;
        private Destination destination;
```

```
        public JmsTemplate getJmsTemplate() {
                return jmsTemplate;
        }

        public void setJmsTemplate(JmsTemplate jmsTemplate) {
                this.jmsTemplate = jmsTemplate;
        }

        public Destination getDestination() {
                return destination;
        }

        public void setDestination(Destination destination) {
                this.destination = destination;
        }

        public void sendMessage(final String msg) {
                System.out.println("Producer sends " + msg);
                jmsTemplate.send(destination, new MessageCreator() {
                        public Message createMessage(Session session) throws JMSException {
                                return session.createTextMessage(msg);
                        }});
        }
}
```

We have seen the producer, let's now look into the consumer code and see how we can make use of `JmsTemplate`.

4.5 Using JMSTemplate to consume messages

In order to receive the message, we need to call `JmsTemplate.receive(destination)` method which takes in the destination. One can also call just the `receive()` method without any destination in which case the default destination will be used. We will see in our next section how one can configure a default destination. `JmsTemplate` will make use of the connection factory to obtain the connection and session object.

`receive()` will block until a message appears on the destination, waiting forever. Its a good practice to specify a receive timeout instead so that `receive()` call returns back after the specified time out. `receiveTimeout` property is used to set the timeout.

```
<bean id="jmsTemplate" class="org.springframework.jms.core.JmsTemplate">
    <property name="connectionFactory" ref="connectionFactory" />
    <property name="receiveTimeout" value="10000" />
</bean>
```

SpringJmsConsumer:

```
package com.javacodegeeks.spring.jms;

import javax.jms.Destination;
import javax.jms.JMSException;
import javax.jms.TextMessage;

import org.springframework.jms.core.JmsTemplate;

public class SpringJmsConsumer {
        private JmsTemplate jmsTemplate;
        private Destination destination;

        public JmsTemplate getJmsTemplate() {
                return jmsTemplate;
        }
```

```
        public void setJmsTemplate(JmsTemplate jmsTemplate) {
                this.jmsTemplate = jmsTemplate;
        }

        public Destination getDestination() {
                return destination;
        }

        public void setDestination(Destination destination) {
                this.destination = destination;
        }

        public String receiveMessage() throws JMSException {
                TextMessage textMessage = (TextMessage) jmsTemplate.receive(destination);
                return textMessage.getText();
        }
}
```

4.6 Complete JmsTemplate example to send/receive messages

Let's now combine the producer and consumer to send and receive message.

- Make sure the broker is started.
- First we load the application context.
- Next, we get the producer bean from the spring container.
- We use the producer bean to send messages.
- Next, we load the consumer bean.
- We will then use the consumer bean to receive messages.

SpringJmsTemplateExample:

```
package com.javacodegeeks.spring.jms;

import java.net.URISyntaxException;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class SpringJmsTemplateExample {
        public static void main(String[] args) throws URISyntaxException, Exception {
                ClassPathXmlApplicationContext context = new ClassPathXmlApplicationContext ↩
                    (
                                "applicationContext.xml");

                try {
                        SpringJmsProducer springJmsProducer = (SpringJmsProducer) context
                                        .getBean("springJmsProducer");
                        springJmsProducer.sendMessage("SomeTask");

                        SpringJmsConsumer springJmsConsumer = (SpringJmsConsumer) context
                                        .getBean("springJmsConsumer");
                        System.out.println("Consumer receives " + springJmsConsumer. ↩
                            receiveMessage());
                } finally {
                        context.close();
```

```
                }
        }
}
```

Output:

```
Producer sends SomeTask
Consumer receives SomeTask
```

4.7 JmsTemplate with Default destination

If our scenario demands of a default destination then we can avoid explicitly injecting destination separately to each producer and consumer bean and instead inject it into `JmsTemplate` bean. We can do this using the property`defaultDestination`.

```
<bean id="jmsTemplate" class="org.springframework.jms.core.JmsTemplate">
    <property name="connectionFactory" ref="connectionFactory" />
    <property name="receiveTimeout" value="10000" />
    <property name="defaultDestination" ref="messageDestination" />
</bean>
```

We can remove the destination properties from the producer and consumer bean declarations.

appContextWithDefaultDestin.xml:

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

        <bean id="connectionFactory" class="org.apache.activemq.ActiveMQConnectionFactory">
                <property name="brokerURL" value="tcp://localhost:61616" />
        </bean>
        <bean id="messageDestination" class="org.apache.activemq.command.ActiveMQQueue">
                <constructor-arg value="messageQueue1" />
        </bean>
        <bean id="jmsTemplate" class="org.springframework.jms.core.JmsTemplate">
                <property name="connectionFactory" ref="connectionFactory" />
                <property name="receiveTimeout" value="10000" />
                <property name="defaultDestination" ref="messageDestination" />
        </bean>

        <bean id="springJmsProducer" class="com.javacodegeeks.spring.jms.SpringJmsProducer" ↩
            >
                <property name="jmsTemplate" ref="jmsTemplate" />
        </bean>

        <bean id="springJmsConsumer" class="com.javacodegeeks.spring.jms.SpringJmsConsumer" ↩
            >
                <property name="jmsTemplate" ref="jmsTemplate" />
        </bean>

</beans>
```

Since the `JmsTemplate` has reference to a default destination, we can simply call `jmsTemplate.send(messageCreator)` without passing in the destination. This form of the `send()` method only takes a `MessageCreator` object. With no destination specified, `JmsTemplate` will assume that you want the message sent to the default destination.

SpringJmsProducer:

```
package com.javacodegeeks.spring.jms;

import javax.jms.Destination;
import javax.jms.JMSException;
import javax.jms.Message;
import javax.jms.Session;

import org.springframework.jms.core.JmsTemplate;
import org.springframework.jms.core.MessageCreator;

public class SpringJmsProducer {
        private JmsTemplate jmsTemplate;
        private Destination destination;

        public JmsTemplate getJmsTemplate() {
                return jmsTemplate;
        }

        public void setJmsTemplate(JmsTemplate jmsTemplate) {
                this.jmsTemplate = jmsTemplate;
        }

        public Destination getDestination() {
                return destination;
        }

        public void setDestination(Destination destination) {
                this.destination = destination;
        }

        public void sendMessage(final String msg) {
                System.out.println("Producer sends " + msg);
                if (destination == null) {
                        jmsTemplate.send(new MessageCreator() {
                                public Message createMessage(Session session)
                                                throws JMSException {
                                        return session.createTextMessage(msg);
                                }
                        });
                } else {
                        jmsTemplate.send(destination, new MessageCreator() {
                                public Message createMessage(Session session)
                                                throws JMSException {
                                        return session.createTextMessage(msg);
                                }
                        });
                }
        }
}
```

Likewise, the consumer bean is modified to call `jmsTemplate.receive()` which doesn't take any destination value.

SpringJmsConsumer:

```
package com.javacodegeeks.spring.jms;

import javax.jms.Destination;
import javax.jms.JMSException;
import javax.jms.TextMessage;

import org.springframework.jms.core.JmsTemplate;
```

```
public class SpringJmsConsumer {
        private JmsTemplate jmsTemplate;
        private Destination destination;

        public JmsTemplate getJmsTemplate() {
                return jmsTemplate;
        }

        public void setJmsTemplate(JmsTemplate jmsTemplate) {
                this.jmsTemplate = jmsTemplate;
        }

        public Destination getDestination() {
                return destination;
        }

        public void setDestination(Destination destination) {
                this.destination = destination;
        }

        public String receiveMessage() throws JMSException {
                TextMessage textMessage;
                if (destination == null) {
                        textMessage = (TextMessage) jmsTemplate.receive();
                } else {
                        textMessage = (TextMessage) jmsTemplate.receive(destination);
                }
                return textMessage.getText();
        }
}
```

We will now modify our previous example of sending and receiving message through JmsTemplate so that it uses the default destination configuration.

SpringJmsTemplateDefaultDestinExample:

```
package com.javacodegeeks.spring.jms;

import java.net.URISyntaxException;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class SpringJmsTemplateDefaultDestinExample {
        public static void main(String[] args) throws URISyntaxException, Exception {
                ClassPathXmlApplicationContext context = new ClassPathXmlApplicationContext ↩
                    (
                                "appContextWithDefaultDestin.xml");

                try {
                        SpringJmsProducer springJmsProducer = (SpringJmsProducer) context
                                        .getBean("springJmsProducer");
                        springJmsProducer.sendMessage("SomeTask");

                        SpringJmsConsumer springJmsConsumer = (SpringJmsConsumer) context
                                        .getBean("springJmsConsumer");
                        System.out.println("Consumer receives " + springJmsConsumer. ↩
                            receiveMessage());
                } finally {
                        context.close();
                }
        }
}
```

Output:

```
Producer sends SomeTask
Consumer receives SomeTask
```

4.8 JmsTemplate with MessageConverter

Think of a scenario where we have to send and receive custom objects, in such cases, if you are actual payload object is different from the custom object then you will end up with some conversion code that will manage the conversion of custom object to JMS message Object and from JMS message object to custom object. If we have to do this at multiple points in your application, then there is a possibility that we will end up with duplication of code. Spring supports message conversion through its `MessageConverter` interface:

MessageConverter:

```
public interface MessageConverter {
    public Message toMessage(Object object, Session session);
    public Object fromMessage(Message message);
}
```

In our example, the custom object is a simple `Person` bean.

Person:

```
package com.javacodegeeks.spring.jms;

public class Person {
        private String name;
        private Integer age;
        public Person(String name, Integer age) {
                this.name = name;
                this.age = age;
        }
        public String getName() {
                return name;
        }
        public Integer getAge() {
                return age;
        }
        public String toString() {
                return "Person: name(" + name + "), age(" + age + ")";
        }
}
```

Here is our converter which converts `Person` to `MapMessage` and manufactures `Person` from `MapMessage`. `JmsTemplate` interacts with this message converter, for sending messages, `toMessage()` is called to convert an object to a Message. On the receiving the message, the `fromMessage()` method is called to convert an incoming Message into an Object.

PersonMessageConverter:

```
package com.javacodegeeks.spring.jms;

import javax.jms.JMSException;
import javax.jms.MapMessage;
import javax.jms.Message;
import javax.jms.Session;

import org.springframework.jms.support.converter.MessageConversionException;
import org.springframework.jms.support.converter.MessageConverter;
```

```
public class PersonMessageConverter implements MessageConverter{

        public Message toMessage(Object object, Session session)
                        throws JMSException, MessageConversionException {
                Person person = (Person) object;
                MapMessage message = session.createMapMessage();
                message.setString("name", person.getName());
                message.setInt("age", person.getAge());
                return message;
        }

        public Object fromMessage(Message message) throws JMSException,
                        MessageConversionException {
                MapMessage mapMessage = (MapMessage) message;
                Person person = new Person(mapMessage.getString("name"), mapMessage.getInt( ↩
                    "age"));
                return person;
        }

}
```

Instead of explicitly calling `JmsTemplate.send()`, we now call `JmsTemplate.convertAndSend()` method which takes in the `Person` object itself.

SpringJmsPersonProducer:

```
package com.javacodegeeks.spring.jms;

import org.springframework.jms.core.JmsTemplate;

public class SpringJmsPersonProducer {

        private JmsTemplate jmsTemplate;

        public JmsTemplate getJmsTemplate() {
                return jmsTemplate;
        }

        public void setJmsTemplate(JmsTemplate jmsTemplate) {
                this.jmsTemplate = jmsTemplate;
        }

        public void sendMessage(final Person person) {
                getJmsTemplate().convertAndSend(person);
        }
}
```

Likewise, on the receiving end, we won't need to call `fromMessage()` to convert the message returned from `JmsTemplate's receive()`. Instead, we'll now call `JmsTemplate.receiveAndConvert()`. which receives the message from default destination and converts the message to the custom object.

SpringJmsPersonConsumer:

```
package com.javacodegeeks.spring.jms;

import javax.jms.JMSException;

import org.springframework.jms.core.JmsTemplate;

public class SpringJmsPersonConsumer {

        private JmsTemplate jmsTemplate;
```

```
        public JmsTemplate getJmsTemplate() {
                return jmsTemplate;
        }

        public void setJmsTemplate(JmsTemplate jmsTemplate) {
                this.jmsTemplate = jmsTemplate;
        }

        public Person receiveMessage() throws JMSException {
                Person person = (Person) getJmsTemplate().receiveAndConvert();
                return person;
        }
}
```

4.9 Configuring MessageConverter

Finally we have to associate the message converter with the `JmsTemplate` bean. Let's configure it as a in Spring. The following XML will handle that:

```
<bean id="personMessageConverter" class="com.javacodegeeks.spring.jms. ↩
    PersonMessageConverter" />
```

Next, the JmsTemplate bean needs to be fixed, we'll wire the `personMessageConverter` bean into JmsTemplate's `messageConverter` property.

appContextWithMessageConverter.xml:

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

        <bean id="connectionFactory" class="org.apache.activemq.ActiveMQConnectionFactory">
                <property name="brokerURL" value="tcp://localhost:61616" />
        </bean>
        <bean id="messageDestination" class="org.apache.activemq.command.ActiveMQQueue">
                <constructor-arg value="messageQueue1" />
        </bean>

        <bean id="jmsTemplate" class="org.springframework.jms.core.JmsTemplate">
                <property name="connectionFactory" ref="connectionFactory" />
                <property name="receiveTimeout" value="10000" />
                <property name="defaultDestination" ref="messageDestination" />
                <property name="messageConverter" ref="personMessageConverter" />
        </bean>

        <bean id="personMessageConverter" class="com.javacodegeeks.spring.jms. ↩
            PersonMessageConverter" />

        <bean id="springJmsPersonProducer" class="com.javacodegeeks.spring.jms. ↩
            SpringJmsPersonProducer">
                <property name="jmsTemplate" ref="jmsTemplate" />
        </bean>

        <bean id="springJmsPersonConsumer" class="com.javacodegeeks.spring.jms. ↩
            SpringJmsPersonConsumer">
                <property name="jmsTemplate" ref="jmsTemplate" />
        </bean>
```

```
</beans>
```

Let's now test the producer/consumer example by sending a person object.

SpringJmsMessageConverterExample:

```
package com.javacodegeeks.spring.jms;

import java.net.URISyntaxException;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class SpringJmsMessageConverterExample {
        public static void main(String[] args) throws URISyntaxException, Exception {
                ClassPathXmlApplicationContext context = new ClassPathXmlApplicationContext ↩
                    (
                                "appContextWithMessageConverter.xml");

                try {
                        SpringJmsPersonProducer springJmsProducer = ( ↩
                            SpringJmsPersonProducer) context
                                    .getBean("springJmsPersonProducer");
                        Person joe = new Person("Joe", 32);
                        System.out.println("Sending person " + joe);
                        springJmsProducer.sendMessage(joe);

                        SpringJmsPersonConsumer springJmsConsumer = ( ↩
                            SpringJmsPersonConsumer) context
                                    .getBean("springJmsPersonConsumer");
                        System.out.println("Consumer receives " + springJmsConsumer. ↩
                            receiveMessage());
                } finally {
                        context.close();
                }
        }
}
```

Output:

```
Sending person Person: name(Joe), age(32)
Consumer receives Person: name(Joe), age(32)
```

4.10 Download the Eclipse Project

This was an example about spring JMSTemplate.

Download

You can download the full source code of this example here: **springJmsTemplateExample.zip**

Chapter 5

How to Start Developing Layered Web Applications with Spring

Spring is a great framework to develop enterprise Java web applications. It really eases life of Java developers by providing tons of features. In this example, we will show you how to start developing layered web applications with Spring.

Our preferred development environment is Spring Tool Suite 3.8.2 based on Eclipse 4.6.1 version. However, as we are going to create the example as maven project, you can easily work within your own IDE as well. We are also using Spring Application Framework 4.3.1.RELEASE along with JDK 1.8_u112, Apache Tomcat 8.5.8, JSTL 1.2, and H2 database version 1.4.192.

Let's begin.

5.1 Create a new Maven WebApp project

Write click on Package Explorer and select New>Maven Project to create an new maven project.

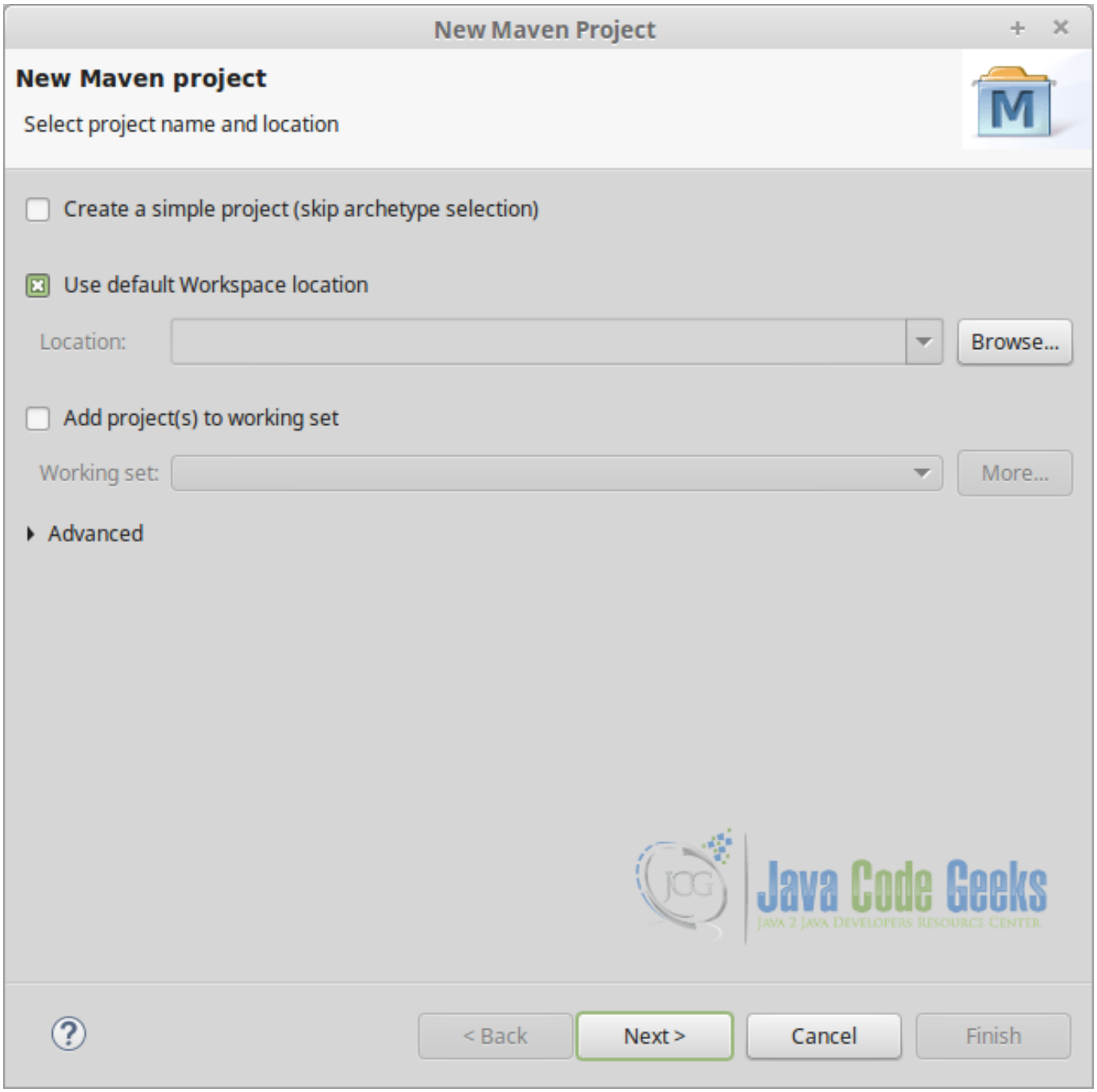

Figure 5.1: create new maven project

Click Next button, and select maven-archetype-webapp from among available archetypes.

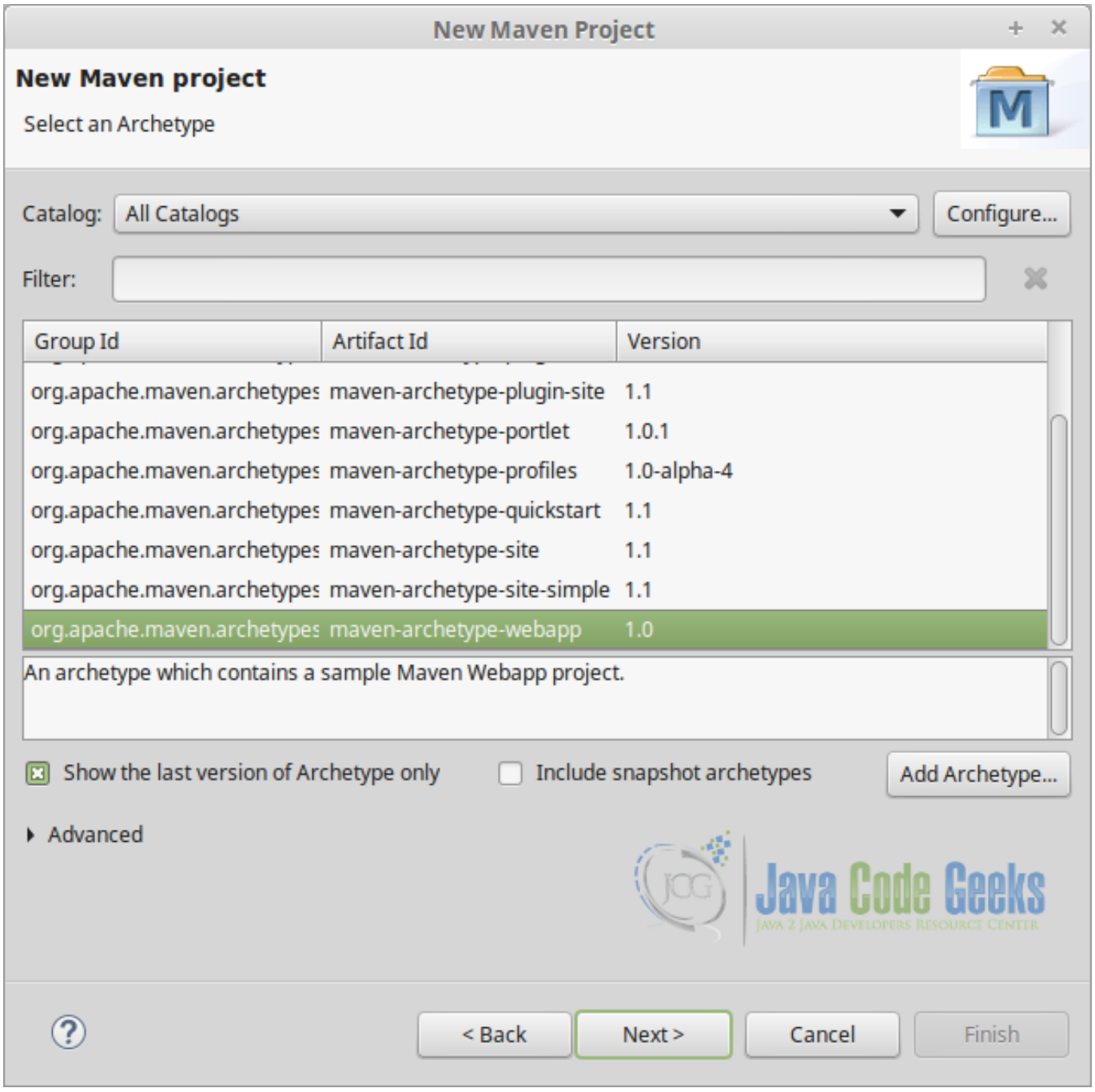

Figure 5.2: select maven webapp archetype

Click next button again, and provide group id and artifact id values as seen in the following screenshot.

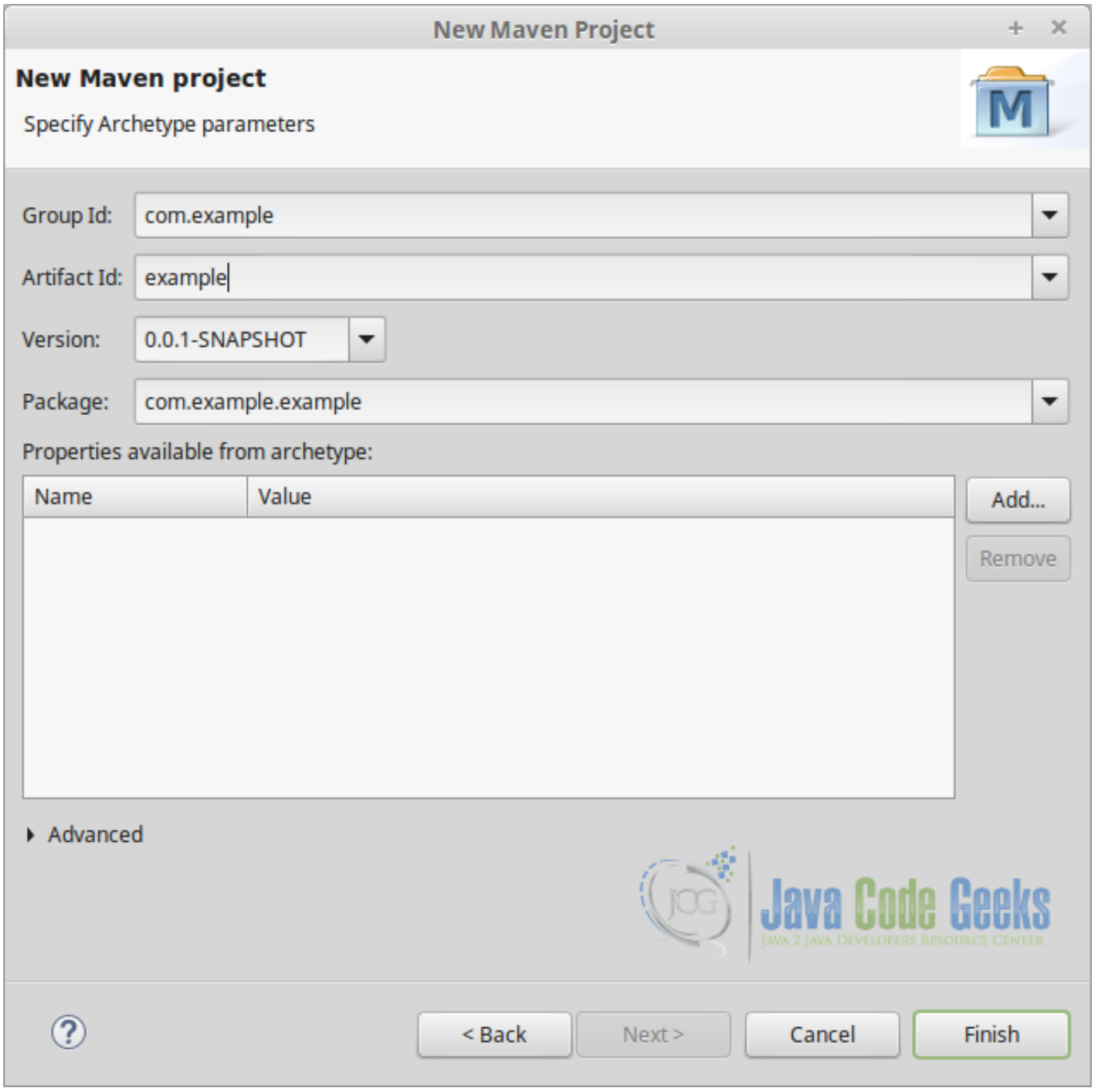

Figure 5.3: configure maven webapp project

Finally, click Finish button to finish creating your web application. Maven-archetype-webapp only create minimum number of files and directories required to run the web application in a Servlet Container. You have to manually create src/main/java, src/test/java and src/test/resources standard maven source folders in your project.

Write click on your project example and select New>Folder to create src/main/java, src/test/java and src/test/resources source folders consecutively.

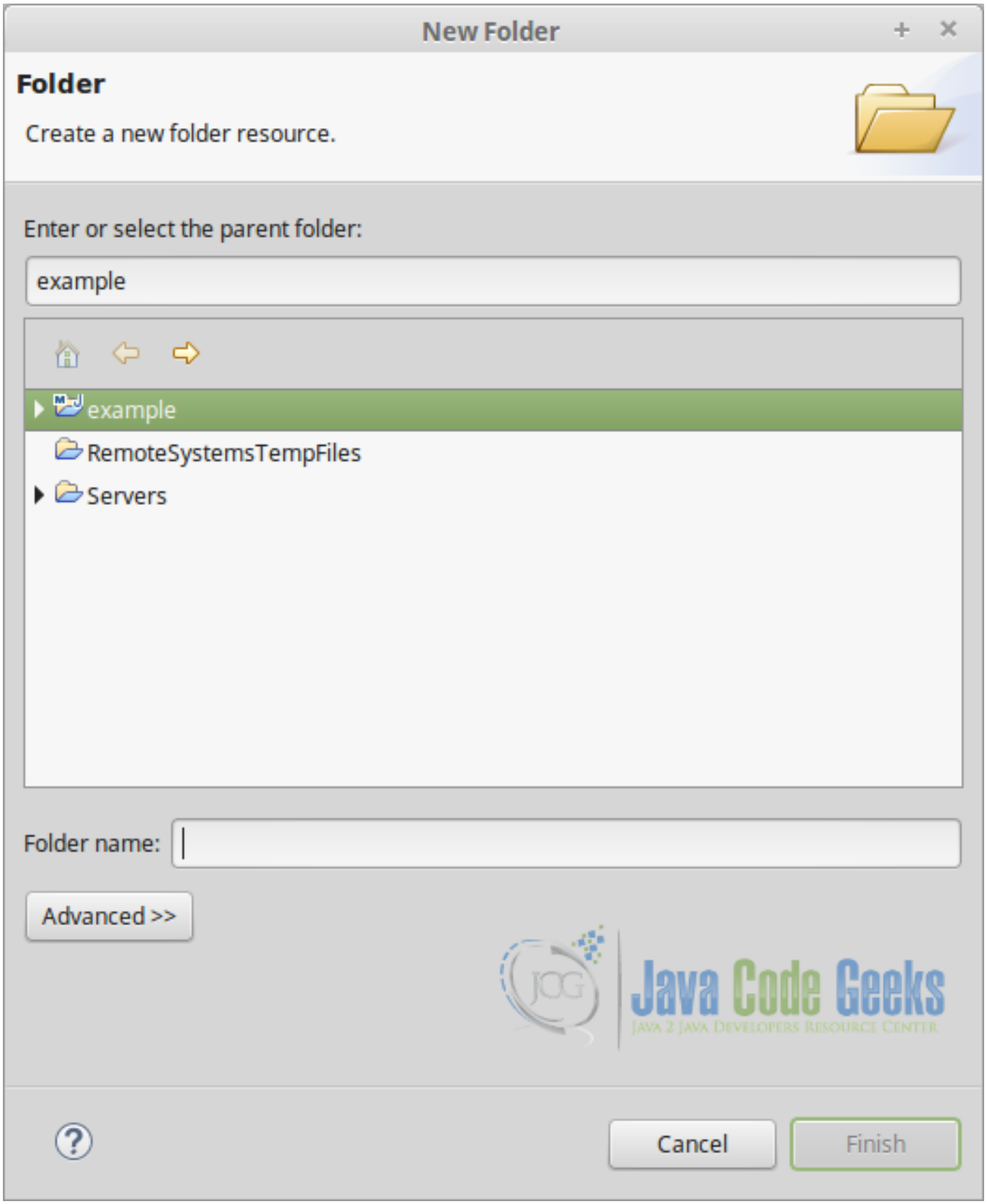

Figure 5.4: create source folders

After creating those source folders, click pom.xml in the project root folder in order to open up pom.xml editor, and add maven.compiler.source and maven.compiler.target properties with value 1.8 into it.

5.2 Add necessary dependencies in your project

Add following dependencies into your pom.xml. You can make use of pom.xml editor you opened up in the previous step.

```
<dependency>
                        <groupId>javax.servlet</groupId>
                        <artifactId>javax.servlet-api</artifactId>
                        <version>3.1.0</version>
                        <scope>provided</scope>
                </dependency>

                <dependency>
                        <groupId>javax.servlet.jsp</groupId>
                        <artifactId>javax.servlet.jsp-api</artifactId>
                        <version>2.3.1</version>
                        <scope>provided</scope>
                </dependency>

                <dependency>
                        <groupId>javax.servlet</groupId>
                        <artifactId>jstl</artifactId>
                        <version>1.2</version>
                </dependency>

                <dependency>
                        <groupId>org.springframework</groupId>
                        <artifactId>spring-webmvc</artifactId>
                        <version>4.3.1.RELEASE</version>
                </dependency>
                <dependency>
                        <groupId>org.springframework</groupId>
                        <artifactId>spring-jdbc</artifactId>
                        <version>4.3.1.RELEASE</version>
                </dependency>
                <dependency>
                        <groupId>log4j</groupId>
                        <artifactId>log4j</artifactId>
                        <version>1.2.17</version>
                </dependency>
                <dependency>
                        <groupId>com.h2database</groupId>
                        <artifactId>h2</artifactId>
                        <version>1.4.192</version>
                </dependency>
                <dependency>
                        <groupId>junit</groupId>
                        <artifactId>junit</artifactId>
                        <version>4.12</version>
                        <scope>test</scope>
                </dependency>
```

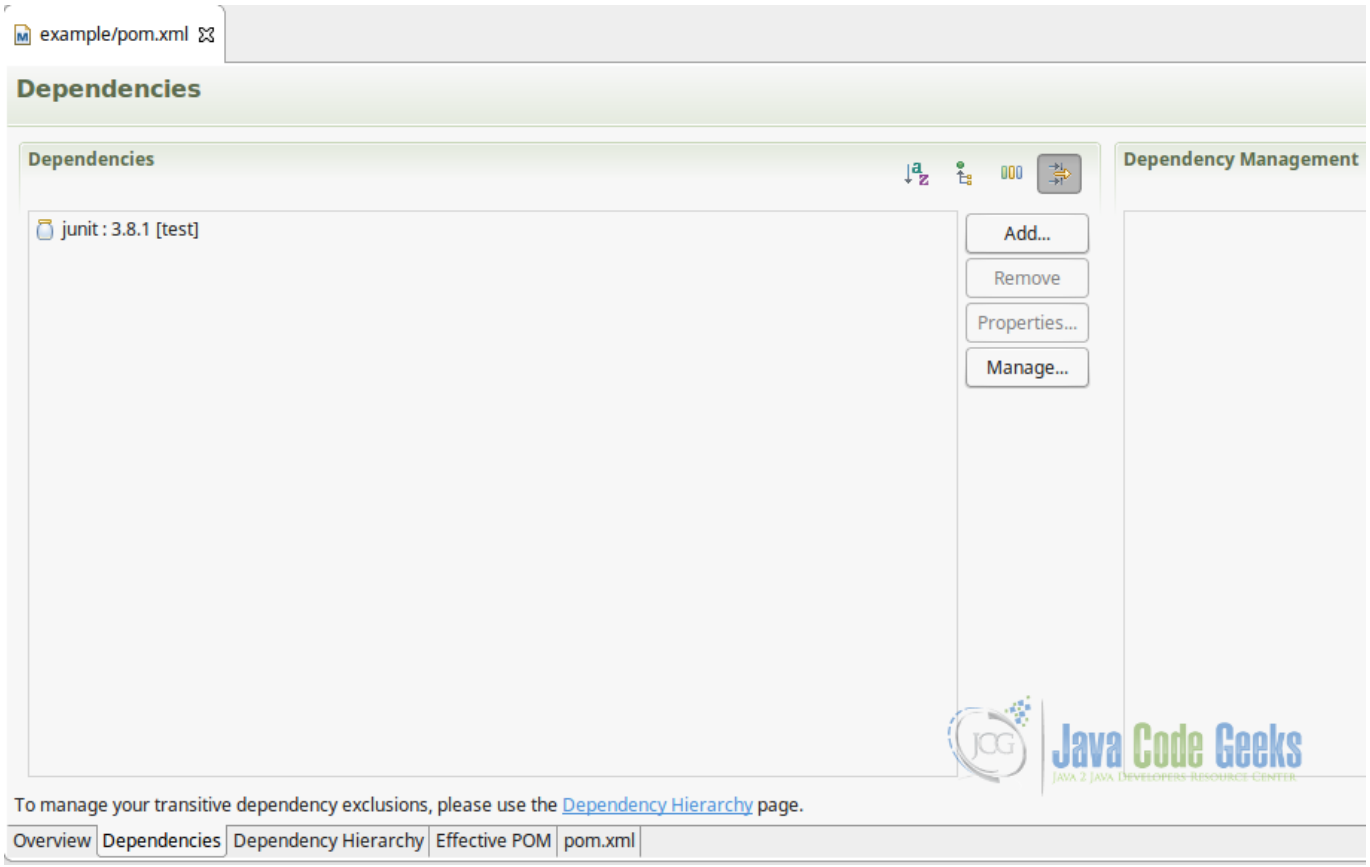

Figure 5.5: add necessary dependencies

Note that junit dependency already exists in your pom.xml when you first create your webapp project. It is added by webapp archetype by default. We only change its version to a newer value.

Figure 5.6: add dependency

You can either add those dependencies via add Dependency dialog, or switch into source view of pom.xml and copy all of them into <dependencies></dependencies> section. After this step, added dependencies should have been listed as follows.

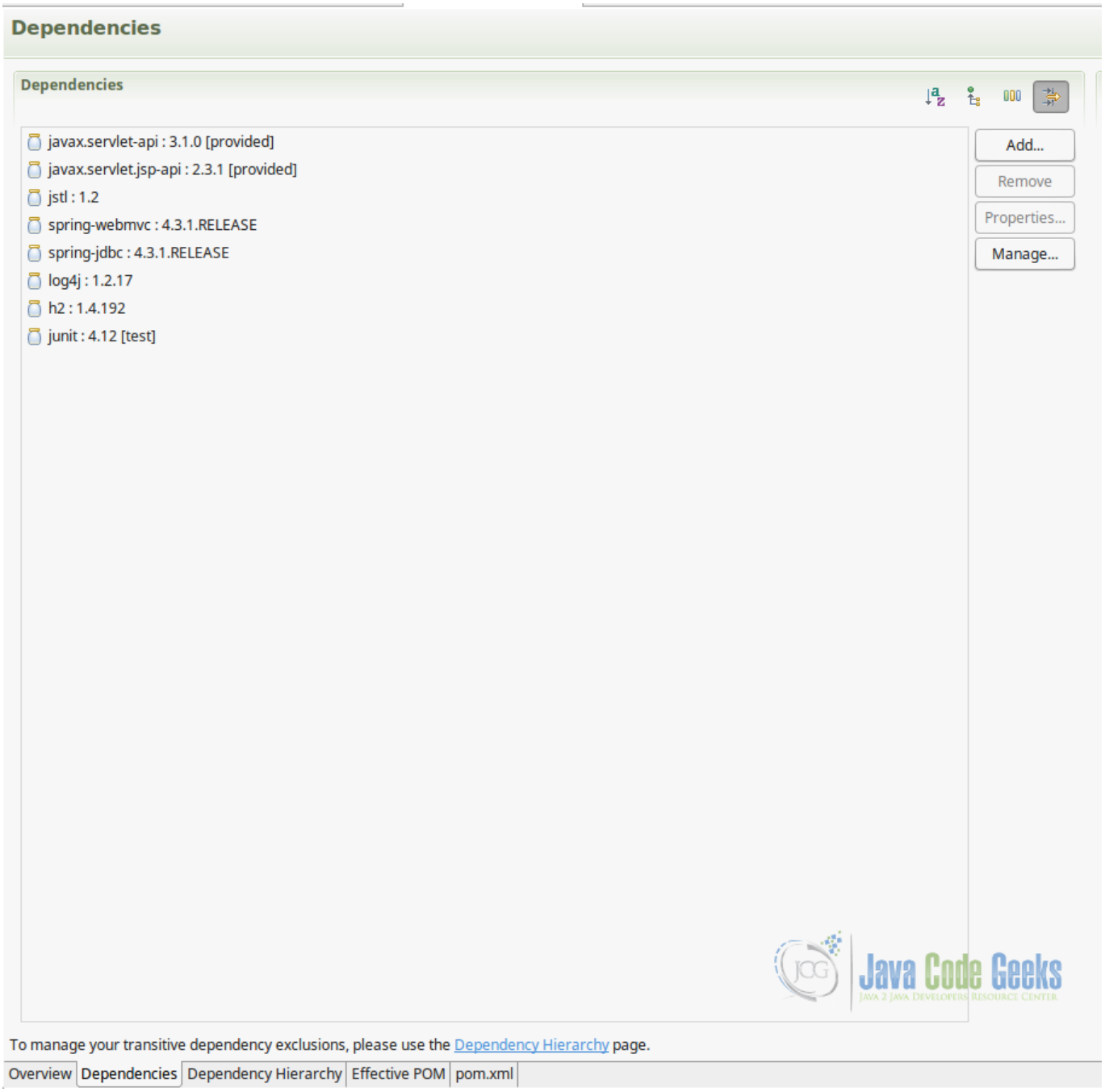

Figure 5.7: list of added dependencies

Finally perform a project update by right clicking the project and then clicking “Update Project” through Maven>Update Project...

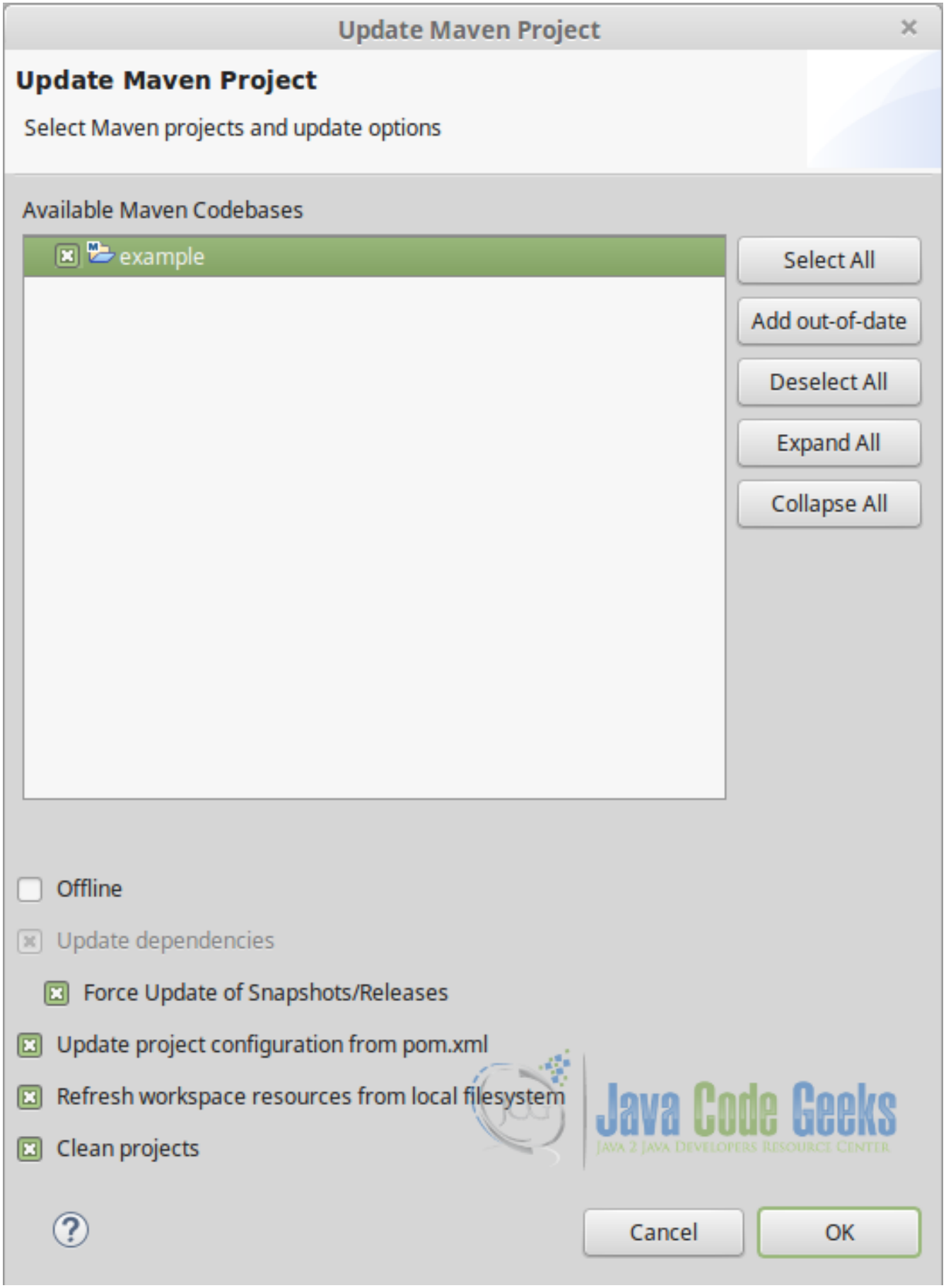

Figure 5.8: update maven project

You should have seen something similar in your Package Explorer as below. JRE System Library should have been changed into JavaSE-1.8 and so on.

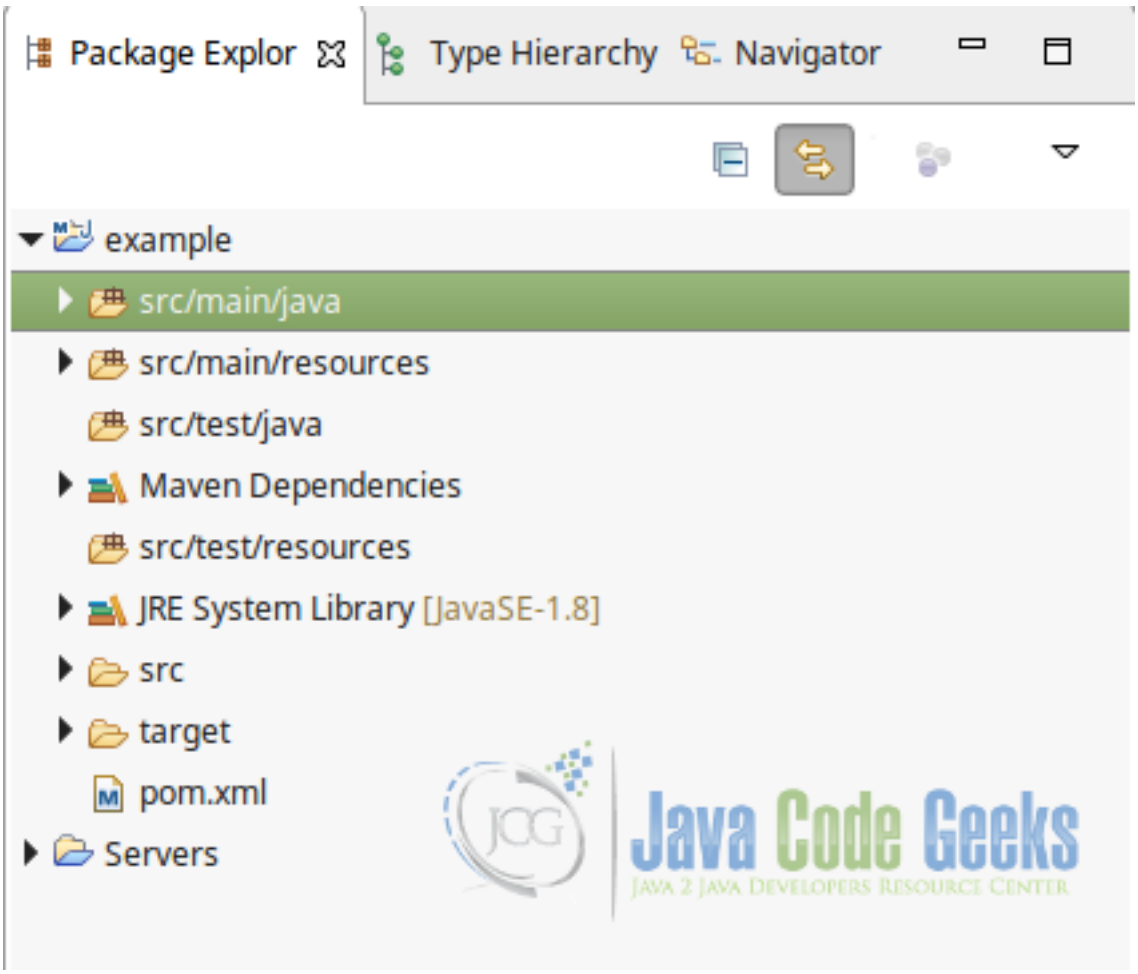

Figure 5.9: maven project update finished

5.3 Create log4j.xml

Create log4j.xml file under src/main/resources folder with the following content. It will help us to see log messages produced by Spring during execution of test methods and trace what is going on during those executions.

log4j.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE log4j:configuration PUBLIC "-//LOG4J" "log4j.dtd">
<log4j:configuration xmlns:log4j="https://jakarta.apache.org/log4j/">

        <appender name="CONSOLE" class="org.apache.log4j.ConsoleAppender">
                <layout class="org.apache.log4j.EnhancedPatternLayout">
                        <param name="ConversionPattern"
                                value="%d{HH:mm:ss,SSS} - %p - %C{1.}.%M(%L): %m%n" />
                </layout>
        </appender>

        <logger name="org.springframework">
                <level value="DEBUG" />
```

```
        </logger>

        <root>
                <level value="INFO" />

                <appender-ref ref="CONSOLE" />

        </root>

</log4j:configuration>
```

5.4 Prepare DDL and DML scripts to initialize database

Create schema.sql and data.sql files within src/main/resources with the following contents.

5.4.1 schema.sql

schema.sql

```
CREATE SEQUENCE PUBLIC.T_PERSON_SEQUENCE START WITH 1;

CREATE CACHED TABLE PUBLIC.T_PERSON(
    ID BIGINT NOT NULL,
    FIRST_NAME VARCHAR(255),
    LAST_NAME VARCHAR(255)
);

ALTER TABLE PUBLIC.T_PERSON ADD CONSTRAINT PUBLIC.CONSTRAINT_PERSON_PK PRIMARY KEY(ID);
```

5.4.2 data.sql

data.sql

```
INSERT INTO T_PERSON (ID,FIRST_NAME,LAST_NAME) VALUES (T_PERSON_SEQUENCE.NEXTVAL, 'John',' ↩
    Doe');
INSERT INTO T_PERSON (ID,FIRST_NAME,LAST_NAME) VALUES (T_PERSON_SEQUENCE.NEXTVAL, 'Joe',' ↩
    Doe');
```

5.5 Write Domain Class, Service and DAO Classes

5.5.1 Person.java

We are going to create a simple domain class with name Person as follows. It has only three attributes, id, firstName and lastName, and accessor methods for them.

Person.java

```
package com.example.model;

public class Person {
        private Long id;
        private String firstName;
        private String lastName;
        public Long getId() {
```

```
                return id;
        }
        public void setId(Long id) {
                this.id = id;
        }
        public String getFirstName() {
                return firstName;
        }
        public void setFirstName(String firstName) {
                this.firstName = firstName;
        }
        public String getLastName() {
                return lastName;
        }
        public void setLastName(String lastName) {
                this.lastName = lastName;
        }
}
```

We also create Service and DAO classes as follows, in order to perform simple persistence operations with our domain model.

5.5.2 PersonDao.java

PersonDao is a simple interface which defines basic persistence operations over Person instances like findById, create a new Person, update or delete an existing one.

PersonDao.java

```
package com.example.dao;

import java.util.List;

import com.example.model.Person;

public interface PersonDao {
        List<Person> findAll();
        Person findById(Long id);
        void create(Person person);
        void update(Person person);
        void delete(Long id);
}
```

5.5.3 JdbcPersonDao.java

JdbcPersonDao is an implementation of PersonDao interface which employs JdbcTemplate bean of Spring in order to implement persistence operations via JDBC API. @Repository annotation causes a singleton scope bean to be created in Spring Container.

JdbcPersonDao.java

```
package com.example.dao;

import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
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.core.RowMapper;
import org.springframework.jdbc.support.GeneratedKeyHolder;
import org.springframework.jdbc.support.KeyHolder;
import org.springframework.stereotype.Repository;

import com.example.model.Person;

@Repository
public class JdbcPersonDao implements PersonDao {
        private JdbcTemplate jdbcTemplate;

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

        @Override
        public List<Person> findAll() {
                return jdbcTemplate.query("select id, first_name, last_name from t_person",
                                new RowMapper<Person>() {

                                        @Override
                                        public Person mapRow(ResultSet rs, int rowNum)  ↩
                                            throws SQLException {
                                                Person person = new Person();
                                                person.setId(rs.getLong("id"));
                                                person.setFirstName(rs.getString(" ↩
                                                    first_name"));
                                                person.setLastName(rs.getString("last_name" ↩
                                                    ));
                                                return person;
                                        }
                                });
        }

        @Override
        public Person findById(Long id) {
                return jdbcTemplate.queryForObject("select first_name, last_name from  ↩
                    t_person where id = ?",
                                new RowMapper<Person>() {

                                        @Override
                                        public Person mapRow(ResultSet rs, int rowNum)  ↩
                                            throws SQLException {
                                                Person person = new Person();
                                                person.setId(id);
                                                person.setFirstName(rs.getString(" ↩
                                                    first_name"));
                                                person.setLastName(rs.getString("last_name" ↩
                                                    ));
                                                return person;
                                        }
                                }, id);
        }

        @Override
        public void create(Person person) {
                KeyHolder keyHolder = new GeneratedKeyHolder();
                jdbcTemplate.update(new PreparedStatementCreator() {

                        @Override
                        public PreparedStatement createPreparedStatement(Connection con)  ↩
```

```
                        throws SQLException {
                            PreparedStatement stmt = con.prepareStatement("insert into  ↩
                                t_person(id,first_name,last_name) values( ↩
                                t_person_sequence.nextval,?,?)");
                            stmt.setString(1, person.getFirstName());
                            stmt.setString(2, person.getLastName());
                            return stmt;
                        }
                }, keyHolder);
                person.setId(keyHolder.getKey().longValue());
        }

        @Override
        public void update(Person person) {
                jdbcTemplate.update("update t_person set first_name = ?, last_name = ?  ↩
                    where id = ?", person.getFirstName(),
                            person.getLastName(), person.getId());
        }

        @Override
        public void delete(Long id) {
                jdbcTemplate.update("delete from t_person where id = ?", id);
        }
}
```

5.5.4 PersonService.java

PersonService interface defines basic service methods to be consumed by the cotroller layer.

PersonService.java

```
package com.example.service;

import java.util.List;

import com.example.model.Person;

public interface PersonService {
        List<Person> findAll();
        Person findById(Long id);
        void create(Person person);
        void update(Person person);
        void delete(Long id);
}
```

5.5.5 PersonServiceImpl.java

PersonServiceImpl is a transactional service implementation of PersonService interface which uses PersonDao bean in order to perform persistence operations. Its role is simply delegating to its DAO bean apart from being transactional in this context.

@Service annotation causes a singleton scope bean to be created in Spring Container, and @Transactional annotation makes all of its public method transactional by default.

PersonServiceImpl.java

```
package com.example.service;

import java.util.List;

import org.springframework.beans.factory.annotation.Autowired;
```

```
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

import com.example.dao.PersonDao;
import com.example.model.Person;

@Service
@Transactional
public class PersonServiceImpl implements PersonService {
        private PersonDao personDao;

        @Autowired
        public void setPersonDao(PersonDao personDao) {
                this.personDao = personDao;
        }


        @Override
        public List<Person> findAll() {
                return personDao.findAll();
        }


        @Override
        public Person findById(Long id) {
                return personDao.findById(id);
        }

        @Override
        public void create(Person person) {
                personDao.create(person);
        }

        @Override
        public void update(Person person) {
                personDao.update(person);
        }

        @Override
        public void delete(Long id) {
                personDao.delete(id);
        }
}
```

5.6 Write Controller Classes and JSPs to handle UI logic

We will make use of Spring MVC to handle web requests in order to perform CRUD operations related with person records. We create a separate Controller class and a corresponding JSP file for each persistence operation that will be available to our users.

5.6.1 PersonListController and personList.jsp

PersonListController class handles web request to display returned Persons from PersonService.findAll() method.

PersonListController.java

```
package com.example.controller;

import org.springframework.beans.factory.annotation.Autowired;
```

```
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;

import com.example.service.PersonService;

@Controller
public class PersonListController {
        @Autowired
        private PersonService personService;

        @RequestMapping(value = "/listPersons", method = RequestMethod.GET)
        public String findAllPersons(Model model) {
                model.addAttribute("persons", personService.findAll());
                return "personList";
        }
}
```

@Controller annotation causes a singleton bean to be created in Spring Container. @RequestMapping annotation over the methods maps methods with request URIs to be handled by those controller beans. For example, PersonListController.findAllPersons method is mapped with /listPersons request URI accessed with an HTTP GET via the corresponding @RequestMapping annotation. @Autowire annotation injects a service bean of type PersonService available in the container.

Before creating following JSP file, create first a folder named as jsp within src/main/webapp/WEB-INF folder in your project, and then place all those JSP files under that directory. Alhough src/main/webapp folder is accessible by users at runtime, any file or directoy within WEB-INF folder, on the other hand, is not. Placing JSP files under a directoy within WEB-INF folder limits their accessibility only through those Controller beans. Hence, users won't be able to type names of those JSP over the browser's URL address bar in order to access them independently from their related Controllers.

personList.jsp

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
        pageEncoding="UTF-8"%>
<%@ taglib prefix="c" uri="https://java.sun.com/jsp/jstl/core"%>
<%@ page isELIgnored="false"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "https://www.w3.org/TR/html4 ↩
    /loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>Person List View</title>
</head>
<body>
        <h1>Person List View</h1>
        <a href = "<%=request.getContextPath()%>/mvc/createPerson">Create</a>
        <br/>
        <br/>

                <thead>
                        |ID|First Name|Last Name|Action
                </thead>
                <c:forEach items="${persons}" var="person">
                        |${person.id}|${person.firstName}|${person.lastName}|
                                <form action="<%=request.getContextPath()%>/mvc/ ↩
                                    updatePerson/${person.id}" method="get">
                                        <input type="submit" value="Update">
                                </form>
                                |
                                <form action="<%=request.getContextPath()%>/mvc/ ↩
                                    deletePerson/${person.id}" method="get">
                                        <input type="submit" value="Delete">
```

```
                                        </form>

                </c:forEach>

        <br />
        <font color="blue"> ${message} </font>
</body>
</html>
```

5.6.2 PersonCreateController and personCreate.jsp

PersonCreateController.java

```
package com.example.controller;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
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.bind.annotation.SessionAttributes;
import org.springframework.web.bind.support.SessionStatus;
import org.springframework.web.servlet.mvc.support.RedirectAttributes;

import com.example.model.Person;
import com.example.service.PersonService;

@Controller
@SessionAttributes("person")
public class PersonCreateController {
        @Autowired
        private PersonService personService;

        @RequestMapping(value = "/createPerson", method = RequestMethod.GET)
        public String startCreatingNewPerson(Model model) {
                model.addAttribute("person", new Person());
                return "personCreate";
        }

        @RequestMapping(value = "/createPersonFailed", method = RequestMethod.GET)
        public String createPersonFailed() {
                return "personCreate";
        }

        @RequestMapping(value = "/createPerson", method = RequestMethod.POST)
        public String performCreate(@ModelAttribute Person person, RedirectAttributes ↩
            redirectAttributes,
                        SessionStatus sessionStatus) {
                String message = null;
                String viewName = null;
                try {
                        personService.create(person);
                        message = "Person created. Person id :" + person.getId();
                        viewName = "redirect:/mvc/listPersons";
                        sessionStatus.setComplete();
                } catch (Exception ex) {
                        message = "Person create failed";
                        viewName = "redirect:/mvc/createPersonFailed";
                }
```

```
                redirectAttributes.addFlashAttribute("message", message);
                return viewName;
        }
}
```

personCreate.jsp

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
        pageEncoding="UTF-8"%>
<%@ taglib prefix="form" uri="https://www.springframework.org/tags/form"%>
<%@ page isELIgnored="false"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "https://www.w3.org/TR/html4 ↩
    /loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>Person Create View</title>
</head>
<body>
        <h1>Person Create View</h1>
        <form:form modelAttribute="person" method="post"
                servletRelativeAction="/mvc/createPerson">

                        |First Name|<form:input path="firstName" />

                        |Last Name|<form:input path="lastName" />


                <form:button name="Create">Create</form:button>
        </form:form>
        <br />
        <font color="red"> ${message} </font>
</body>
</html>
```

5.6.3 PersonUpdateController and personUpdate.jsp

PersonUpdateController.java

```
package com.example.controller;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.SessionAttributes;
import org.springframework.web.bind.support.SessionStatus;
import org.springframework.web.servlet.mvc.support.RedirectAttributes;

import com.example.model.Person;
import com.example.service.PersonService;

@Controller
@SessionAttributes("person")
public class PersonUpdateController {
        @Autowired
        private PersonService personService;
```

```
        @RequestMapping(value = "/updatePerson/{id}", method = RequestMethod.GET)
        public String selectForUpdate(@PathVariable Long id, Model model) {
                model.addAttribute("person", personService.findById(id));
                return "personUpdate";
        }

        @RequestMapping(value="/updatePersonFailed", method=RequestMethod.GET)
        public String updatePersonFailed() {
                return "personUpdate";
        }

        @RequestMapping(value = "/updatePerson", method = RequestMethod.POST)
        public String performUpdate(@ModelAttribute Person person, RedirectAttributes  ↩
            redirectAttributes,
                        SessionStatus sessionStatus) {
                String message = null;
                String viewName = null;
                try {
                        personService.update(person);
                        message = "Person updated. Person id :" + person.getId();
                        viewName = "redirect:/mvc/listPersons";
                        sessionStatus.setComplete();
                } catch (Exception ex) {
                        message = "Person update failed. ";
                        viewName = "redirect:/mvc/updatePersonFailed";
                }
                redirectAttributes.addFlashAttribute("message", message);
                return viewName;
        }
}
```

personUpdate.jsp

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
        pageEncoding="UTF-8"%>
<%@ taglib prefix="form" uri="https://www.springframework.org/tags/form"%>
<%@ page isELIgnored="false"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "https://www.w3.org/TR/html4 ↩
    /loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>Person Update View</title>
</head>
<body>
        <h1>Person Update View</h1>
        <form:form  modelAttribute="person" method="post"
                servletRelativeAction="/mvc/updatePerson">
                |=======================
                        |ID|<form:input path="id" readonly="true" />
                        |First Name|<form:input path="firstName" /> <form:errors
                                                path="firstName" />
                        |Last Name|<form:input path="lastName" /> <form:errors path=" ↩
                            lastName" />

                |=======================
                <form:errors>
                </form:errors>
                <form:button name="Update">Update</form:button>
        </form:form>
        <font color="red"> ${message} </font>
```

```
</body>
</html>
```

5.6.4 PersonDeleteController and personDelete.jsp

PersonDeleteController.java

```
package com.example.controller;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.SessionAttributes;
import org.springframework.web.bind.support.SessionStatus;
import org.springframework.web.servlet.mvc.support.RedirectAttributes;

import com.example.model.Person;
import com.example.service.PersonService;

@Controller
@SessionAttributes("person")
public class PersonDeleteController {
        @Autowired
        private PersonService personService;

        @RequestMapping(value = "/deletePerson/{id}", method = RequestMethod.GET)
        public String selectForDelete(@PathVariable Long id, Model model) {
                model.addAttribute("person", personService.findById(id));
                return "personDelete";
        }

        @RequestMapping(value = "/deletePersonFailed", method = RequestMethod.GET)
        public String deletePersonFailed() {
                return "personDelete";
        }

        @RequestMapping(value = "/deletePerson", method = RequestMethod.POST)
        public String delete(@ModelAttribute Person person, RedirectAttributes  ↩
            redirectAttributes, SessionStatus sessionStatus) {
                String message = null;
                String viewName = null;
                try {
                        personService.delete(person.getId());
                        message = "Person deleted. Person id :" + person.getId();
                        viewName = "redirect:/mvc/listPersons";
                        sessionStatus.setComplete();
                } catch (Exception ex) {
                        message = "Person delete failed.";
                        viewName = "redirect:/mvc/deletePersonFailed";
                }
                redirectAttributes.addFlashAttribute("message", message);
                return viewName;
        }
}
```

personDelete.jsp

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
        pageEncoding="UTF-8"%>
<%@ taglib prefix="form" uri="https://www.springframework.org/tags/form"%>
<%@ page isELIgnored="false"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "https://www.w3.org/TR/html4 ↩
    /loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>Person Delete View</title>
</head>
<body>
        <h1>Person Delete View</h1>
        <form:form modelAttribute="person" method="post"
                servletRelativeAction="/mvc/deletePerson">


                                |ID|<form:input path="id" readonly="true" />|First Name|< ↩
                                    form:input path="firstName" readonly="true" />|Last Name ↩
                                    |<form:input path="lastName" readonly="true" />< ↩
                                    form:button name="Delete">Delete</form:button>
        </form:form>
        <font color="red"> ${message} </font>
</body>
</html>
```

5.7 Configure your web application to bootstrap with Spring

We will configure Spring Container with Java based configuration approach as follows.

5.7.1 WebAppConfig.java

WebAppConfig class contains necessary directives and bean definitions for Spring Container to provide required functionalities.

WebAppConfig.java

```
package com.example.config;

import javax.sql.DataSource;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.datasource.DataSourceTransactionManager;
import org.springframework.jdbc.datasource.embedded.EmbeddedDatabaseBuilder;
import org.springframework.jdbc.datasource.embedded.EmbeddedDatabaseType;
import org.springframework.transaction.PlatformTransactionManager;
import org.springframework.transaction.annotation.EnableTransactionManagement;
import org.springframework.web.servlet.ViewResolver;
import org.springframework.web.servlet.config.annotation.EnableWebMvc;
import org.springframework.web.servlet.view.InternalResourceViewResolver;

@Configuration
@EnableTransactionManagement
@ComponentScan(basePackages = "com.example")
@EnableWebMvc
public class WebAppConfig {
```

```
        @Bean
        public DataSource dataSource() {
                return new EmbeddedDatabaseBuilder().setType(EmbeddedDatabaseType.H2)
                                .addScripts("classpath:/schema.sql", "classpath:/data.sql") ↩
                                    .build();
        }

        @Bean
        public PlatformTransactionManager transactionManager(DataSource dataSource) {
                return new DataSourceTransactionManager(dataSource);
        }

        @Bean
        public JdbcTemplate jdbcTemplate(DataSource dataSource) {
                return new JdbcTemplate(dataSource);
        }

        @Bean
        public ViewResolver viewResolver() {
                InternalResourceViewResolver viewResolver = new  ↩
                    InternalResourceViewResolver();
                viewResolver.setPrefix("/WEB-INF/jsp/");
                viewResolver.setSuffix(".jsp");
                return viewResolver;
        }
}
```

@Configuration annotation marks it as a Spring Configuration class so that Spring will process it as ApplicationContext metadata source. @EnableTransactionManagement annotation enables annotation based declarative transaction support in the container.

@ComponentScan annotation causes Spring to scan base packages given as attribute value, in order to create beans out of classes under those packages which have @Controller, @Service, @Repository and @Component on top of them.

@EnableWebMvc annotation activates annotation based MVC capabilities of the container, like handling requests mapped via @RequestMapping etc.

5.7.2 WebAppInitializer.java

Spring provides a mechanism in order to create ApplicationContext without touching web.xml at all, purely in Java way in other words. Following WebAppInitializer class extends from AbstractDispatcherServletInitializer, executed by a special ServletContainerInitializer available in the Spring distribution, configures DispatcherServlet and its WebApplicationContext using given metadata sources.

In our configuration requests coming to our web application will need to have /mvc prefix so that they will be intercepted by Spring's DispatcherServlet which dispatches web requests to corresponding handler methods at runtime.

WebAppInitializer.java

```
package com.example.config;

import org.springframework.web.context.WebApplicationContext;
import org.springframework.web.context.support.AnnotationConfigWebApplicationContext;
import org.springframework.web.servlet.support.AbstractDispatcherServletInitializer;

public class WebAppInitializer extends AbstractDispatcherServletInitializer {

        @Override
        protected WebApplicationContext createServletApplicationContext() {
                AnnotationConfigWebApplicationContext wac = new  ↩
                    AnnotationConfigWebApplicationContext();
                wac.register(WebAppConfig.class);
```

```
                return wac;
        }

        @Override
        protected String[] getServletMappings() {
                return new String[] { "/mvc/*" };
        }

        @Override
        protected WebApplicationContext createRootApplicationContext() {
                return null;
        }
}
```

5.8 Configure your IDE to run Tomcat instance

Right click on the Server tab view, and select New>Server in order to make a new server configuration within your IDE as follows.

Figure 5.10: new server

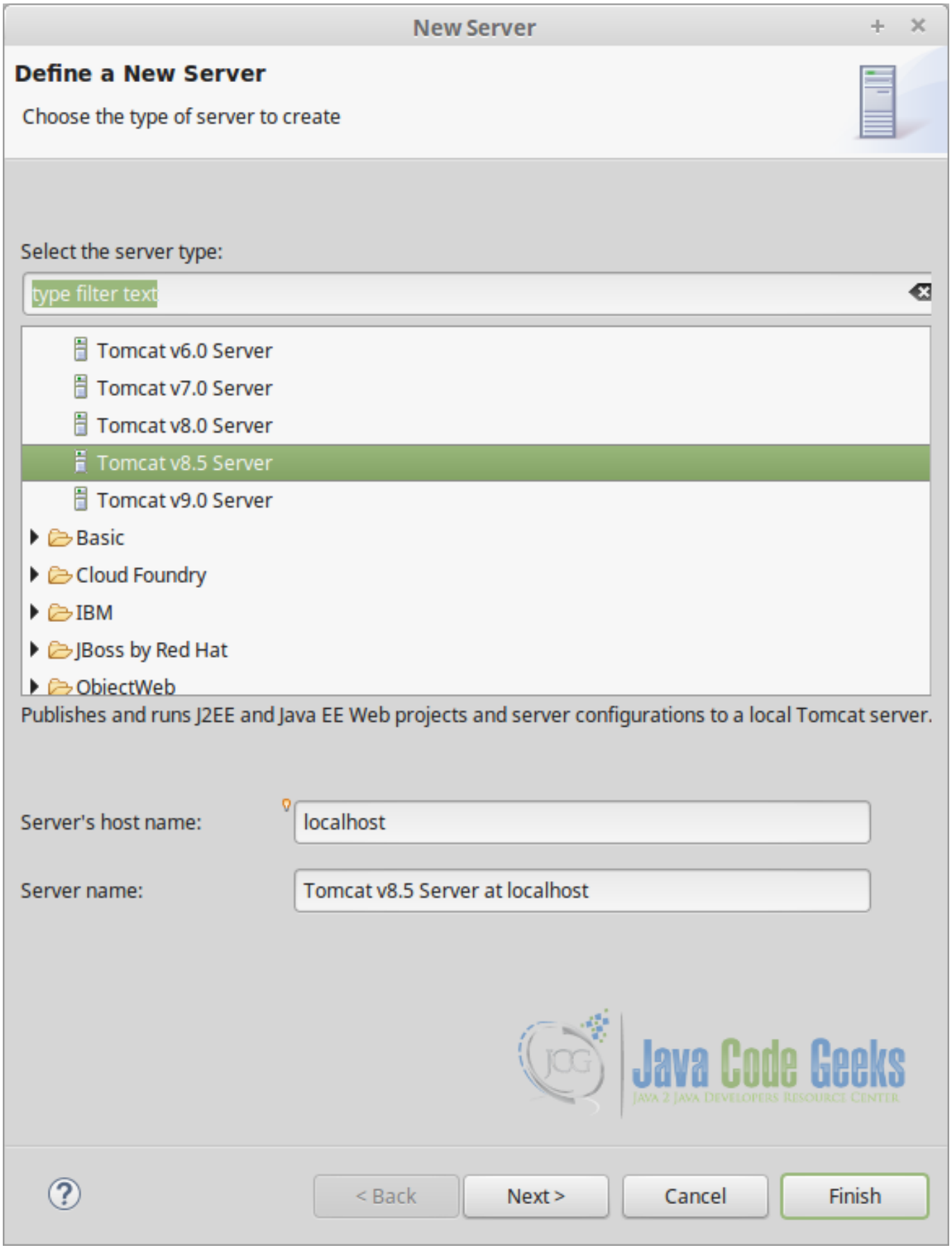

Figure 5.11: select apache tomcat 8.5

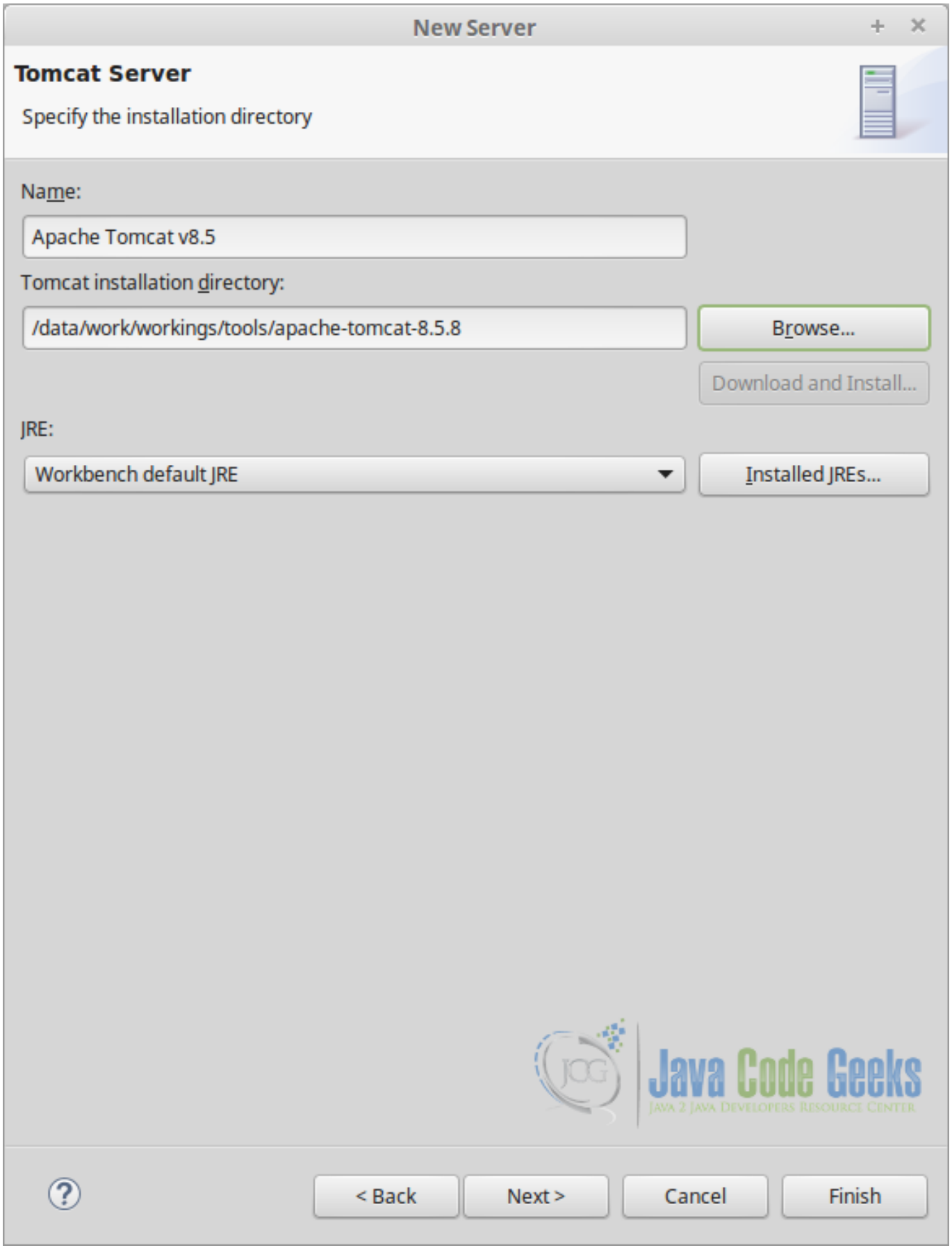

Figure 5.12: select tomcat location

Figure 5.13: list of available projects for deployment

Figure 5.14: project added into configured projects

At the end of those steps, you should see something similar below in your Servers view.

Figure 5.15: server configuration finished

5.9 Run Tomcat instance and access your webapp through your browser

After configuring your server instance and adding your webapp as configured project into the server instance, click start icon in the Servers view in order to bootstrap your webapp. After several hunderd lines of log output, you should see something similar to the following output in your console.

```
17:08:41,214 - DEBUG - o.s.w.s.FrameworkServlet.initWebApplicationContext(568): Published ↩
    WebApplicationContext of servlet 'dispatcher' as ServletContext attribute with name [org ↩
    .springframework.web.servlet.FrameworkServlet.CONTEXT.dispatcher]
17:08:41,214 - INFO - o.s.w.s.FrameworkServlet.initServletBean(508): FrameworkServlet ' ↩
    dispatcher': initialization completed in 1055 ms
17:08:41,214 - DEBUG - o.s.w.s.HttpServletBean.init(139): Servlet 'dispatcher' configured ↩
    successfully
Nov 29, 2016 5:08:41 PM org.apache.coyote.AbstractProtocol start
INFO: Starting ProtocolHandler [http-nio-8080]
Nov 29, 2016 5:08:41 PM org.apache.coyote.AbstractProtocol start
INFO: Starting ProtocolHandler [ajp-nio-8009]
Nov 29, 2016 5:08:41 PM org.apache.catalina.startup.Catalina start
INFO: Server startup in 2010 ms
```

This indicates that your webapp has been deployed into the server successfully and is available. Launch your favourite browser and type https://localhost:8080/example/mvc/listPersons to address bar. Following page will be displayed listing persons in the application.

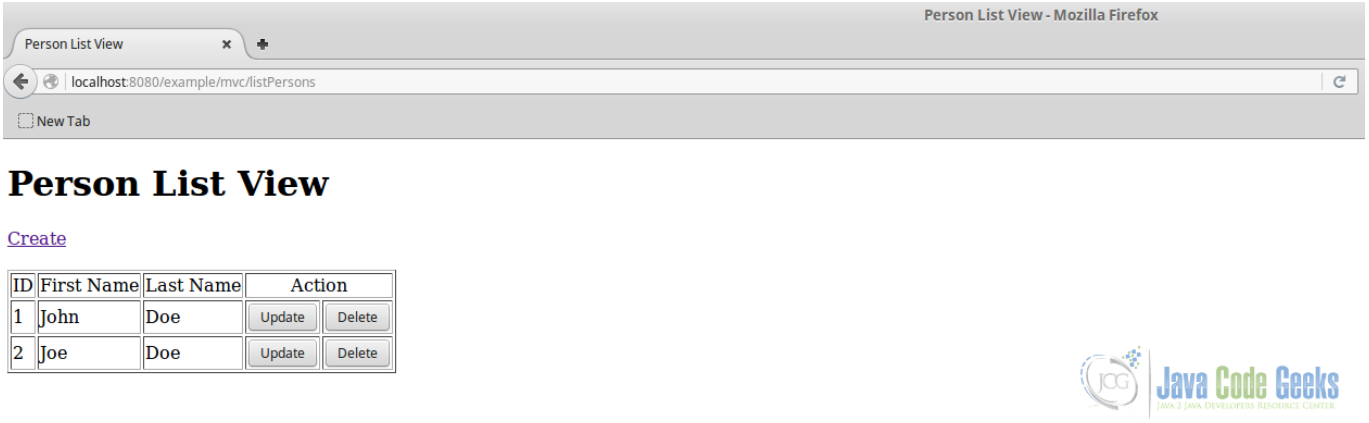

Figure 5.16: access to example web app

You can create a new person, update or delete existing ones through those links and buttons shown on the page.

5.10 Summary

In this example, we created a maven web application project with webapp archetype, created domain class, classes corresponding to dao, service and controller layers, and JSP files to interact with user. After creation of necessary classes, we configured our web application to bootstrap with Spring, and deployed it into Tomcat to run.

5.11 Download the Source Code

Download

You can download the full source code of this example here: **HowToStartDevelopingWebappsWithSpring**

Chapter 6

Angularjs and Spring Integration Tutorial

HTML5, rich browser-based features, and the single page application are extremely valuable tools for modern development. Every application requires a server side (backend) framework besides the client side (frontend) framework.

The server side can serve content, render dynamic HTML, authenticate users, secure access to protect resources or interact with Javascript in the browser through HTTP and JSON. Spring has been always one of the popular server sides framework in java world.

On the other hand, Angularjs becomes popular for the single page applications in the client side. In this tutorial, we show how this two framework integrate easily and works together.

6.1 What is Spring?

The **Spring Framework** is a lightweight solution for enterprise applications. Spring is modular and allows you to use only those parts that you need, without having to bring in the rest. Spring is designed to be non-intrusive, meaning that your domain logic code generally has no dependencies on the framework itself. We show here how to integrate the spring easily with Angularjs in presentation layer.

6.2 What Is Angular?

AngularJS is a structural framework for dynamic web apps. It lets you use HTML as your template language and lets you extend HTML's syntax to express your application's components clearly and succinctly. Angular's data binding and dependency injection eliminate much of the code you would otherwise have to write. And it all happens within the browser, making it an ideal partner with any server technology.

6.3 Create a New Project

Now, lets create a project and go through more details. The following project is created in IntellijIDEA 15 CE. The project developed based on jdk 1.8 and uses maven 3 and tomcat 7.

First, create a Maven project in your IDEA. The source directory of the project should be as below.

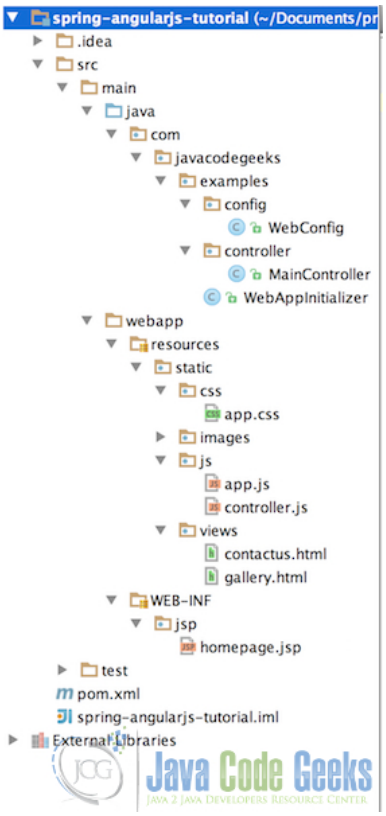

Figure 6.1: Web application directory

6.3.1 Maven dependencies

The first step is to configure the `pom.xml` file to include all required dependencies in the project. In this tutorial, we use **spring-context-4.2.4-RELEASE** and **spring-webmvc-4.2.4-RELEASE** to configure the spring. Also, we use webjars libraries to include all required js files.

WebJars is simply taking the concept of a JAR and applying it to client-side libraries or resources. For example, the `Angularjs` library may be packaged as a JAR and made available to your Spring MVC application. Many WebJars are available through Maven Central with a GroupID of `org.webjars`. A complete list is available at webjars.org.

JavaScript package management is not a new concept. In fact, npm and bower are two of the more popular tools, and currently offer solutions to managing JavaScript dependencies. Spring's Understanding JavaScript Package Managers guide has more information on these. Most JavaScript developers are likely familiar with npm and bower and make use of those in their projects. However, WebJars utilizes Maven's dependency management model to include JavaScript libraries in a project, making it more accessible to Java developers.

This tutorial illustrates how simple it is to use WebJars in your Spring MVC application, and how WebJars provide a convenient way of managing JavaScript packages and dependencies.

pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="https://maven.apache.org/POM/4.0.0"
         xmlns:xsi="https://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="https://maven.apache.org/POM/4.0.0 https://maven.apache.org/ ↩
             xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <packaging>war</packaging>
    <groupId>spring-angularjs-tutorial</groupId>
```

```
    <artifactId>spring-angularjs-tutorial</artifactId>
    <version>1.0-SNAPSHOT</version>

    <dependencies>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-context</artifactId>
            <version>4.2.4.RELEASE</version>
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-webmvc</artifactId>
            <version>4.2.4.RELEASE</version>
        </dependency>
        <dependency>
            <groupId>org.webjars</groupId>
            <artifactId>angularjs</artifactId>
            <version>1.4.8</version>
            <scope>runtime</scope>
        </dependency>
        <dependency>
            <groupId>org.webjars</groupId>
            <artifactId>bootstrap</artifactId>
            <version>3.3.6</version>
            <scope>runtime</scope>
        </dependency>
        <dependency>
            <groupId>javax.servlet</groupId>
            <artifactId>javax.servlet-api</artifactId>
            <version>3.1.0</version>
            <scope>provided</scope>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>3.3</version>
                <configuration>
                    <source>1.8</source>
                    <target>1.8</target>
                </configuration>
            </plugin>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-war-plugin</artifactId>
                <version>2.6</version>
                <configuration>
                    <failOnMissingWebXml>false</failOnMissingWebXml>
                </configuration>
            </plugin>
            <plugin>
                <groupId>org.apache.tomcat.maven</groupId>
                <artifactId>tomcat7-maven-plugin</artifactId>
                <version>2.2</version>
                <configuration>
                    <path>/</path>
                </configuration>
            </plugin>
        </plugins>
```

```
    </build>
</project>
```

maven-compiler-plugin is used to compile the project and maven-war-plugin is used to build the war file. As we configure the web app in java file in this tutorial, there is no web.xml file in the source directory. So, the following configuration is required in the maven-war-plugin to prevent any further exceptions regarding missing web.xml file <failOnMissingWebXml>false</failOnMissingWebXml>.

Another plugin that is used is tomcat7-mabven-plugin, to run the application without installing any tomcat server (You can install the tomcat and deploy the project in your tomcat).

6.3.2 Web app java-based configuration

Since Spring 3, WebApplicationInitializer is implemented in order to configure the ServletContext programmatically in replacement of the WEB-INF/web.xml file. Most Spring users building a web application will need to register Spring's DispatcherServlet. Here is the equivalent DispatcherServlet registration logic in form of a java class.

WebAppInitializer.java

```
package com.javacodegeeks.examples;

import org.springframework.web.WebApplicationInitializer;
import org.springframework.web.context.support.AnnotationConfigWebApplicationContext;
import org.springframework.web.servlet.DispatcherServlet;

import javax.servlet.ServletContext;
import javax.servlet.ServletException;
import javax.servlet.ServletRegistration;

public class WebAppInitializer implements WebApplicationInitializer {

    private static final String CONFIG_LOCATION = "com.javacodegeeks.examples.config";

    @Override
    public void onStartup(ServletContext servletContext) throws ServletException {

        System.out.println("***** Initializing Application for " + servletContext. ↩
            getServerInfo() + " *****");

        // Create ApplicationContext
        AnnotationConfigWebApplicationContext applicationContext = new  ↩
            AnnotationConfigWebApplicationContext();
        applicationContext.setConfigLocation(CONFIG_LOCATION);

        // Add the servlet mapping manually and make it initialize automatically
        DispatcherServlet dispatcherServlet = new DispatcherServlet(applicationContext);
        ServletRegistration.Dynamic servlet = servletContext.addServlet("mvc-dispatcher",  ↩
            dispatcherServlet);

        servlet.addMapping("/");
        servlet.setAsyncSupported(true);
        servlet.setLoadOnStartup(1);
    }
}
```

The following class extends WebMvcConfigurerAdapter to customise the java-based configuration for SpringMVC. It is as opposed to the mvc-dispatcher.xml. To configure the resources and viewResolver, addResourceHandlers and getViewResolver are overridden as below.

WebConfig.java

```
package com.javacodegeeks.examples.config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.servlet.ViewResolver;
import org.springframework.web.servlet.config.annotation.DefaultServletHandlerConfigurer;
import org.springframework.web.servlet.config.annotation.EnableWebMvc;
import org.springframework.web.servlet.config.annotation.ResourceHandlerRegistry;
import org.springframework.web.servlet.config.annotation.WebMvcConfigurerAdapter;
import org.springframework.web.servlet.view.InternalResourceViewResolver;

@EnableWebMvc
@Configuration
@ComponentScan("com.javacodegeeks.examples")
public class WebConfig extends WebMvcConfigurerAdapter{

    @Override
    public void addResourceHandlers(ResourceHandlerRegistry registry) {
        registry.addResourceHandler("/resources/static/js/**")
                .addResourceLocations("/resources/static/js/");
        registry.addResourceHandler("/resources/static/css/**")
                .addResourceLocations("/resources/static/css/");
        registry.addResourceHandler("/resources/static/views/**")
                .addResourceLocations("/resources/static/views/");
        registry.addResourceHandler("/resources/static/**")
                .addResourceLocations("/resources/static/");
        registry.addResourceHandler("/webjars/**")
                .addResourceLocations("/webjars/");
    }

    @Bean
    public ViewResolver getViewResolver(){
        InternalResourceViewResolver resolver = new InternalResourceViewResolver();
        resolver.setPrefix("/WEB-INF/jsp/");
        resolver.setSuffix(".jsp");
        return resolver;
    }

    @Override
    public void configureDefaultServletHandling(DefaultServletHandlerConfigurer configurer) ↩
         {
        configurer.enable();
    }
}
```

6.3.3 SpringMVC controller and jsp

The following class is only a simple controller which is implemented to handle the request to `'/'` and render the request to `homepage.jsp`.

MainController.java

```
package com.javacodegeeks.examples.controller;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;

@Controller
public class MainController {
```

```
    @RequestMapping("/")
    public String homepage(){
        return "homepage";
    }
}
```

In the `homepage.jsp`, there are some front end code to display links in the page which handled by `Angularjs`. Also, there are some script tags which included all required Angularjs js files.

homepage.jsp

```
<!DOCTYPE html>
<!--[if lt IE 7]>      <html lang="en" ng-app="app" class="no-js lt-ie9 lt-ie8 lt-ie7"> <![ ↩
    endif]-->
<!--[if IE 7]>         <html lang="en" ng-app="app" class="no-js lt-ie9 lt-ie8"> <![endif] ↩
    -->
<!--[if IE 8]>         <html lang="en" ng-app="app" class="no-js lt-ie9"> <![endif]-->
<!--[if gt IE 8]><!--> <html lang="en" ng-app="app" class="no-js"> <!--<![endif]-->
<head>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <title>Spring and Angularjs Tutorial</title>
    <meta name="description" content="">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <link rel="stylesheet" href="../resources/static/css/app.css">
</head>
<body>
<h2>Spring and Angularjs Tutorial</h2>

    <ul class="menu-list">
        <li><a href="#/gallery">Photo Gallery</a></li>
        <li><a href="#/contactus">Contact</a></li>
    </ul>


<script src="./webjars/angularjs/1.4.8/angular.js"></script>
<script src="./webjars/angularjs/1.4.8/angular-resource.js"></script>
<script src="./webjars/angularjs/1.4.8/angular-route.js"></script>
<script src="../resources/static/js/app.js"></script>
<script src="../resources/static/js/controller.js"></script>
<link rel="stylesheet" href="./webjars/bootstrap/3.3.6/css/bootstrap.css">
</body>
</html>
```

`ng-view` is a directive that complements the `$route` service by including the rendered template of the current route into the main layout. Every time the current route changes, the included view changes with it according to the configuration of the `$routeProvider`.

6.3.4 Angularjs controllers and js files

`app.js` file defines the application module configuration and routes. To handle a request to e.g. `'/'`, it needs an Angularjs module, called ngRoute. To use `ngRoute` and inject it into our application. We use `angular.module` to add the `ngRoute` module to our app as shown below.

app.js

```
var app = angular.module('app', ['ngRoute','ngResource']);
app.config(function($routeProvider){
    $routeProvider
        .when('/gallery',{
```

```
            templateUrl: 'resources/static/views/gallery.html',
            controller: 'galleryController'
        })
        .when('/contactus',{
            templateUrl: 'resources/static/views/contactus.html',
            controller: 'contactusController'
        })
        .otherwise(
            { redirectTo: '/'}
        );
});
```

Then, in the `app.config`, each route is mapped to a template and controller.

`Controller.js` contains implementation of controllers. The **controller** is simply a constructor function that takes a `$scope` parameter. You might notice that we are injecting the `$scope` service into our controller. Yes, AngularJS comes with a dependency injection container built in to it.

Here, a headingtitle is set in scope to display in the view, either gallery or contactInfo.

controller.js

```
app.controller('galleryController', function($scope) {
    $scope.headingTitle = "Photo Gallery Items";
});

app.controller('contactusController', function($scope) {
    $scope.headingTitle = "Contact Info";
});
```

The concept of a scope in Angular is crucial. A scope can be seen as the glue which allows the template, model and controller to work together. Angular uses scopes, along with the information contained in the template, data model, and controller, to keep models and views separate, but in sync. Any changes made to the model are reflected in the view; any changes that occur in the view are reflected in the model. contactus.html

```
    <h3>{{headingTitle}}</h3>
    <div>
        <ul type="disc">
            <li>Address: Unit 10, Sydney, NSW, Australia</li>
            <li>Phone: 1111 2222</li>
            <li>Fax: 4444 5555</li>
        </ul>

</div>
```

In this two html files, you can see the `{{headingTitle}}` which will be filled later by the value that is set in scope.

gallery.html

```
    <h3>{{headingTitle}}</h3>
    <div class="gallery-section">
        <img class="photo" src="./resources/static/images/images.jpeg"/>
        <img class="photo" src="./resources/static/images/images2.jpeg">

</div>
```

6.3.5 Build and run the application on tomcat

Now, it is time to deploy and run the project. To do so, go to the project directory and run:

```
mvn clean install
```

Then, run the application on tomcat.

```
mvn tomcat7:run
```

And you can navigate the project as below.

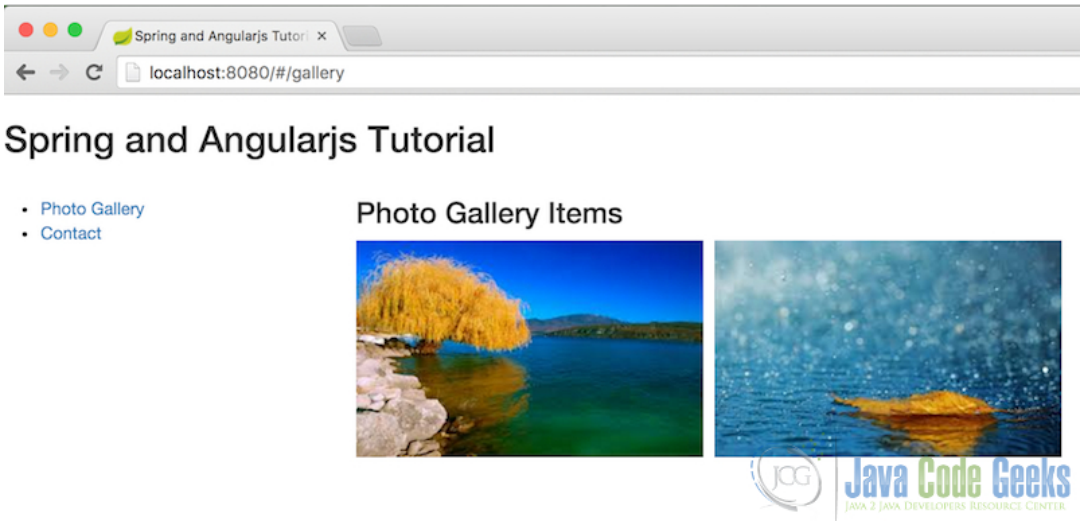

Figure 6.2: Angularjs and Spring integration web app

6.4 Download the source code

This was a Tutorial about Angularjs and Spring Integration.

Download

You can download the full source code of this example here: **Angularjs and Spring integration tutorial**

Chapter 7

Spring MVC Application with Spring Security Example

In one of our past examples, we learned to create a simple Spring MVC web-applciation. In this example we will demonstrate how we can implement Spring-Security to secure our web-application. We shall discuss and demonstrate both Authentication as well as the Authorization aspect of an application's security.

7.1 Introduction to Spring Security

Security of a web-application revolves around three major concepts :

- Authentication
- Authorization
- Encryption

First let's understand What is Authentication and Authorization?

- **Authentication** is the process of determining if the user is, who he claims to be. If the user enters his username as XYZ, then he should be able to prove that he is XYZ by providing the password known only to user XYZ.
- **Authorization** is usually the next step after authentication wherein the system determines if the authenticated user is privileged to access the resource requested.

We shall leave out Encryption as it is beyond the scope of this write-up.

Spring Security provides authentication and authorization in a very flexible manner and is also easy to configure and interpret. Let's start with project setup.

7.2 Project Setup

We shall use Maven to setup our project. Open Eclipse and create a simple Maven project and check the skip archetype selection checkbox on the dialogue box that appears. Replace the content of the existing `pom.xml` with the one provided below:

pom.xml

```
<project xmlns="https://maven.apache.org/POM/4.0.0" xmlns:xsi="https://www.w3.org/2001/ ↩
    XMLSchema-instance"
        xsi:schemaLocation="https://maven.apache.org/POM/4.0.0 https://maven.apache.org/ ↩
            maven-v4_0_0.xsd">
        <modelVersion>4.0.0</modelVersion>
        <groupId>com.javacodegeeks.examples</groupId>
        <artifactId>SpringWebwithSpringSecurity</artifactId>
        <packaging>war</packaging>
        <version>0.0.1-SNAPSHOT</version>
        <name>SpringWebwithSpringSecurity Maven Webapp</name>
        <url>https://maven.apache.org</url>
        <dependencies>
                <dependency>
                        <groupId>org.springframework</groupId>
                        <artifactId>spring-core</artifactId>
                        <version>4.2.3.RELEASE</version>
                </dependency>

                <dependency>
                        <groupId>taglibs</groupId>
                        <artifactId>standard</artifactId>
                        <version>1.1.2</version>
                </dependency>


                <dependency>
                        <groupId>org.springframework.security</groupId>
                        <artifactId>spring-security-core</artifactId>
                        <version>4.0.3.RELEASE</version>
                </dependency>

                <dependency>
                        <groupId>org.springframework.security</groupId>
                        <artifactId>spring-security-web</artifactId>
                        <version>4.0.3.RELEASE</version>
                </dependency>

                <dependency>
                        <groupId>org.springframework.security</groupId>
                        <artifactId>spring-security-config</artifactId>
                        <version>4.0.3.RELEASE</version>
                </dependency>

                <dependency>
                        <groupId>javax.servlet.jsp.jstl</groupId>
                        <artifactId>javax.servlet.jsp.jstl-api</artifactId>
                        <version>1.2.1</version>
                        <scope>compile</scope>
                </dependency>

                <dependency>
                        <groupId>javax.servlet.jsp.jstl</groupId>
                        <artifactId>jstl-api</artifactId>
                        <version>1.2</version>
                </dependency>



                <dependency>
                        <groupId>org.springframework</groupId>
                        <artifactId>spring-webmvc</artifactId>
```

```
                        <version>4.2.3.RELEASE</version>
                </dependency>

                <dependency>
                        <groupId>org.springframework</groupId>
                        <artifactId>spring-web</artifactId>
                        <version>4.2.3.RELEASE</version>
                </dependency>

        </dependencies>
        <build>
                <finalName>SpringWebwithSpringSecurity</finalName>
                <pluginManagement>
                        <plugins>
                                <plugin>
                                        <groupId>org.apache.maven.plugins</groupId>
                                        <artifactId>maven-compiler-plugin</artifactId>
                                        <configuration>
                                                <source>1.8</source>
                                                <target>1.8</target>
                                        </configuration>
                                </plugin>
                        </plugins>
                </pluginManagement>
        </build>
        <properties>
                <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        </properties>

</project>
```

This will import the required JAR dependencies in the project. We can now start with the actual Spring-Security implementation.

7.3 Project Implementation

Figure 7.1: Project Structure

Let's start with the gateway of the J2EE web-application, the `WEB.xml`. We need to declare the `SpringSecurityFilterChain`.

web.xml

```
<web-app xmlns="https://java.sun.com/xml/ns/javaee" xmlns:xsi="https://www.w3.org/2001/ ↩
    XMLSchema-instance"
        xsi:schemaLocation="https://java.sun.com/xml/ns/javaee
          https://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
        version="3.0">

        <display-name>Servlet 3.0 Web Application</display-name>
        <display-name>Spring Security Example</display-name>

        <context-param>
                <param-name>contextConfigLocation</param-name>
                <param-value>
                        classpath:/security-config.xml
                </param-value>
        </context-param>

        <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>

        <listener>
                <listener-class>org.springframework.web.context.ContextLoaderListener</ ↩
                    listener-class>
        </listener>

        <servlet>
                <servlet-name>Spring-Controller</servlet-name>
                <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet- ↩
                    class>
                <init-param>
                        <param-name>contextConfigLocation</param-name>
                        <param-value>classpath:/springWeb.xml</param-value>
                </init-param>
                <load-on-startup>1</load-on-startup>
        </servlet>
        <servlet-mapping>
                <servlet-name>Spring-Controller</servlet-name>
                <url-pattern>/</url-pattern>
        </servlet-mapping>

        <welcome-file-list>
                <welcome-file>jsp/login.jsp</welcome-file>
        </welcome-file-list>

</web-app>
```

Spring Security intercepts the incoming request via a Servlet filter - `springSecurityFilterChain`. The `DelegatingFilterProxy` is a proxy for actual spring bean object which implements the `javax.servlet.Filter` interface. This filter guards the web-application from a host of malicious attacks like CSRF , Session Fixation, XSS etc.

We pass the location of spring security config file - `security-config.xml` to the filter via the `contextConfigLocation` web context parameter. Let's have a look at `security-config.xml`:

security-config.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans:beans xmlns="https://www.springframework.org/schema/security"
        xmlns:xsi="https://www.w3.org/2001/XMLSchema-instance" xmlns:beans="https://www. ↩
            springframework.org/schema/beans"
        xmlns:sec="https://www.springframework.org/schema/security"
        xmlns:context="https://www.springframework.org/schema/context"
        xsi:schemaLocation="
      https://www.springframework.org/schema/security
      https://www.springframework.org/schema/security/spring-security-4.0.xsd
      https://www.springframework.org/schema/beans
      https://www.springframework.org/schema/beans/spring-beans-4.0.xsd
      https://www.springframework.org/schema/context
          https://www.springframework.org/schema/context/spring-context-4.0.xsd">

        <sec:http auto-config="true" use-expressions="true">
                <sec:form-login login-page="/login"
                        login-processing-url="/authenticateUser" default-target-url="/ ↩
                            welcome"
                        authentication-failure-url="/login" username-parameter="username"
                        password-parameter="password" />

                <sec:access-denied-handler error-page="/403.jsp" />

                <sec:intercept-url pattern="/login" access="permitAll" />
```

```
                <sec:intercept-url pattern="/**" access="hasAuthority('AUTH_USER')" />
                <sec:session-management invalid-session-url="/login" />
                <sec:logout delete-cookies="JSESSIONID" logout-url="/logout" />
        </sec:http>

        <context:component-scan base-package="com.jcg.examples" />

        <sec:authentication-manager>
                <authentication-provider ref="customAuthenticationProvider" />
        </sec:authentication-manager>

</beans:beans>
```

This is the file where we configure the actual security parameters for our application. It acts as a container for all HTTP-related security settings.

`sec:form-login` is the login form shown to the user when he tries to access any resource in the web-application. If we do not provide a login form, the spring provides its default login page with a username, password fields and submit button. The `username-parameter` and `password-parameter` are the names of the username and the password fields that the login page has. When these attributes are not explicitly provided they default to `j_username` and `j_password`. It is wise to rename to hide underlying technology. Spring extracts the username and password from the request using the names provided and provides them in the `org.springframework.security.core.Authentication` object.

The `login-processing-url` is the actual `url` which holds the resource to authenticate the user. We have defined a custom authenticator class and mapped it to `/authenticateUser` URL. We will look in to this class in detail in the next section.

The developer may define multiple `sec:intercept-url`. This specifies the roles authorized to access the resource mapped by this filter-pattern. The user may also use `hasRole` expression to authenticate based in the user-roles, but in that case the role-name must start with `ROLE_` or else the user is denied the access. The user may also choose to waive authentication process for certain resources from all security check like the `login` page, Javascript and CSS files. Not doing so may lead to infinite redirects to the same login page.

`sec:logout` tag is used to customize the logging-out process from the web-application.

The `sec:authentication-manager` is the actual bean that authenticates the user based on the username and password he enters in the login page. Here's how the custom authenticator class looks like:

CustomAuthenticationProvider.java

```
package com.jcg.examples.authentication;


import java.util.ArrayList;
import java.util.List;

import org.springframework.security.authentication. ↩
    AuthenticationCredentialsNotFoundException;
import org.springframework.security.authentication.AuthenticationProvider;
import org.springframework.security.authentication.UsernamePasswordAuthenticationToken;
import org.springframework.security.core.Authentication;
import org.springframework.security.core.AuthenticationException;
import org.springframework.security.core.GrantedAuthority;
import org.springframework.stereotype.Component;


@Component
public class CustomAuthenticationProvider implements AuthenticationProvider
{
                @Override
                public Authentication authenticate(Authentication authentication) throws  ↩
                    AuthenticationException
                {
                                String userName = authentication.getName();
```

```
                                String password = authentication.getCredentials().toString ↩
                                    ();

                                if (authorizedUser(userName, password))
                                {
                                                List<GrantedAuthority> grantedAuths = new  ↩
                                                    ArrayList<>();
                                                grantedAuths.add(()-> {return "AUTH_USER" ↩
                                                    ;});
                                                Authentication auth = new  ↩
                                                    UsernamePasswordAuthenticationToken( ↩
                                                    userName, password, grantedAuths);
                                                System.out.println(auth.getAuthorities());
                                                return auth;
                                }
                                else
                                {
                                                throw new  ↩
                                                    AuthenticationCredentialsNotFoundException ↩
                                                    ("Invalid Credentials!");
                                }
                }

                private boolean authorizedUser(String userName, String password)
                {
                                System.out.println("username is :" + userName+" and  ↩
                                    password is "+password );
                                if("Chandan".equals(userName) && "Chandan".equals(password) ↩
                                    )
                                                return true;
                                return false;
                }

                @Override
                public boolean supports(Class<?> authentication)
                {
                                return UsernamePasswordAuthenticationToken.class. ↩
                                    isAssignableFrom(authentication);
                }
}
```

Our custom authenticator class implements the `org.springframework.security.authentication.AuthenticationProvider` interface. The interface provides us with simple method which will help simplify the process of user-authentication for us.

`authenticate(Authentication authentication)` : This method takes the authentication request object as a parameter. This object contains the username and password the user entered in the login page. Upon successful authentication the users roles are populated in a new `org.springframework.security.authentication.UsernamePasswordAuthenticationToken` authentication object. The resource requested by the user is then checked against the role in this authentication object. If the role matches the access rights for the user is allowed to access the resource. If not, the user is redirected to the error-page defined in the `sec:access-denied-handler` tag.

In this example we have implemented the `org.springframework.security.core.GrantedAuthority` interface using the lambda expression and provided the user with the `AUTH_USER` role.

Once the user is successfully authenticated and authorized, the url is directed to `DispatcherServlet` configured in the `web.xml`. The `DispatcherServlet` in turn invokes the `Controller` method mapped to the resource's url.

Here's a simple configuration xml for the initialization of the controllers. Remember to pass this xml file-name in the `init-param` of the `DispatcherServlet` in `web.xml`.

springWeb.xml

```
<beans xmlns="https://www.springframework.org/schema/beans"
        xmlns:context="https://www.springframework.org/schema/context"
        xmlns:xsi="https://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="
   https://www.springframework.org/schema/beans
   https://www.springframework.org/schema/beans/spring-beans-3.0.xsd
   https://www.springframework.org/schema/context
   https://www.springframework.org/schema/context/spring-context-3.0.xsd">

        <context:component-scan base-package="com.jcg.examples" />

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

</beans>
```

We have also configured the JSP view-resolver for the view resolution. Let's see have a look at the JSP files:

login.jsp

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
        pageEncoding="ISO-8859-1"%>
<%@taglib uri="https://java.sun.com/jsp/jstl/core" prefix="c"%>
<%@taglib uri="https://www.springframework.org/tags/form" prefix="form"%>
<%@taglib uri="https://www.springframework.org/tags" prefix="spring"%>
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Login</title>
</head>
<body>
<c:if test="${not empty SPRING_SECURITY_LAST_EXCEPTION}">
      <font color="red">
        Your login attempt was not successful due to <c:out value="${ ↩
            SPRING_SECURITY_LAST_EXCEPTION.message}"/>.
      </font>
</c:if>
        <form name="loginForm" action="authenticateUser" method="post">
                User-name<input type="text" name="username" /><br /> Password <input
                        type="password" name="password" /> <input type="hidden"
                        name="${_csrf.parameterName}" value="${_csrf.token}" /> <input
                        type="submit" value="Submit">
        </form>
</body>
</html>
```

As I have already explained, the names of the username and the password have been configured in the `sec:form-login` tag of the `security-config.xml` as is the authentication URL. There is also a hidden field which stores a random token to be submitted with the request. This helps to guard against the `https://en.wikipedia.org/wiki/Cross-site_request_forgery[CSRF]` attack.

Here's the `login.jsp` in browser:

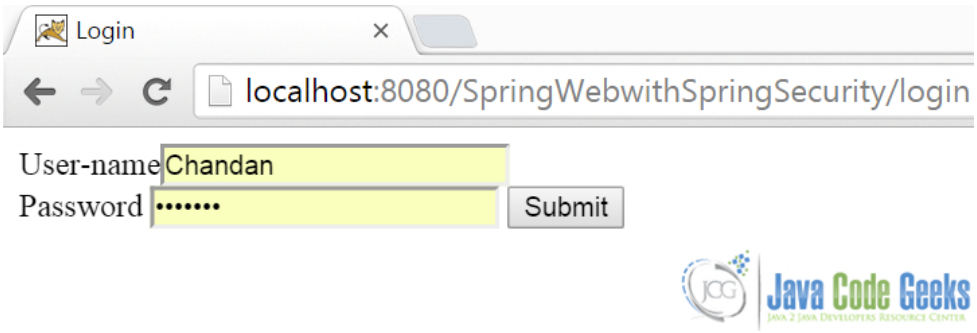

Figure 7.2: Login Page

Upon successful authentication, the user is rendered the `welcome.jsp`

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
    <%@taglib uri="https://java.sun.com/jsp/jstl/core" prefix="c"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "https://www.w3.org/TR/html4 ↩
    /loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>WELCOME</title>
</head>
<body>
Welcome! Your login was successful...!

<a href="<c:url value="/logout" />">Logout</a>
</body>
</html>
```

Here's how it looks:

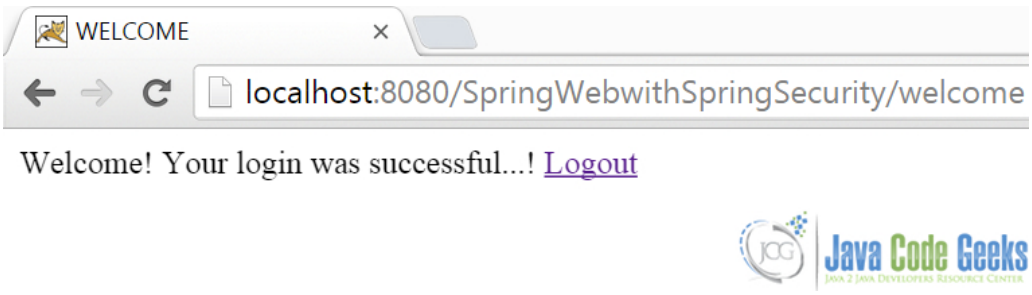

Figure 7.3: Welcome Page

In case the user enters wrong password he is redirected back to the login page with the message that is thrown from the `CustomAuthenticationProvider#authenticate` method. We can access the message using the `SPRING_SECURITY_LAST_EXCEPTION` variable in the `login.jsp`.

If the user is authenticated but his role does not allow him to access the resource, he is re-directed to the Access Denied page as shown here:

Figure 7.4: Access Denied

7.4 Download the Source Code

Here, we studied how we can user spring-security to enable access control in our web-application.

Download

You can download the source code of this example here: **SpringWebwithSpringSecurity.zip**

Chapter 8

Spring MVC Hibernate Tutorial

8.1 Introduction

To develop web applications these days, we use Modern View Controller architecture. Spring provides MVC framework with ready components that can be used to develop flexible and loosely coupled web applications. MVC framework provides separation between input logic, business logic and UI logic.

- Model encapsulates the application data
- View is responsible for rendering the model data
- Controller is responsible for processing user requests and building model and passing it to view for rendering

8.2 Environment

We will use the following environment for demo of Spring MVC with Hibernate example.

- Windows 7
- Java version 8
- Eclipse Kepler 4.3
- Maven 3.0.4
- MySQL 5.0.86
- Hibernate 4.3.6 Final
- Tomcat 7.0.64
- Spring 4.1.9 Release
- MySQL JDBC Connector 5.0.4

8.3 Spring MVC Framework

As stated in introduction of this section, Spring MVC framework comprised of three logical sections of Model, View and Controller. This framework is designed around a `DispatcherServlet` that receives and sends all HTTP requests and responses. The sequence of events to an incoming HTTP request to `DispatcherServlet` is

- DispatcherServlet communicates with `HandlerMapping` to call the appropriate Controller once it receives an HTTP request.
- Controller takes the request and calls the appropriate service methods based on used GET or POST method. The service method using Model data will return view name to DispatcherServlet
- DispatcherServlet will send that view name to ViewResolver to return the appropriate view for the request.
- In selected View, DispatcherServlet will send the model data to display rendered page in browser

8.4 Hibernate For Model

Hibernate maps Java classes to database tables and from Java data types to SQL data types. Hibernate lies between relational database and Java objects to handle all the work in persisting those objects based on accurate O/R configuration.

Hibernate provides following advantages

- Hibernate handles all the mapping of java classes to database tables using XML configuration without writing any code.
- It provides APIs for storing and retrieving objects directly to and from the database.
- If there is a change in database or in any table, you only need to change in XML configuration file.
- Hibernate does not require an application server to operate.
- Minimize database access with smart fetching strategies.
- Provides simply querying of data.

For this example, we will use hibernate to build our model. Our model for this example will be based on Employee and Company.

8.5 Example

In this example, we will configure Spring with Hibernate. We will be writing a simple CRUD web application with a web form asking user input to save information in MySQL database using Hibernate. We will have few options to show the database data on webpage.

8.5.1 Maven Project and POM dependencies

For this example, we will create a Dynamic Web project and then convert that into Maven Project. In eclipse, create a new Dynamic Web Project with name `SpringMVCSampleApp` and select Apache Tomcat 7.0 for Target Runtime as shown in below picture.

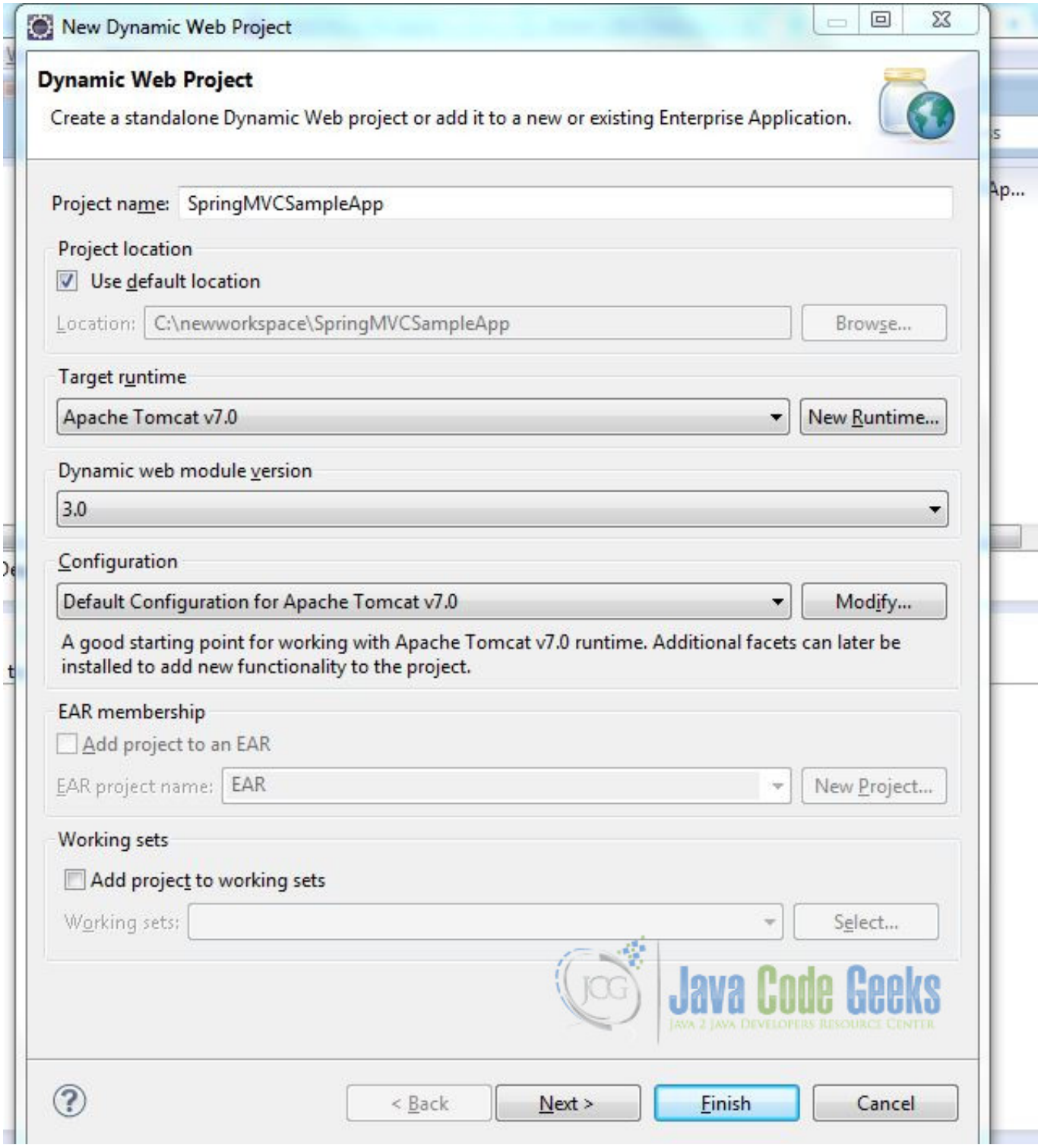

Figure 8.1: Dynamic Web Project - SpringMVCSampleApp

To convert this Dyanmic Web Project to Maven Web Project, on next screen, create a directory structure as shown in picture below. This is required for Maven project.

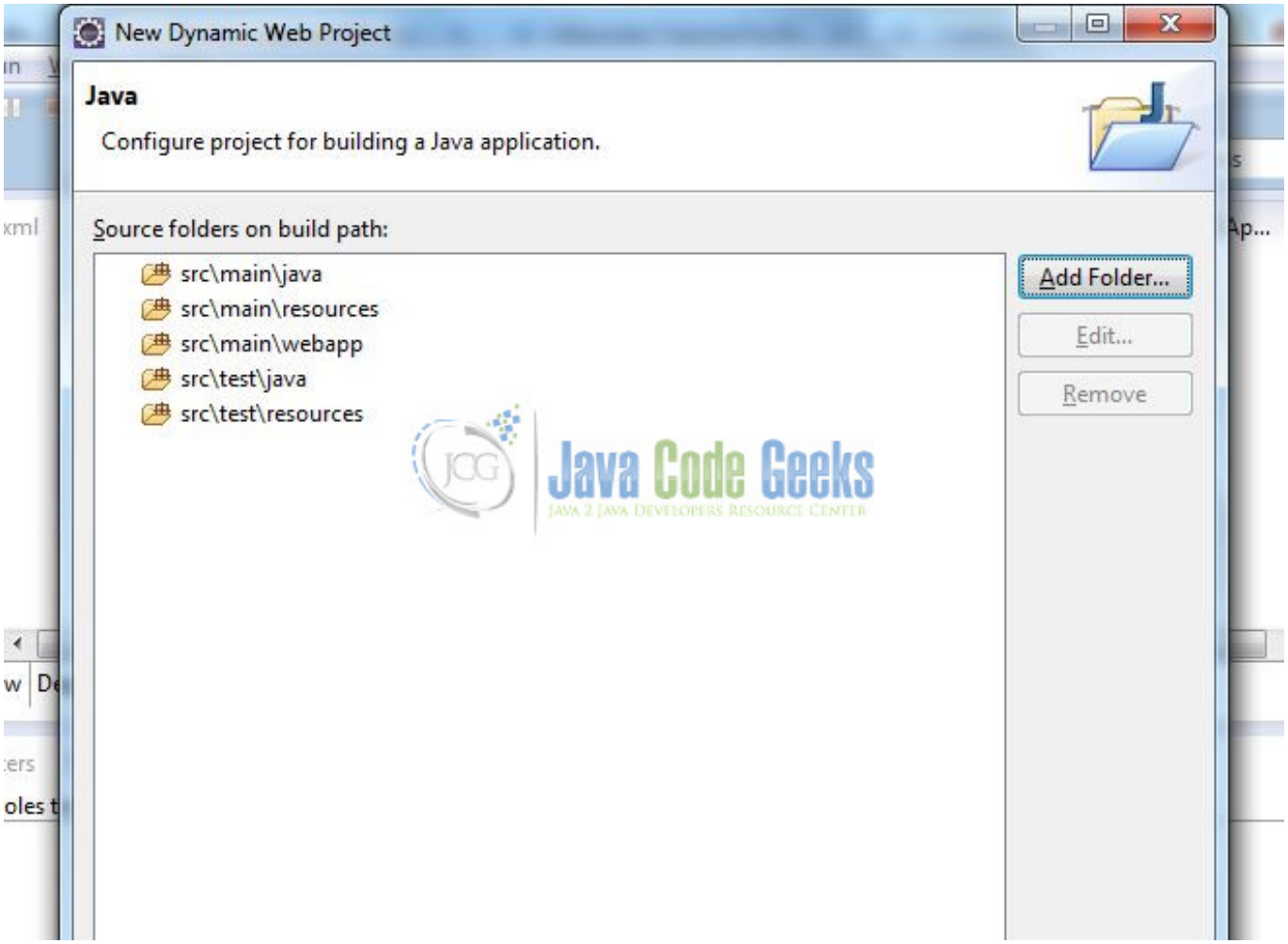

Figure 8.2: Directory Structure

On next screen, select `Generate web.xml deployment descriptor` option and click Finish. Now right click on the project in eclipse and select option Configure → Convert To Maven Project. Select the default option of WAR for packaging. This will create a POM dependencies xml file in the project. Move all the contents from WebContent folder in project directory in eclipse to `src/main/webapp` directory. You can delete the WebContent folder.

We will update pom.xml file to add the required dependencies for this project. This is how the final pom.xml file will look:

pom.xml

```
<project xmlns="https://maven.apache.org/POM/4.0.0" xmlns:xsi="https://www.w3.org/2001/ ↩
    XMLSchema-instance" xsi:schemaLocation="https://maven.apache.org/POM/4.0.0 https://maven ↩
    .apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>SpringMVCSampleApp</groupId>
  <artifactId>SpringMVCSampleApp</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <packaging>war</packaging>
  <build>
    <plugins>
      <plugin>
        <artifactId>maven-compiler-plugin</artifactId>
        <version>3.1</version>
        <configuration>
          <source>1.7</source>
```

```
        <target>1.7</target>
      </configuration>
    </plugin>
    <plugin>
      <artifactId>maven-war-plugin</artifactId>
      <version>2.3</version>
      <configuration>
        <warSourceDirectory>WebContent</warSourceDirectory>
        <failOnMissingWebXml>false</failOnMissingWebXml>
      </configuration>
    </plugin>
  </plugins>
</build>
<properties>
      <springframework.version>4.1.9.RELEASE</springframework.version>
      <hibernate.version>4.3.6.Final</hibernate.version>
      <mysql.connector.version>5.0.4</mysql.connector.version>
      <joda-time.version>2.3</joda-time.version>
      <testng.version>6.9.4</testng.version>
</properties>
<dependencies>
<!-- Spring -->
      <dependency>
          <groupId>org.springframework</groupId>
          <artifactId>spring-core</artifactId>
          <version>${springframework.version}</version>
      </dependency>
      <dependency>
          <groupId>org.springframework</groupId>
          <artifactId>spring-web</artifactId>
          <version>${springframework.version}</version>
      </dependency>
      <dependency>
          <groupId>org.springframework</groupId>
          <artifactId>spring-webmvc</artifactId>
          <version>${springframework.version}</version>
      </dependency>
      <dependency>
          <groupId>org.springframework</groupId>
          <artifactId>spring-tx</artifactId>
          <version>${springframework.version}</version>
      </dependency>
      <dependency>
          <groupId>org.springframework</groupId>
          <artifactId>spring-orm</artifactId>
          <version>${springframework.version}</version>
      </dependency>

              <!-- Hibernate -->
      <dependency>
          <groupId>org.hibernate</groupId>
          <artifactId>hibernate-core</artifactId>
          <version>${hibernate.version}</version>
      </dependency>

      <!-- jsr303 validation -->
      <dependency>
          <groupId>javax.validation</groupId>
          <artifactId>validation-api</artifactId>
          <version>1.1.0.Final</version>
      </dependency>
      <dependency>
```

```
            <groupId>org.hibernate</groupId>
            <artifactId>hibernate-validator</artifactId>
            <version>5.1.3.Final</version>
        </dependency>

        <!-- MySQL -->
        <dependency>
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId>
            <version>${mysql.connector.version}</version>
        </dependency>

        <!-- Joda-Time -->
        <dependency>
            <groupId>joda-time</groupId>
            <artifactId>joda-time</artifactId>
            <version>${joda-time.version}</version>
        </dependency>

        <!-- To map JodaTime with database type -->
        <dependency>
            <groupId>org.jadira.usertype</groupId>
            <artifactId>usertype.core</artifactId>
            <version>3.0.0.CR1</version>
        </dependency>

        <!-- Servlet+JSP+JSTL -->
        <dependency>
            <groupId>javax.servlet</groupId>
            <artifactId>javax.servlet-api</artifactId>
            <version>3.1.0</version>
        </dependency>

        <dependency>
            <groupId>javax.servlet.jsp</groupId>
            <artifactId>javax.servlet.jsp-api</artifactId>
            <version>2.3.1</version>
        </dependency>
        <dependency>
            <groupId>javax.servlet</groupId>
            <artifactId>jstl</artifactId>
            <version>1.2</version>
        </dependency>


        <!-- Testing dependencies -->
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-test</artifactId>
            <version>${springframework.version}</version>
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>org.testng</groupId>
            <artifactId>testng</artifactId>
            <version>${testng.version}</version>
            <scope>test</scope>
        </dependency>
    </dependencies>
</project>
```

8.5.2 Configure Hibernate

To configure hibernate, we will be using annotation rather than usual hibernate.cfg.xml. Let's create a class `HibernateConfiguration` in `src` folder with package name `com.javacodegeeks.configuration`. The code will look like below

HibernateConfiguration.java

```
package com.javacodegeeks.configuration;

import java.util.Properties;

import javax.sql.DataSource;

import org.hibernate.SessionFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.PropertySource;
import org.springframework.core.env.Environment;
import org.springframework.orm.hibernate4.HibernateTransactionManager;
import org.springframework.orm.hibernate4.LocalSessionFactoryBean;
import org.springframework.transaction.annotation.EnableTransactionManagement;
import org.springframework.jdbc.datasource.DriverManagerDataSource;



@Configuration
@EnableTransactionManagement
@ComponentScan({ "com.javacodegeeks.configuration" })
@PropertySource(value = { "classpath:application.properties" })

public class HibernateConfiguration {

    @Autowired
    private Environment environment;

    @Bean
    public LocalSessionFactoryBean sessionFactory() {
        LocalSessionFactoryBean sessionFactory = new LocalSessionFactoryBean();
        sessionFactory.setDataSource(dataSource());
        sessionFactory.setPackagesToScan(new String[] { "com.javacodegeeks.model" });
        sessionFactory.setHibernateProperties(hibernateProperties());
        return sessionFactory;
     }

    @Bean
    public DataSource dataSource() {
        DriverManagerDataSource dataSource = new DriverManagerDataSource();
        dataSource.setDriverClassName(environment.getRequiredProperty("jdbc.driverClassName ↩
            "));
        dataSource.setUrl(environment.getRequiredProperty("jdbc.url"));
        return dataSource;
    }

    private Properties hibernateProperties() {
        Properties properties = new Properties();
        properties.put("hibernate.dialect", environment.getRequiredProperty("hibernate. ↩
            dialect"));
        properties.put("hibernate.show_sql", environment.getRequiredProperty("hibernate. ↩
            show_sql"));
        properties.put("hibernate.format_sql", environment.getRequiredProperty("hibernate. ↩
            format_sql"));
```

```
        return properties;
    }

    @Bean
    @Autowired
    public HibernateTransactionManager transactionManager(SessionFactory s) {
       HibernateTransactionManager txManager = new HibernateTransactionManager();
       txManager.setSessionFactory(s);
       return txManager;
    }

}
```

From this, we still have to configure `properties` file in classpath. This file will be `application.properties` and it will look like below

application.properties

```
#DB properties:
jdbc.driverClassName=org.gjt.mm.mysql.Driver
jdbc.url=jdbc:mysql://localhost:3306/springmvc

#Hibernate Configuration:
hibernate.dialect=org.hibernate.dialect.MySQL5InnoDBDialect
hibernate.show_sql=true
hibernate.format_sql=true
#entitymanager.packages.to.scan=com.javacodegeeks
```

8.5.3 Domain Entity Class

In this section, we will create our domain entity class (POJO). This is the object we will use to modify through our sample web application. We will create an Employee entity object and we will use Hibernate annotation and JPA (Java persistence API) annotations to map it to employee database table. Along with creating entity class, make sure to create a database `SpringMVC` and a table `employee`.

Employee.java

```
package com.javacodegeeks.model;

import java.math.BigDecimal;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Table;
import javax.validation.constraints.Digits;
import javax.validation.constraints.NotNull;
import javax.validation.constraints.Size;

import org.hibernate.annotations.Type;
import org.hibernate.validator.constraints.NotEmpty;
import org.joda.time.LocalDate;
import org.springframework.format.annotation.DateTimeFormat;


@Entity
@Table(name="EMPLOYEE")
public class Employee {
```

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

@Size(min=3, max=50)
@Column(name = "NAME", nullable = false)
private String name;

@NotNull
@DateTimeFormat(pattern="dd/MM/yyyy")
@Column(name = "JOINING_DATE", nullable = false)
@Type(type="org.jadira.usertype.dateandtime.joda.PersistentLocalDate")
private LocalDate joiningDate;

@NotNull
@Digits(integer=8, fraction=2)
@Column(name = "SALARY", nullable = false)
private BigDecimal salary;

@NotEmpty
@Column(name = "SSN", unique=true, nullable = false)
private String ssn;

public int getId() {
    return id;
}

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

public String getName() {
    return name;
}

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

public LocalDate getJoiningDate() {
    return joiningDate;
}

public void setJoiningDate(LocalDate joiningDate) {
    this.joiningDate = joiningDate;
}

public BigDecimal getSalary() {
    return salary;
}

public void setSalary(BigDecimal salary) {
    this.salary = salary;
}

public String getSsn() {
    return ssn;
}

public void setSsn(String ssn) {
    this.ssn = ssn;
}
```

```
    @Override
    public int hashCode() {
        final int prime = 31;
        int result = 1;
        result = prime * result + id;
        result = prime * result + ((ssn == null) ? 0 : ssn.hashCode());
        return result;
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj)
            return true;
        if (obj == null)
            return false;
        if (!(obj instanceof Employee))
            return false;
        Employee other = (Employee) obj;
        if (id != other.id)
            return false;
        if (ssn == null) {
            if (other.ssn != null)
                return false;
        } else if (!ssn.equals(other.ssn))
            return false;
        return true;
    }

    @Override
    public String toString() {
        return "Employee [id=" + id + ", name=" + name + ", joiningDate="
                + joiningDate + ", salary=" + salary + ", ssn=" + ssn + "]";
    }

}
```

8.5.4 Service Layer

We will create a Service interface and its implementation. This layer will provide a cohesive and high-level logic to application. Controller of MVC invokes this layer.

EmployeeService.java

```
package com.javacodegeeks.service;

import java.util.List;

import com.javacodegeeks.model.Employee;

public interface EmployeeService {

        Employee findById(int id);

    void saveEmployee(Employee employee);

    void updateEmployee(Employee employee);

    void deleteEmployeeBySsn(String ssn);

    List findAllEmployees();
```

```
    Employee findEmployeeBySsn(String ssn);

    boolean isEmployeeSsnUnique(Integer id, String ssn);

}
```

We will create a implementation class to add the logic for all these methods which will access the persistent layer.

EmployeeServiceImpl.java

```
package com.javacodegeeks.service;

import java.util.List;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

import com.javacodegeeks.dao.EmployeeDAO;
import com.javacodegeeks.model.Employee;

@Service("employeeService")
@Transactional
public class EmployeeServiceImpl implements EmployeeService {

        @Autowired
    private EmployeeDAO dao;

        @Override
        public void deleteEmployeeBySsn(String ssn) {
                // TODO Auto-generated method stub
                dao.deleteEmployeeBySsn(ssn);
        }

        @Override
        public Employee findById(int id) {
                return dao.findById(id);
        }

        @Override
        public void saveEmployee(Employee employee) {
                // TODO Auto-generated method stub
                 dao.saveEmployee(employee);
        }

        @Override
        public void updateEmployee(Employee employee) {
                // TODO Auto-generated method stub
                Employee entity = dao.findById(employee.getId());
        if(entity!=null){
            entity.setName(employee.getName());
            entity.setJoiningDate(employee.getJoiningDate());
            entity.setSalary(employee.getSalary());
            entity.setSsn(employee.getSsn());
        }

        }

        @Override
        public List findAllEmployees() {
                return dao.findAllEmployees();
```

```
        }

        @Override
        public Employee findEmployeeBySsn(String ssn) {
                return dao.findEmployeeBySsn(ssn);
        }

        public boolean isEmployeeSsnUnique(Integer id, String ssn) {
        Employee employee = findEmployeeBySsn(ssn);
        return ( employee == null || ((id != null) && (employee.getId() == id)));
    }

}
```

In your eclipse, you will see number of build errors at this moment and that is because, we have not added DAO layer yet. That we will see in next section.

8.5.5 DAO Layer

DAO (data access object) layer provides access to persistence layer. Service objects invoke this layer. For our application, we will create an abstract class for `AbstractDAO` which will provide us `createEntityCriteria` when we will implement `EmployeeDAO`. This is a generic base class for DAO implementation.

AbstractDAO.java

```
package com.javacodegeeks.dao;

import java.io.Serializable;
import java.lang.reflect.ParameterizedType;

import org.hibernate.Criteria;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.springframework.beans.factory.annotation.Autowired;

public abstract class AbstractDAO {

        private final Class persistentClass;

    @SuppressWarnings("unchecked")
    public AbstractDAO(){
        this.persistentClass =(Class) ((ParameterizedType) this.getClass(). ↩
            getGenericSuperclass()).getActualTypeArguments()[1];
    }

    @Autowired
    private SessionFactory sessionFactory;

    protected Session getSession(){
        return sessionFactory.getCurrentSession();
    }

    @SuppressWarnings("unchecked")
    public T getByKey(PK key) {
        return (T) getSession().get(persistentClass, key);
    }

    public void persist(T entity) {
        getSession().persist(entity);
    }
```

```
    public void delete(T entity) {
        getSession().delete(entity);
    }

    protected Criteria createEntityCriteria(){
        return getSession().createCriteria(persistentClass);
    }
}
```

Now, we will create `EmployeeDAO` interface.

EmployeeDAO.java

```
package com.javacodegeeks.dao;

import java.util.List;

import com.javacodegeeks.model.Employee;

public interface EmployeeDAO {

        Employee findById(int id);

    void saveEmployee(Employee employee);

    void deleteEmployeeBySsn(String ssn);

    List findAllEmployees();

    Employee findEmployeeBySsn(String ssn);
}
```

To implement `EmployeeDAO`, we will extend our abstract class and implement interface.

EmployeeDAOImpl.java

```
package com.javacodegeeks.dao;

import java.util.List;
import org.hibernate.Criteria;
import org.hibernate.Query;
import org.hibernate.criterion.Restrictions;
import org.springframework.stereotype.Repository;

import com.javacodegeeks.model.Employee;

@Repository("employeeDao")
public class EmployeeDAOImpl extends AbstractDAO implements EmployeeDAO {

        @Override
        public void deleteEmployeeBySsn(String ssn) {
                Query query = getSession().createSQLQuery("delete from Employee where ssn = ↩
                     :ssn");
        query.setString("ssn", ssn);
        query.executeUpdate();
        }

        @Override
        public Employee findById(int id) {
                return getByKey(id);
        }

        @Override
```

```
        public void saveEmployee(Employee employee) {

                persist(employee);
        }

        @Override
        public List findAllEmployees() {
                Criteria criteria = createEntityCriteria();
        return (List) criteria.list();
        }

        @Override
        public Employee findEmployeeBySsn(String ssn) {
                Criteria criteria = createEntityCriteria();
        criteria.add(Restrictions.eq("ssn", ssn));
        return (Employee) criteria.uniqueResult();
        }
}
```

8.5.6 Configure Spring MVC

In `AppConfig`, we will implement a method to get a `ViewResolver` which handles our jsp view. We will also be adding ways to handle our error messages through messages.properties.

ApConfig.java

```
package com.javacodegeeks.configuration;

import org.springframework.context.MessageSource;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.support.ResourceBundleMessageSource;
import org.springframework.web.servlet.ViewResolver;
import org.springframework.web.servlet.config.annotation.EnableWebMvc;
import org.springframework.web.servlet.view.InternalResourceViewResolver;
import org.springframework.web.servlet.view.JstlView;

@Configuration
@EnableWebMvc
@ComponentScan(basePackages = "com.javacodegeeks")
public class AppConfig {

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

    @Bean
    public MessageSource messageSource() {
        ResourceBundleMessageSource messageSource = new ResourceBundleMessageSource();
        messageSource.setBasename("messages");
        return messageSource;
    }

}
```

In resources folder, we will create messages.properties file.

messages.properties

```
Size.employee.name=Name must be between {2} and {1} characters long
NotNull.employee.joiningDate=Joining Date can not be blank
NotNull.employee.salary=Salary can not be blank
Digits.employee.salary=Only numeric data with max 8 digits and with max 2 precision is allowed
NotEmpty.employee.ssn=SSN can not be blank
typeMismatch=Invalid format
non.unique.ssn=SSN {0} already exist. Please fill in different value.
```

8.5.7 Initializer Class

This is the class where our web application's request will be sent to handle. `DispatcherServlet` will handle our request.

AppInitializer.java

```
package com.javacodegeeks.configuration;

import javax.servlet.ServletContext;
import javax.servlet.ServletException;
import javax.servlet.ServletRegistration;

import org.springframework.web.WebApplicationInitializer;
import org.springframework.web.context.support.AnnotationConfigWebApplicationContext;
import org.springframework.web.servlet.DispatcherServlet;

public class AppInitializer implements WebApplicationInitializer {

    public void onStartup(ServletContext container) throws ServletException {

        AnnotationConfigWebApplicationContext ctx = new AnnotationConfigWebApplicationContext();
        ctx.register(AppConfig.class);
        ctx.setServletContext(container);

        ServletRegistration.Dynamic servlet = container.addServlet(
                "dispatcher", new DispatcherServlet(ctx));

        servlet.setLoadOnStartup(1);
        servlet.addMapping("/");
    }

}
```

8.5.8 Application Controller

In this section, we will add Controller which will server GET and POST requests. This is a Spring based controller with annotation based handling of requests .

AppController.java

```
package com.javacodegeeks.controller;

import java.util.List;
import java.util.Locale;

import javax.validation.Valid;
```

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.MessageSource;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.stereotype.Controller;
import org.springframework.ui.ModelMap;
import org.springframework.validation.BindingResult;
import org.springframework.validation.FieldError;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;

import com.javacodegeeks.model.Employee;
import com.javacodegeeks.service.EmployeeService;


@Controller
@RequestMapping("/")
@ComponentScan("com.javacodegeeks")
public class AppController {

    @Autowired
    EmployeeService service;

    @Autowired
    MessageSource messageSource;

     // This method will list all existing employees.
    @RequestMapping(value = { "/", "/list" }, method = RequestMethod.GET)
    public String listEmployees(ModelMap model) {

        List employees = service.findAllEmployees();
        model.addAttribute("employees", employees);
        return "allemployees";
    }

    // This method will provide the medium to add a new employee.
    @RequestMapping(value = { "/new" }, method = RequestMethod.GET)
    public String newEmployee(ModelMap model) {
        Employee employee = new Employee();
        model.addAttribute("employee", employee);
        model.addAttribute("edit", false);
        return "registration";
    }

     // This method will be called on form submission, handling POST request for
     // saving employee in database. It also validates the user input
    @RequestMapping(value = { "/new" }, method = RequestMethod.POST)
    public String saveEmployee(@Valid Employee employee, BindingResult result,
            ModelMap model) {

        if (result.hasErrors()) {
            return "registration";
        }

         // Preferred way to achieve uniqueness of field [ssn] should be implementing  ↩
             custom @Unique annotation
         // and applying it on field [ssn] of Model class [Employee].Below mentioned peace  ↩
             of code [if block] is
          // to demonstrate that you can fill custom errors outside the validation
         // framework as well while still using internationalized messages.
```

```
        if(!service.isEmployeeSsnUnique(employee.getId(), employee.getSsn())){
            FieldError ssnError =new FieldError("employee","ssn",messageSource.getMessage(" ↩
                non.unique.ssn", new String[]{employee.getSsn()}, Locale.getDefault()));
            result.addError(ssnError);
            return "registration";
        }

        service.saveEmployee(employee);

        model.addAttribute("success", "Employee " + employee.getName() + " registered ↩
            successfully");
        return "success";
    }

     // This method will provide the medium to update an existing employee.
    @RequestMapping(value = { "/edit-{ssn}-employee" }, method = RequestMethod.GET)
    public String editEmployee(@PathVariable String ssn, ModelMap model) {
        Employee employee = service.findEmployeeBySsn(ssn);
        model.addAttribute("employee", employee);
        model.addAttribute("edit", true);
        return "registration";
    }


     // This method will be called on form submission, handling POST request for
     // updating employee in database. It also validates the user input

    @RequestMapping(value = { "/edit-{ssn}-employee" }, method = RequestMethod.POST)
    public String updateEmployee(@Valid Employee employee, BindingResult result,
            ModelMap model, @PathVariable String ssn) {

        if (result.hasErrors()) {
            return "registration";
        }

        if(!service.isEmployeeSsnUnique(employee.getId(), employee.getSsn())){
            FieldError ssnError =new FieldError("employee","ssn",messageSource.getMessage(" ↩
                non.unique.ssn", new String[]{employee.getSsn()}, Locale.getDefault()));
            result.addError(ssnError);
            return "registration";
        }

        service.updateEmployee(employee);

        model.addAttribute("success", "Employee " + employee.getName()  + " updated ↩
            successfully");
        return "success";
    }

     // This method will delete an employee by it's SSN value.
    @RequestMapping(value = { "/delete-{ssn}-employee" }, method = RequestMethod.GET)
    public String deleteEmployee(@PathVariable String ssn) {
        service.deleteEmployeeBySsn(ssn);
        return "redirect:/list";
    }

}
```

8.5.9 Views

We will write our Views of MVC in java server pages (jsp). We will need a registration page, showing all employees page and a page to confirm our action to add an employee to our database. Under src → java → webapp → WEB-INF, create a folder views. Inside `views`, we will write our views.

registration.jsp

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<%@ taglib prefix="form" uri="https://www.springframework.org/tags/form"%>
<%@ taglib prefix="c" uri="https://java.sun.com/jsp/jstl/core" %>

<html>

<head>
    <meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
    <title>Employee Registration Form</title>

<style>

    .error {
        color: #ff0000;
    }
</style>

</head>

<body>

    <h2>Registration Form</h2>

    <form:form method="POST" modelAttribute="employee">
        <form:input type="hidden" path="id" id="id"/>

            |<label for="name">Name: </label> |<form:input path="name" id="name"/>|< ↩
                form:errors path="name" cssClass="error"/>

            |<label for="joiningDate">Joining Date: </label> |<form:input path="joiningDate ↩
                " id="joiningDate"/>|<form:errors path="joiningDate" cssClass="error"/>

            |<label for="salary">Salary: </label> |<form:input path="salary" id="salary"/>| ↩
                <form:errors path="salary" cssClass="error"/>

            |<label for="ssn">SSN: </label> |<form:input path="ssn" id="ssn"/>|<form:errors ↩
                 path="ssn" cssClass="error"/>

            |
                    <c:choose>
                        <c:when test="${edit}">
                            <input type="submit" value="Update"/>
                        </c:when>
                        <c:otherwise>
                            <input type="submit" value="Register"/>
                        </c:otherwise>
                    </c:choose>


    </form:form>
    <br/>
    <br/>
    Go back to <a href="<c:url value='/list' />">List of All Employees</a>
```

```
</body>
</html>
```

To show all employees, we will need `allemployees.jsp`.

allemployees.jsp

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<%@ taglib prefix="c" uri="https://java.sun.com/jsp/jstl/core" %>
<html>
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
    <title>University Enrollments</title>

    <style>
        tr:first-child{
            font-weight: bold;
            background-color: #C6C9C4;
        }
    </style>

</head>


<body>
    <h2>List of Employees</h2>

        |NAME|Joining Date|Salary|SSN|
        <c:forEach items="${employees}" var="employee">
            |${employee.name}|${employee.joiningDate}|${employee.salary}|<a href="<c:url  ↩
                value='/edit-${employee.ssn}-employee' />">${employee.ssn}</a>|<a href="< ↩
                c:url value='/delete-${employee.ssn}-employee' />">delete</a>
        </c:forEach>

    <br/>
    <a href="<c:url value='/new' />">Add New Employee</a>
</body>
</html>
```

For a confirmation page, we will create a `success.jsp`

sucess.jsp

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<%@ taglib prefix="c" uri="https://java.sun.com/jsp/jstl/core" %>

<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Registration Confirmation Page</title>
</head>
<body>
    message : ${success}
    <br/>
    <br/>
    Go back to <a href="<c:url value='/list' />">List of All Employees</a>

</body>

</html>
```

8.5.10 Deploy and running the app

Once we are done with coding, right click on the project in eclipse to maven clean and then maven install. If you face the error saying `can not find symbol:method addServlet`, ignore the error and export the project as a WAR file. Now copy this WAR file in your `$TOMCAT_HOME/webapps` directory. Go to commandline in `$TOMCAT_HOME/bin` directory and run `start.bat`. This will start our webserver. Now we can go to browser and access our web application. You might run into the below error

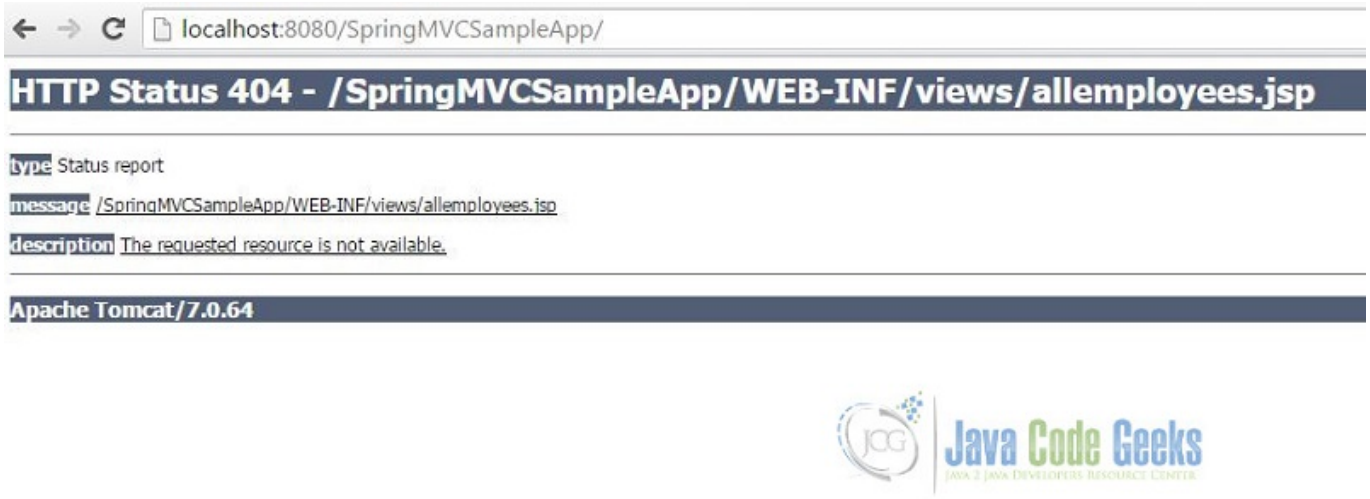

Figure 8.3: Error while running SpringMVCSampleApp

To correct that error, go into `$TOMCAT_HOME/webapps/SpringMVCSampleApp/WEB-INF/classes/WEB-INF` directory and copy `views` folder and paste into `$TOMCAT_HOME/webapps/SpringMVCSampleApp/WEB-INF` directory. Basically, we have mis-configured our views on classpath.

Now access the webapplication `https://localhost:8080/SpringMVCSampleApp/` in browser, and you will see below output

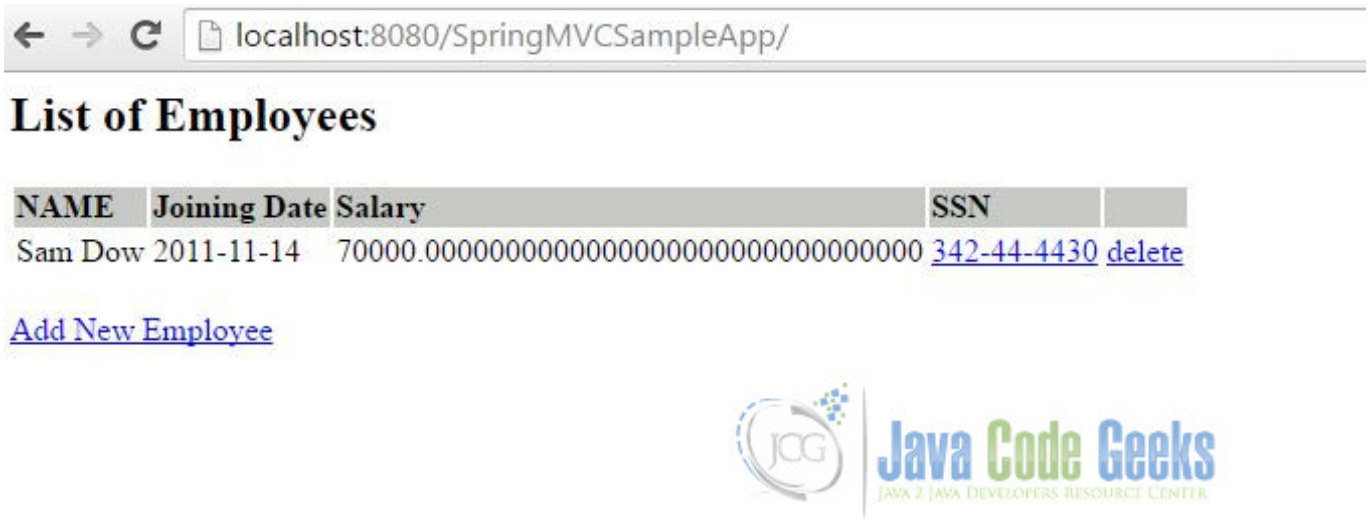

Figure 8.4: Listing of all employees

Figure 8.5: Add new employee

8.6 Download

In this tutorial, we showed how to create a simple CRUD MVC web application using Spring and Hibernate technologies.

Download

You can download the full source code of this example here: **SpringMVCSampleApp**

8.7 Related Articles

Following articles were referred to prepare this tutorial.

- Spring MVC Hibernate integration
- Spring Introduction
- Hibernate Tutorial

Chapter 9

Spring rest template example

Continuing on our Spring Tutorials, we will try to demonstrate the use of `RestTemplate` class available in Spring Framework.

The `RestTemplate` class is the central class in Spring Framework for the synchronous calls by the client to access a REST web-service. This class provides the functionality for consuming the REST Services in a easy and graceful manner. When using the said class the user has to only provide the URL, the parameters(if any) and extract the results received. The `RestTemplate` manages the HTTP connections.

The `RestTemplate` inherits from the `Restoperations` interface and as such, it provides support for consumption of REST web-service for all the major HTTP methods namely GET, POST, PUT, DELETE, OPTIONS and HEAD.

Let's write a sample bean that the `RestTemplate` class will cast the incoming REST response to.

UserBean.java

```
package com.jcg.example.bean;

import com.fasterxml.jackson.annotation.JsonIgnoreProperties;

@JsonIgnoreProperties(ignoreUnknown = true)
public class UserBean
{

                private String userId;

                private String id;

                private String body;

                private String title;



                public String getTitle()
                {
                                return this.title;
                }

                public void setTitle(String title)
                {
                                this.title = title;
                }

                public String getUserId()
                {
                                return this.userId;
                }
```

```
                public void setUserId(String userid)
                {
                                this.userId = userid;
                }

                public String getId()
                {
                                return this.id;
                }

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

                public String getBody()
                {
                                return this.body;
                }

                public void setBody(String body)
                {
                                this.body = body;
                }

                @Override
    public String toString()
    {
                    return "UserBean [userId=" + this.userId + ", id=" + this.id + ", body= ↩
                        " + this.body + ", title=" + this.title + "]";
    }


}
```

`@JsonIgnoreProperties(ignoreUnknown =true)` is used to inform the `RestTemplate` to ignore the properties in the JSON response that are not present in the class it is trying to map to, UserBean in this case.

We shall be calling a sample REST service that returns the following JSON response :

sample.json

```
{
  "userId": 1,
  "id": 1,
  "header": This will be ignored
  "title": "this is a sample title",
  "body": "Sample message in the body"

}
```

As we can see the JSON response received from the has more parameters than our class has. So the RestTemplate will ignore the rest of the properties that are not present in the PoJo we defined above. Please note that as with other the RestTemplate class assumes that we follow the Bean convention of getters and setters for all the properties in the PoJo, otherwise it throws UnknownProperty Exceptions.

Here' s the code that makes an actual call to the Service and maps the response to the PoJo.

RestTemplateExample.java

```
package com.jcg.example;
```

```
import java.util.ArrayList;
import java.util.List;

import org.springframework.http.converter.HttpMessageConverter;
import org.springframework.http.converter.json.MappingJacksonHttpMessageConverter;
import org.springframework.web.client.RestTemplate;

import com.jcg.example.bean.UserBean;

public class RestTemplateExample
{
        public static void main(String[] args)
        {
                RestTemplate restTemplate = new RestTemplate();
                String url = "https://localhost:8080/SpringMVCloginExample/jsp/json.jsp";
                List<HttpMessageConverter> messageConverters =
                                        new ArrayList<HttpMessageConverter>();
                MappingJacksonHttpMessageConverter map =
                                        new MappingJacksonHttpMessageConverter();
                messageConverters.add(map);
                restTemplate.setMessageConverters(messageConverters);
                UserBean bean = restTemplate.getForObject(url, UserBean.class);
                System.out.println("The object received from REST call : "+bean);
        }
}
```

The Output:

```
log4j:WARN No appenders could be found for logger (org.springframework.web.client. ↩
    RestTemplate).
log4j:WARN Please initialize the log4j system properly.
The object received from REST call : UserBean [userId=1, id=1, title=this is a sample title ↩
    ,
body=Sample message in the body]
```

The user can see that the `header` property in the JSON response has been ignored completely and the `UserBean` has been constructed by the RestTemplate as expected. This frees up the application developer from opening a HTTP URL, managing the connection exceptions, Closing the connection tunnel etc.

The user needs to set proper `HTTPMessageConvertors` in the RestTemplate for proper conversion of the messages to the Bean. Here we are using the `MappingJacksonHttpMessageConverter` from the Jackson Library.

In a development environment, the user is encouraged to put this `RestTemplate` instantiation logic alongwith other beans in the XML file. This will help build a loosely coupled application.

Here's the sample way a user can achieve this:

beanConfiguration.xml

```
<bean id="restTemplate" class="org.springframework.web.client.RestTemplate">
        <property name="messageConverters">
                <list>
                        <bean class="org.springframework.http.converter.json. ↩
                            MappingJacksonHttpMessageConverter"/>
                 </list>
        </property>
</bean>
```

The Spring Bean Factory now takes care of instantiation of the Class and injection into the application.

In this example, we demonstrated the consumption of REST Services using only HTTP.GET method. The ,`RestTemplate` however supports all the HTTP methods. Also, the user can pass parameters to the Service using the overloaded versions like `get`

`ForObject(String url, Object request, Class responseType, Object...uriVariables)` `postF orObject(String url, Object request, Class responseType, Object...uriVariables)`

The RestTemplate also supports other custom HTTP methods provided the underlying HTTP library supports the operations.

9.1 Download the Source Code

Here we studied how we can use Spring Framework's `RestTemplate` class to leverage our application and consume the REST service in an effective way.

Download

You can download the source code of this example here: **RestTemplateExample.zip**

Chapter 10

Spring data tutorial for beginners

In this example, we shall demonstrate how to configure Spring Framework to communicate with database using JPA and Hibernate as the JPA vendor.

The benefits of using Spring Data is that it removes a lot of boiler-plate code and provides a cleaner and more readable implementation of `DAO` layer. Also, it helps make the code loosely coupled and as such switching between different JPA vendors is a matter of configuration.

So let's set-up the database for the example. We shall use the MySQL database for this demo.

We create a table "employee" with 2 columns as shown:

```
CREATE TABLE `employee` (
`employee_id`  bigint(20) NOT NULL AUTO_INCREMENT,
`employee_name`  varchar(40) ,
PRIMARY KEY (`employee_id`)
)
```

Now that the table is ready, let's have a look at the libraries we will require for this demo :

- antlr-2.7.7
- aopalliance-1.0
- commons-collections-3.2
- commons-logging-1.1
- dom4j-1.6.1
- hibernate-commons-annotations-4.0.2.Final
- hibernate-core-4.2.6.Final
- hibernate-entitymanager-4.2.6.Final
- hibernate-jpa-2.0-api-1.0.1.Final
- javaee-api-5.0-2
- javassist-3.15.0-GA
- jboss-logging-3.1.0.GA
- jta
- log4j-1.2.14

- mysql-connector-java-5.1.11-bin
- slf4j-api-1.5.6
- slf4j-log4j12-1.5.6
- spring-aop-3.2.4.RELEASE
- spring-beans-3.2.4.RELEASE
- spring-context-3.2.4.RELEASE
- spring-context-support-3.2.4.RELEASE
- spring-core-3.2.4.RELEASE
- spring-expression-3.2.4.RELEASE
- spring-jdbc-3.2.4.RELEASE
- spring-orm-3.2.4.RELEASE
- spring-tx-3.2.4.RELEASE

And here's the project structure :

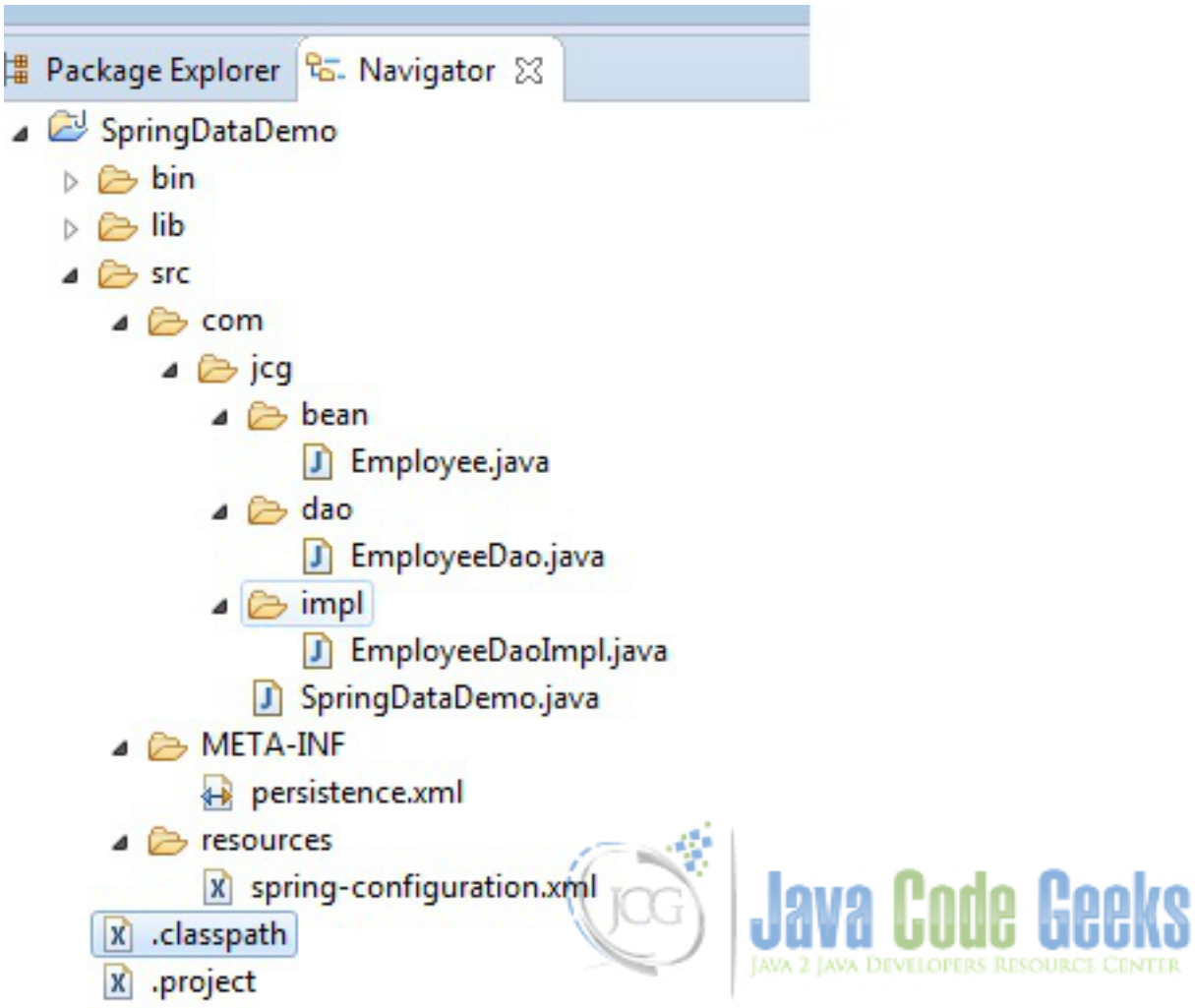

Figure 10.1: Project Structure

Now that the project is all set, we will start writing the code.

First of all, we create the `Employee` class with `employeeId` and `employeeName`. The Person class will be the entity that we will store and retrieve from the database using the JPA.

The `@Entity` marks the class as the JPA Entity. We map the properties of the Employee class with the columns of the employee table and the entity with employee table itself using the `@Table` annotation.

The `toString` method is over-ridden so that we get a meaningful output when we print the instance of the class.

Employee.java

```
package com.jcg.bean;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Table;

@Entity
@Table(name="employee")
public class Employee
{
	@Id
	@GeneratedValue(strategy=GenerationType.AUTO)
	@Column(name = "employee_id")
	private long employeeId;

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

	public Employee()
	{
	}

	public Employee(String employeeName)
	{
		this.employeeName = employeeName;
	}

	public long getEmployeeId()
	{
		return this.employeeId;
	}

	public void setEmployeeId(long employeeId)
	{
		this.employeeId = employeeId;
	}

	public String getEmployeeName()
	{
		return this.employeeName;
	}

	public void setEmployeeName(String employeeName)
	{
		this.employeeName = employeeName;
	}


	@Override
```

```
        public String toString()
        {
                return "Employee [employeeId=" + this.employeeId + ", employeeName=" + this ↩
                    .employeeName + "]";
        }
}
```

Once the Entity is ready, we define the interface for the storage and retrieval of the entity i.e. we shall create a Data Access Interface.

EmployeeDao.java

```
package com.jcg.dao;

import java.sql.SQLException;

import com.jcg.bean.Employee;

public interface EmployeeDao
{
        void save(Employee employee) throws SQLException;

        Employee findByPrimaryKey(long id) throws SQLException;
}
```

We will then, attempt to implement the Data Access Interface and create the actual Data Access Object which will modify the `Person Entity`.

EmployeeDaoImpl.java

```
package com.jcg.impl;


import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;

import org.springframework.stereotype.Repository;
import org.springframework.transaction.annotation.Propagation;
import org.springframework.transaction.annotation.Transactional;

import com.jcg.bean.Employee;
import com.jcg.dao.EmployeeDao;

@Repository("EmployeeDaoImpl")
@Transactional(propagation = Propagation.REQUIRED)
public class EmployeeDaoImpl implements EmployeeDao
{
        @PersistenceContext
        private EntityManager entityManager;


        @Override
        public void save(Employee employee)
        {
                        entityManager.persist(employee);
        }

        @Override
        public Employee findByPrimaryKey(long id)
        {
                        Employee employee = entityManager.find(Employee.class, id);

                        return employee;
```

```
        }

        /**
         * @return the entityManager
         */
        public EntityManager getEntityManager()
        {
                    return entityManager;
        }

        /**
         * @param entityManager the entityManager to set
         */
        public void setEntityManager(EntityManager entityManager)
        {
                    this.entityManager = entityManager;
        }
}
```

The DAO Implementation class is annotated with `@Repository` which marks is as a Repository Bean and prompts the Spring Bean Factory to load the Bean.

`@Transactional` asks the container to provide a transaction to use the methods of this class. `Propagation.REQUIRED` denotes that the same transaction is used if one is available when multiple methods which require transaction are nested. The container creates a single Physical Transaction in the Database and multiple Logical transactions for each nested method. However, if a method fails to successfully complete a transaction, then the entire physical transaction is rolled back. One of the other options is `Propagation.REQUIRES_NEW`, wherein a new physical transaction is created for each method. There are other options which help in having a fine control over the transaction management.

The `@PersistenceContext` annotation tells the container to inject an instance of `entitymanager` in the DAO. The class implements the `save` and `findbyPk` methods which save and fetch the data using the instance of `EntityManager` injected.

Now we define our persistence Unit in the `Persistence.xml` which is put in the `META-INF` folder under `src`. We then mention the class whose instances are to be used Persisted. For this example, it is the `Employee Entity` we created earlier.

persistence.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="https://java.sun.com/xml/ns/persistence"
version="1.0">
<persistence-unit name="jcgPersistence" transaction-type="RESOURCE_LOCAL" >
<class>com.jcg.bean.Employee</class>
</persistence-unit>
</persistence>
```

Now we configure the Spring Container using the `spring-configuration.xml` file.

spring-configuration.xml

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

```
        <context:component-scan base-package="com.jcg" />

        <bean id="dataSource"
                class="org.springframework.jdbc.datasource.DriverManagerDataSource">
                <property name="driverClassName" value="com.mysql.jdbc.Driver" />
                <property name="url" value="jdbc:mysql://localhost:3306/jcg" />
                <property name="username" value="root" />
                <property name="password" value="toor" />
        </bean>

        <bean id="jpaDialect" class="org.springframework.orm.jpa.vendor.HibernateJpaDialect ↩
            " />

        <bean id="entityManagerFactory"
                class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
                <property name="persistenceUnitName" value="jcgPersistence" />
                <property name="dataSource" ref="dataSource" />
                <property name="persistenceXmlLocation" value="META-INF/persistence.xml" />
                <property name="jpaVendorAdapter" ref="jpaVendorAdapter" />
                <property name="jpaDialect" ref="jpaDialect" />
                <property name="jpaProperties">
         <props>
            <prop key="hibernate.hbm2ddl.auto">validate</prop>
            <prop key="hibernate.dialect">org.hibernate.dialect.MySQL5Dialect</prop>
         </props>
      </property>
        </bean>

        <bean id="jpaVendorAdapter"
                class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter">
        </bean>

        <bean id="txManager" class="org.springframework.orm.jpa.JpaTransactionManager">
                <property name="entityManagerFactory" ref="entityManagerFactory" />
                <property name="dataSource" ref="dataSource" />
                <property name="jpaDialect" ref="jpaDialect" />
        </bean>

        <tx:annotation-driven transaction-manager="txManager" />

</beans>
```

We define the beans we need in the `spring-configuration.xml`. The datasource contains the basic configuration properties like URL, user-name, password and the JDBC Driver class-name.

We create a `EntityManagerFactory` using the `LocalContainerEntityManagerFactoryBean`. The properties are provided like the datasource, persistenceUnitName, persistenceUnitLocation, dialect etc. The instance of `EntityManager` gets injected form this `FactoryBean` into the `EmployeeDaoImpl` instance.

Line 51 in the above XML asks the Spring container to manage Transactions using the Spring Container. The TransactionManagerProvider Class is the `JpaTransactionManager` Class.

Now that we have completed all the hard-work, its time to test the configuration:

The `SpringDataDemo` class extracts the `EmployeeDaoImpl` and attempts to save an instance of `Employee` to the employee table and retrieve the same instance from the database.

SpringDataDemo.java

```
package com.jcg;
```

```
import java.sql.SQLException;

import org.springframework.beans.BeansException;
import org.springframework.context.ApplicationContext;
import org.springframework.context.ConfigurableApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

import com.jcg.bean.Employee;
import com.jcg.dao.EmployeeDao;


public class SpringDataDemo
{
        public static void main(String[] args)
        {
                try
                {
                        ApplicationContext context = new ClassPathXmlApplicationContext(" ↩
                            resources\\spring-configuration.xml");

                        //Fetch the DAO from Spring Bean Factory
                        EmployeeDao employeeDao = (EmployeeDao)context.getBean(" ↩
                            EmployeeDaoImpl");
                        Employee employee = new Employee("Employee123");
                        //employee.setEmployeeId("1");

                        //Save an employee Object using the configured Data source
                        employeeDao.save(employee);
                        System.out.println("Employee Saved with EmployeeId "+employee. ↩
                            getEmployeeId());

                        //find an object using Primary Key
                        Employee emp = employeeDao.findByPrimaryKey(employee.getEmployeeId ↩
                            ());
                        System.out.println(emp);

                        //Close the ApplicationContext
                        ((ConfigurableApplicationContext)context).close();
                }
                catch (BeansException | SQLException e)
                {
                        e.printStackTrace();
                }
        }
}
```

10.1 Output:

```
log4j:WARN No appenders could be found for logger (org.springframework.core.env. ↩
    StandardEnvironment).
log4j:WARN Please initialize the log4j system properly.
Employee Saved with EmployeeId 8
Employee [employeeId=8, employeeName=Employee123]
```

As you can see the employee gets saved and we are able to retrieve the employee Object we saved.

10.2 Download the Source Code

Thus we understood how to configure JPA with Spring and what are the benefits of Spring with JPA over vanilla JPA.

Download

You can download the source code of this example here: **SpringDataDemo.zip**

Chapter 11

Spring Batch Tasklet Example

11.1 Introduction

In this article we are going to present an example that demonstrates the working of Spring Batch Tasklet. We will configure a Spring Batch job that reads data from a CSV file into an HSQL database table and then in the Tasklet make a query into the table. As always, the example code is available for download at the end of the article.

But before we begin, a few questions need to be asked and answered. At the outset, what is Spring Batch? Well, it is a light-weight and robust framework for batch processing. And guess what? It is open-source; which is good! Now the question is when would one use batch processing? To answer that, consider a scenario where a large number of operations need to be performed, say process a million database records. And let's say, such processing is a periodic activity happening, say weekly, monthly or daily! Now we want this processing, which could run for hours on end, to run or be scheduled periodically with minimum human intervention. This is when Spring Batch comes to the rescue. And it does its bit in a pretty nice and efficient way as we will see in this example. But before we get our hands dirty, we will take a quick look at a couple of important elements of the Spring Batch Framework. Of course, there are many more elements of interest and importance which could be looked up from the official Spring Batch Documentation. The article is organized as listed below. Feel free to jump to any section of choice.

11.2 Spring Batch Framework: Key Concepts

The following section skims through the key concepts of the framework.

11.2.1 Jobs

The Spring Batch documentation describes it as *an entity that encapsulates the entire batch process*.Think of a Job as an activity, a task; say, processing a million database records. Now performing this one activity involves several smaller activities, like reading the data from the database, processing each record and then writing that record to a file or in a database etc. So a Job basically holds all these logically related bunch of activities that identify a flow or a sequence of actions. A `https://docs.spring.io/spring-batch/apidocs/org/springframework/batch/core/Job.html[Job]` is actually an interface and `https://docs.spring.io/spring-batch/apidocs/org/springframework/batch/core/job/SimpleJob.html[SimpleJob]` is one of its simplest implementations provided by the framework. The batch namespace abstracts away these details and allows one to simply configure a job using the `<job/>` tags as shown below.

```
<job id="processDataJob" job-repository="job-repo" restartable="1">
    <step id="dataload" next="processLoad"/>
    <step id="processLoad"/>
</job>
```

Points to notice about the above job configuration

- It has to have an id/name
- A JobRepository can be specified explicitly as is done above. By default, it takes the job-repository name as `https://docs.spring.io/spring-batch/trunk/reference/html/domain.html#domainJobRepository[jobRepository]`. As the name suggests, it offers the persistence mechanism in the framework.
- The *restartable* property specifies whether the Job once completed could be restarted or not. It is scoped over all the Steps in the Job. It takes a default value of *true*.
- And then a group of Steps has to be configured. Observe how an order of execution of the Steps can be specified using the attribute *next*

11.2.2 Steps

Spring Batch defines Steps as domain objects that identify an independent, sequential phase of the Job. In other words all the details needed to do the actual batch processing are encapsulated in Steps. Hence, each Job can have one or more Steps. Each Step comprises three elements: `ItemReader`, `ItemProcessor` and `ItemWriter` as shown in the diagram below taken from the Spring Batch Documentation.

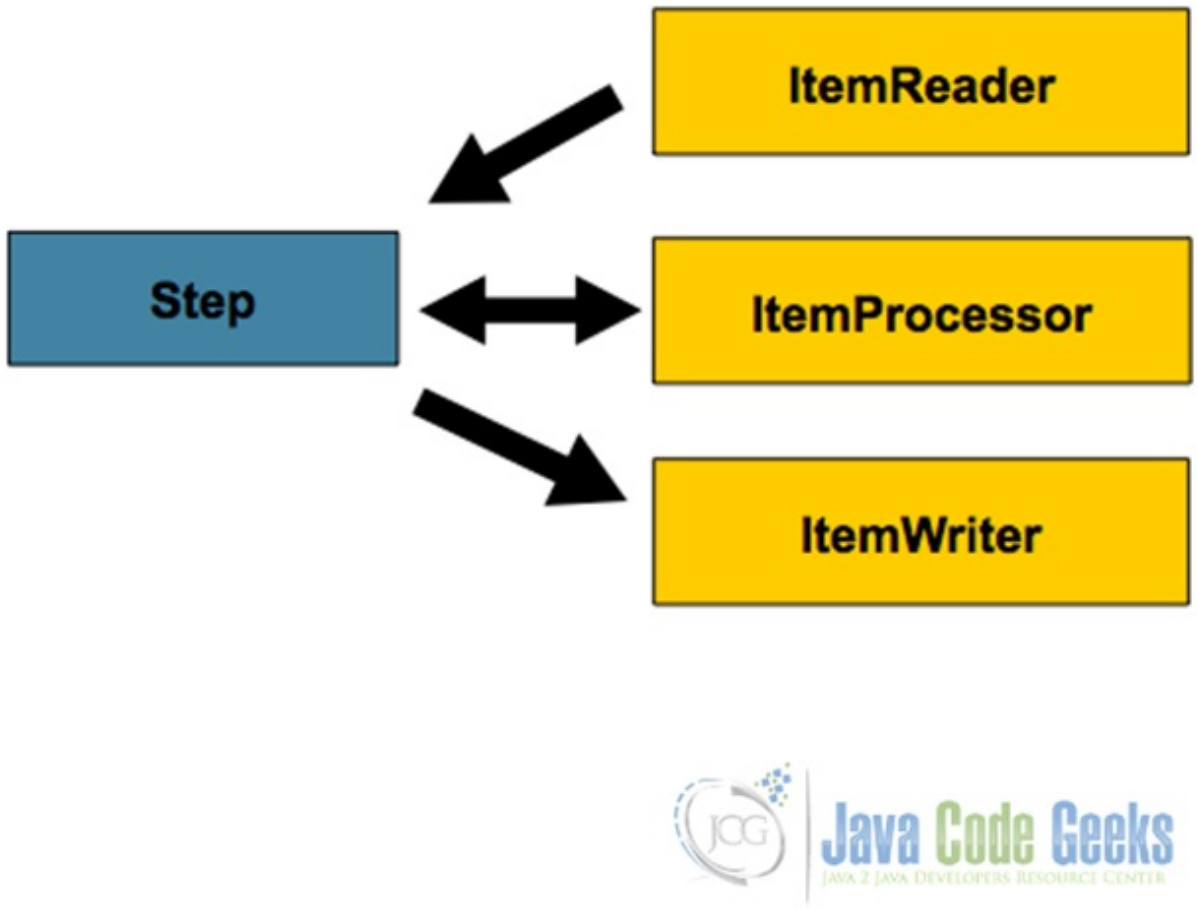

Figure 11.1: Spring Batch Step

11.2.2.1 ItemReader

The https://docs.spring.io/spring-batch/trunk/reference/html/readersAndWriters.html#itemReader[ItemReader] is an abstraction that provides the means by which data is read one item at a time into the Step. It can retrieve the input from different sources and there are different implementations floated by the framework as listed in the https://docs.spring.io/spring-batch/trunk/reference/html/listOfReadersAndWriters.html[appendix]. The input sources are broadly categorized as follows:

- Flat Files: where the data units in each line are separated by tags, spaces or other special characters
- XML Files: the XML File Readers parse, map and validate the data against an XSD schema
- Databases: the readers accessing a database resource return result-sets which can be mapped to objects for processing

11.2.2.2 ItemProcessor

The https://docs.spring.io/spring-batch/trunk/reference/html/readersAndWriters.html#itemProcessor[ItemProcessor] represents the business processing of the data read from the input source. Unlike the ItemReader and ItemWriter, it is an optional attribute in the Step configuration. It is a very simple interface that simply allows passing it an object and transforming it to another with the application of the desired business logic. ItemProcessor Interface

```
public interface ItemProcessor<I,O> {
   O process(I item) throws Exception;
}
```

11.2.2.3 ItemWriter

An https://docs.spring.io/spring-batch/trunk/reference/html/readersAndWriters.html#itemWriter[ItemWriter] is a pretty simple interface which represents the reverse functionality of the ItemReader. It receives a batch or chunk of data that is to be written out either to a file or a database. So a bunch of different ItemWriters are exposed by the framework as listed in this Appendix.

Note that ItemReaders and ItemWriters can also be customized to suit one's specific requirements.

So much for what comprises Steps. Now coming to the processing of Steps; it can happen in two ways: (i) Chunks and (ii) Tasklets.

11.2.2.4 Chunk Processing

Chunk-oriented processing is the most commonly encountered operation style in which the processing happens in certain *chunks* or blocks of data defined by a transaction boundary. That is, the https://docs.spring.io/spring-batch/trunk/reference/html/readersAndWriters.html#itemReader[itemReader] reads a piece of data which are then fed to the https://docs.spring.io/spring-batch/trunk/reference/html/readersAndWriters.html#itemProcessor[itemProcessor] and aggregated till the transaction limit is reached. Once it does, the aggregated data is passed over to the https://docs.spring.io/spring-batch/trunk/reference/html/readersAndWriters.html#itemWriter[itemWriter] to write out the data. The size of the chunk is specified by the *commit-interval* attribute as shown in the snippet below.

Step

```
<step id="springBatchCsvToXmlProcessor">
        <chunk reader="itemReader" writer="xmlWriter" commit-interval="10"></chunk>
</step>
```

The following diagram from the Spring Documentation summarizes the operation pretty well.

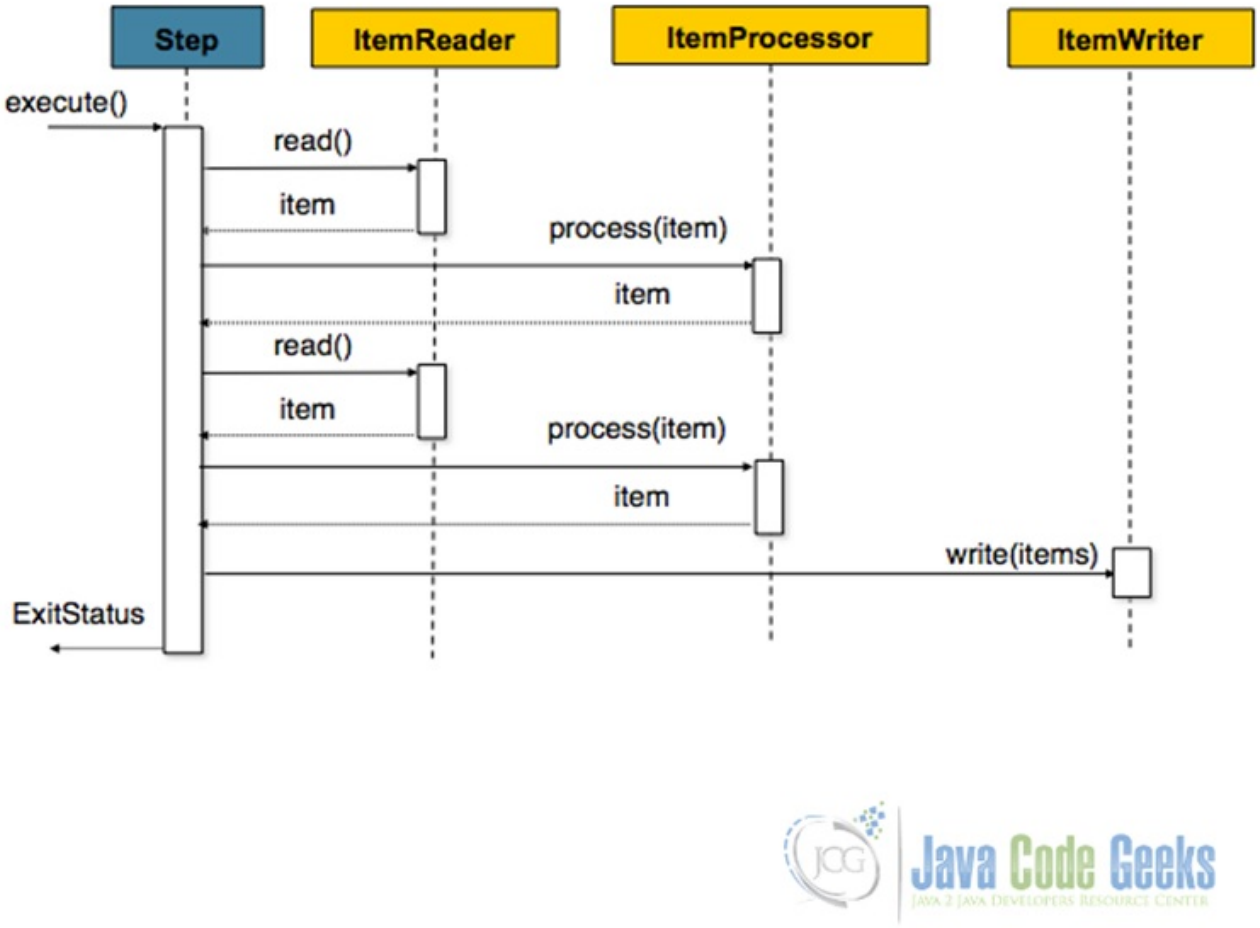

Figure 11.2: Chunk-Oriented Processing

11.2.2.5 TaskletStep Processing

Now consider a scenario which involves just one task, say invoking a Stored Procedure or making a remote call or anything that does not involve an entire sequence of reading and processing and writing data but just one operation. Hence, we have the `https://docs.spring.io/spring-batch/trunk/reference/html/configureStep.html#taskletStep[Tasklet]` which is a simple interface with just one method `execute`. The following code snippet shows how to configure a TaskletStep.

`TaskletStep`

```
<step id="step1">
    <tasklet ref="myTasklet"/>
</step>
```

Points worth a note in the above configuration are as follows:

- The *ref* attribute of the <tasklet/> element must be used that holds a reference to bean defining the Tasklet object
- No <chunk/> element should be used inside the <tasklet/> element
- The TaskletStep repeatedly calls the `execute` method of the implementing class until it either encounters a `RepeatStatus.FINISHED` flag or an exception.

- And each call to a Tasklet is wrapped in a transaction

11.2.3 Tasklet Example

Now that we have had a quick briefing on the concepts of `Jobs`,`Steps`,`Chunk-Processing` and `Tasklet-Processing`; we should be good to start walking through our Tasklet example. We will be using Eclipse IDE and Maven. And we will use the in-memory database HSQL. In this example, we will simply read from a CSV file and write it to an HSQL database table. And once the operation gets done, we will use the Tasklet to make a query into the database table. Simple enough! Let's begin.

11.2.3.1 Tools used

- Maven 2.x
- Eclipse IDE
- JDK 1.6

11.2.3.2 Create a Maven Project

- Fire up Eclipse from a suitable location/folder
- Click on File→ New→Project..
- From the pop-up box choose Maven→Maven Project→Next
- In the next window that comes up, choose the creation of a simple project skipping archetype selection and then click Next.

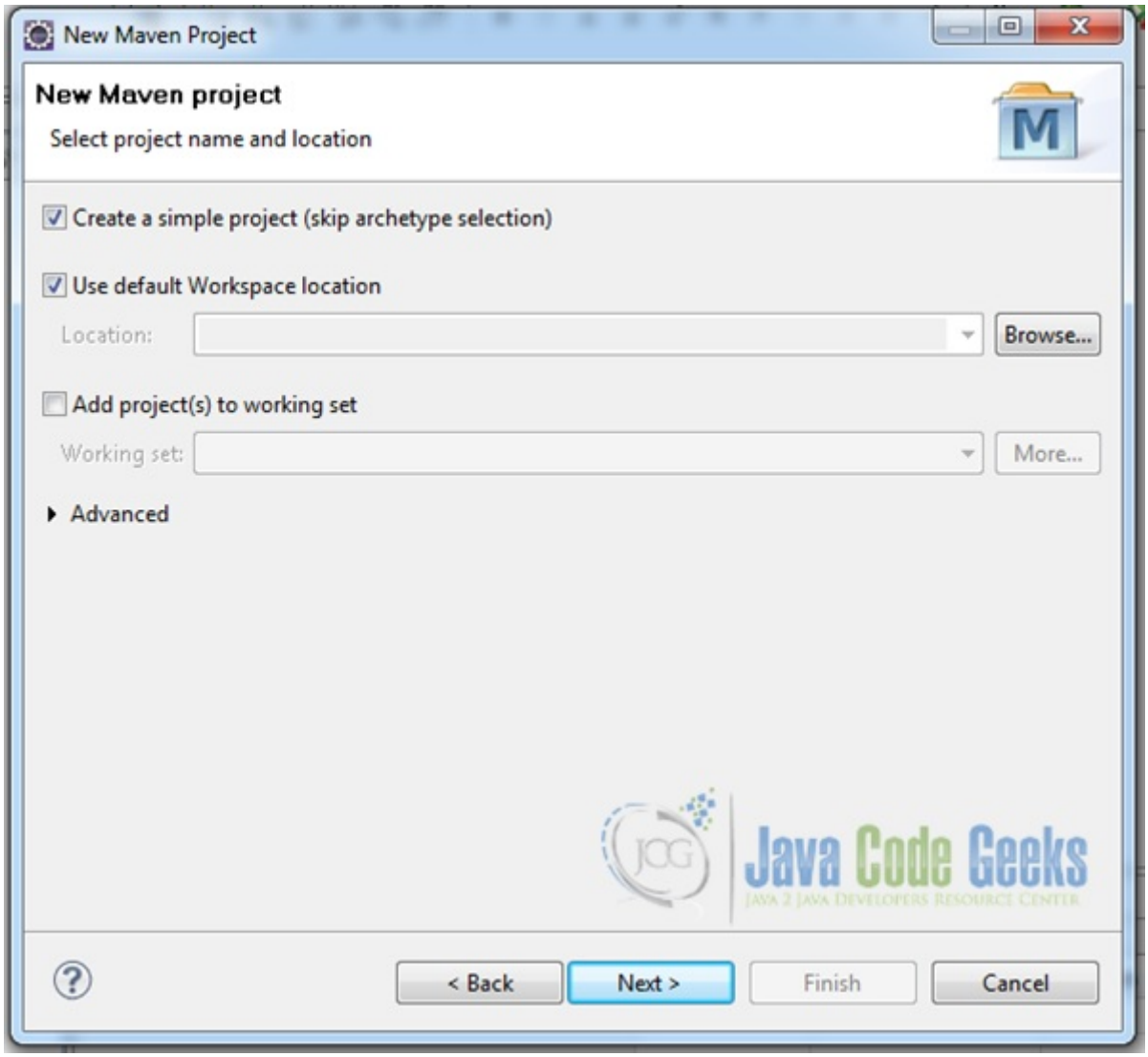

Figure 11.3: Skip ArcheType Selection

- In the next screen, just supply the `groupId` and `artifactId` values as shown in the screenshot below and click on *Finish*

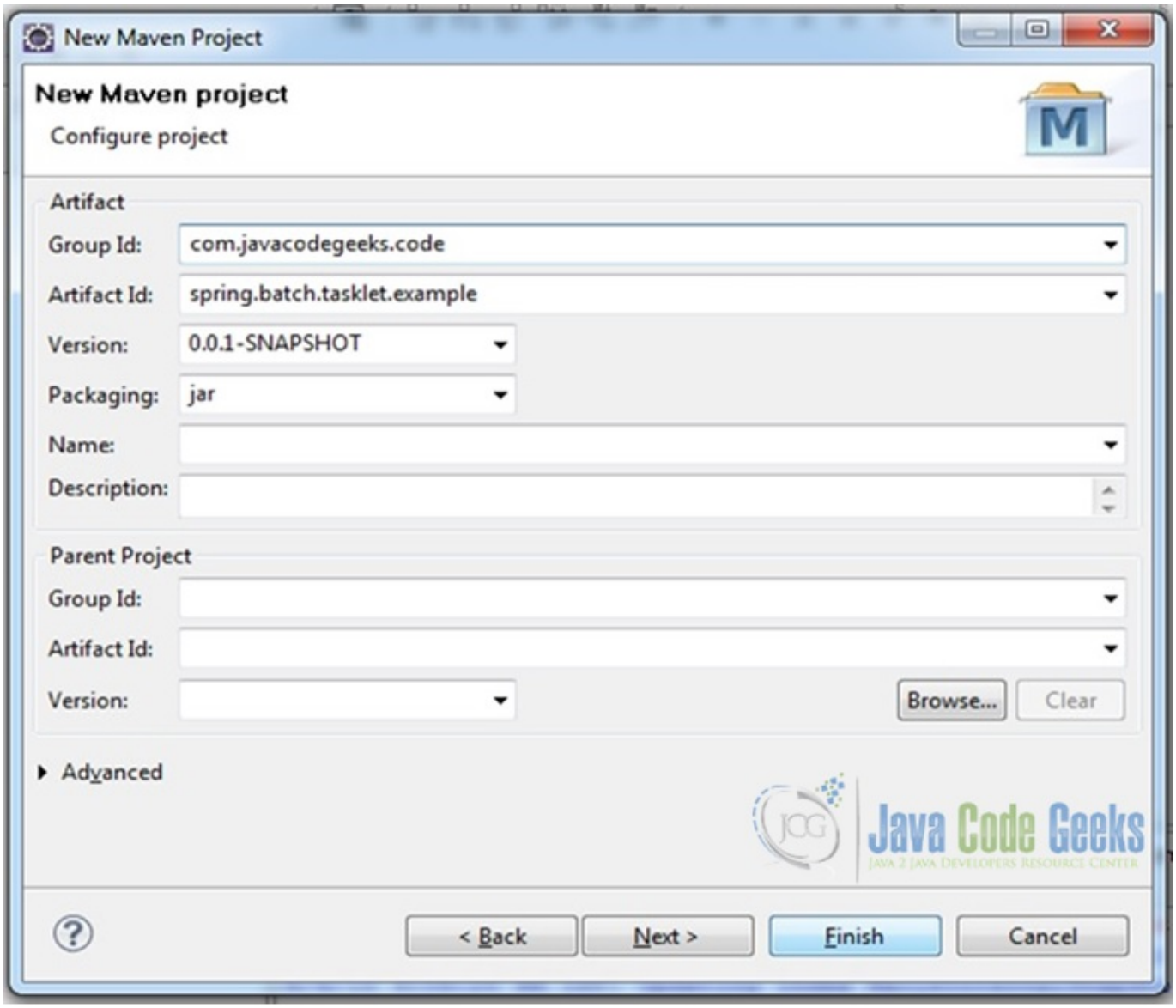

Figure 11.4: Create Maven Project

- This should give the following final project structure

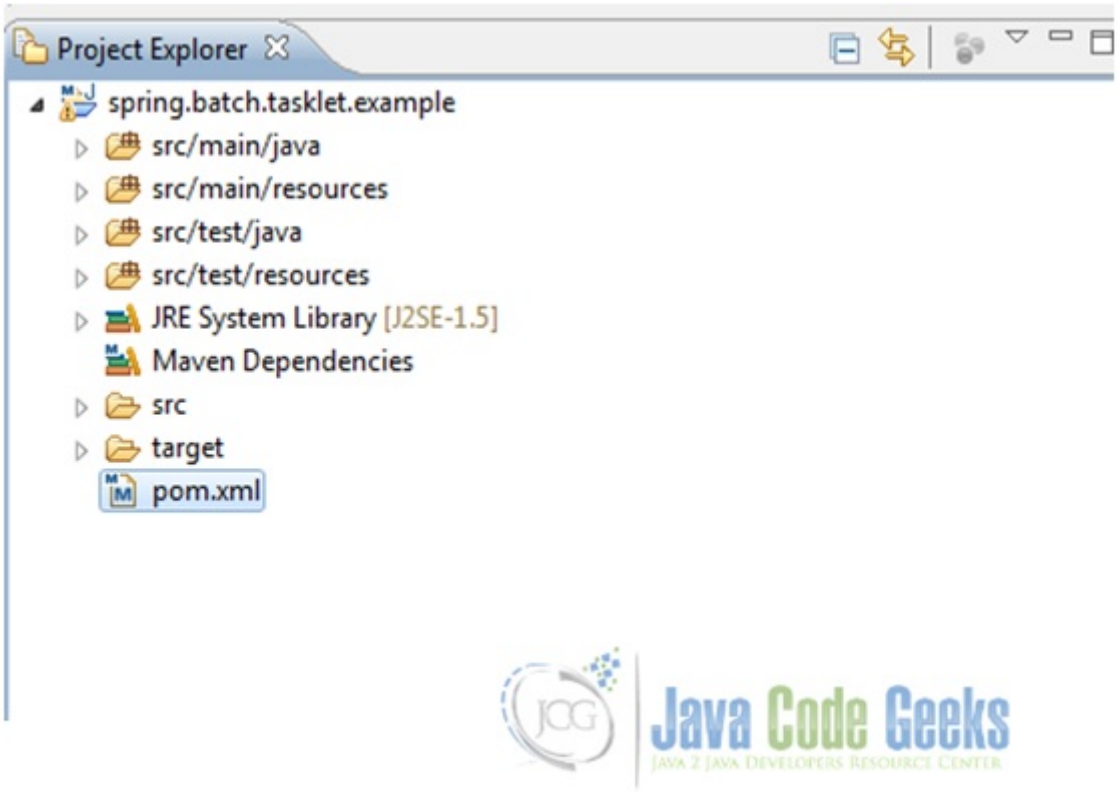

Figure 11.5: Project Structure

- Then after add some more folders and packages so that we have the following project created.

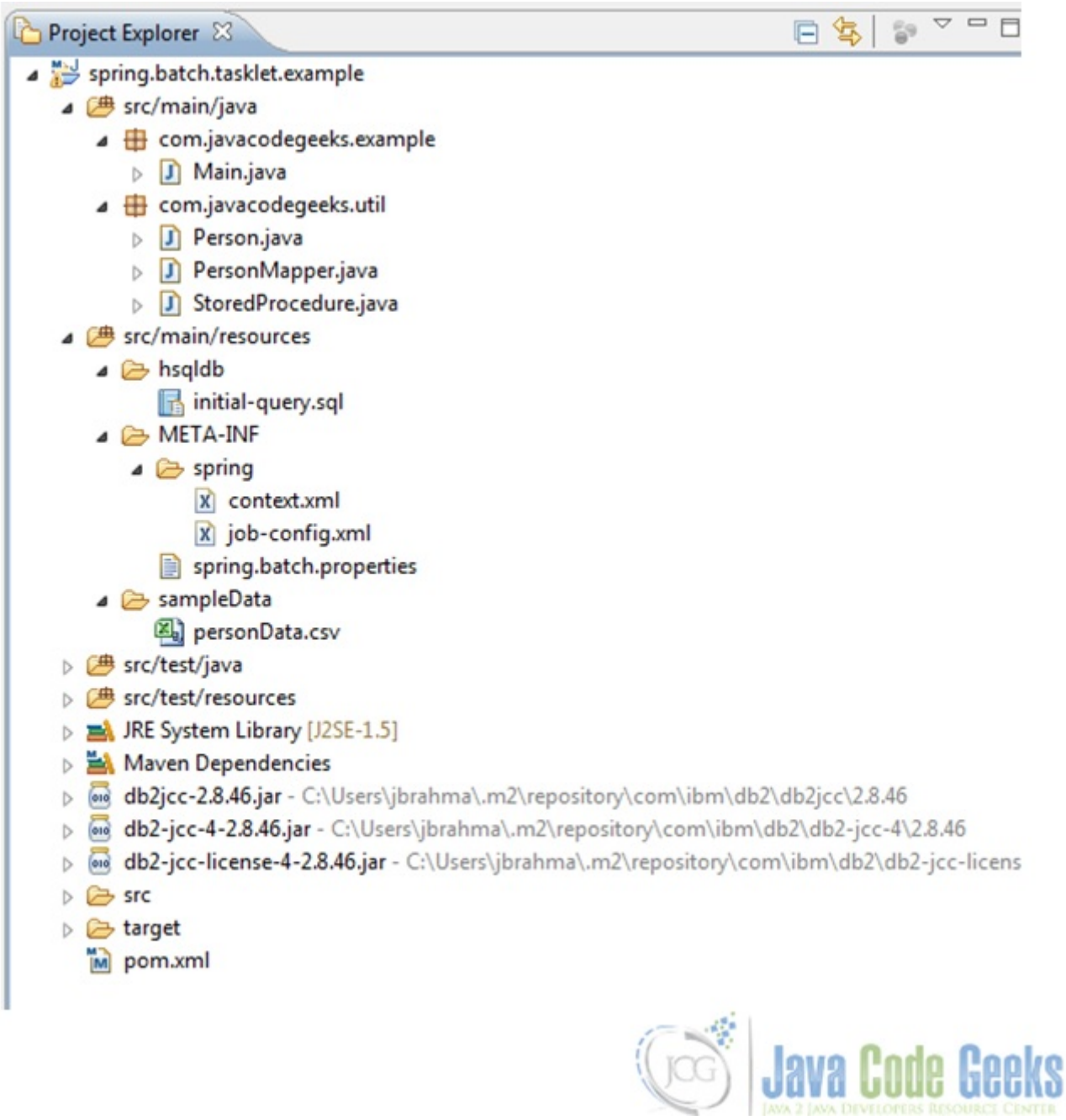

Figure 11.6: Final Project Structure

11.2.3.3 Add Dependencies

In the `pom.xml` file add the following dependencies. Note that Spring-Batch internally imports Spring-core etc. Hence, we are not importing Spring-Core explicitly.

`pom.xml`

```
<project xmlns="https://maven.apache.org/POM/4.0.0" xmlns:xsi="https://www.w3.org/2001/ ↩
    XMLSchema-instance" xsi:schemaLocation="https://maven.apache.org/POM/4.0.0 https://maven ↩
    .apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.javacodegeeks.code</groupId>
```

```
  <artifactId>spring.batch.tasklet.example</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <properties>
        <spring.batch.version>3.0.3.RELEASE</spring.batch.version>
        <spring.jdbc.version>4.0.5.RELEASE</spring.jdbc.version>
        <commons.version>1.4</commons.version>
        <hsql.version>1.8.0.7</hsql.version>
  </properties>
  <dependencies>
        <dependency>
                <groupId>org.springframework.batch</groupId>
                <artifactId>spring-batch-core</artifactId>
                <version>${spring.batch.version}</version>
        </dependency>
        <dependency>
                <groupId>org.springframework</groupId>
                <artifactId>spring-jdbc</artifactId>
                <version>${spring.jdbc.version}</version>
   </dependency>
   <dependency>
                <groupId>commons-dbcp</groupId>
                <artifactId>commons-dbcp</artifactId>
                <version>${commons.version}</version>
        </dependency>
        <dependency>
                <groupId>hsqldb</groupId>
                <artifactId>hsqldb</artifactId>
                <version>${hsql.version}</version>
        </dependency>

  </dependencies>
</project>
```

11.2.3.4 Add db2* jars

The db2-jcc* jars are required to connect to the HSQL database.

Right click on the project→ Java Build Path→Libraries→Add External jars

Choose the jar files and click *OK*. *These jars are available with the example code for download.*

11.2.3.5 HSQL Table Creation

Under `src/main/resources/hsqldb`, add a file `initial-query` with the following table creation query in it `initial-query`

```
DROP TABLE IF EXISTS PERSON_DATA;

CREATE TABLE PERSON_DATA(
firstName VARCHAR(20),
lastName VARCHAR(20),
address VARCHAR(50),
age INT,
empId INT
);
```

11.2.3.6 Supply Sample Data

Under `src/main/resources`, add a `personData.csv` file under the `sampleData` folder with some data. For example,

firstName	lastName	address	age	empId
"Alex",	"Borneo",	"101, Wellington, London",	31,	111390
"Theodora",	"Rousevelt",	"2nd Cross, Virgina, USA",	25,	111909
"Artemisia",	"Brown",	"West Southampton,NJ",	23,	111809
"Cindrella",	"James",	"Middletown, New Jersey,"	28,	111304

Figure 11.7: Browser Output

11.2.3.7 Data Model

Next, create a simple POJO class `Person.java` with attributes as firstName, lastName etc and their `getters` and `setters`
`Person.java`

```
package com.javacodegeeks.util;

public class Person {
   String firstName,lastName,address;
   int age, empId;


        public String getFirstName() {
                return firstName;
        }
        public void setFirstName(String firstName) {
                this.firstName = firstName;
        }

        public String getLastName() {
                return lastName;
        }
        public void setLastName(String lastName) {
                this.lastName = lastName;
        }

        public String getAddress() {
                return address;
        }
        public void setAddress(String address) {
                this.address = address;
        }

        public int getAge() {
                return age;
        }
        public void setAge(int age) {
                this.age = age;
        }

        public int getEmpId() {
                return empId;
        }
        public void setEmpId(int empId) {
                this.empId = empId;
        }

        @Override
```

```
        public String toString(){
                return firstName+" "+ lastName+" "+ address;
        }

}
```

11.2.3.8 RowMapper

Next, we will need a `PersonMapper.java` class that maps the data to the POJO `PersonMapper.java`

```
package com.javacodegeeks.util;

import java.sql.ResultSet;
import java.sql.SQLException;

import org.springframework.jdbc.core.RowMapper;

public class PersonMapper implements RowMapper {
           public Person mapRow(ResultSet rs, int rowNum) throws SQLException {
                      Person person = new Person();
                      person.setFirstName(rs.getString("firstName"));
                      person.setLastName(rs.getString("lastName"));
                      person.setAddress(rs.getString("address"));
                      person.setAge(rs.getInt("age"));
                      person.setEmpId(rs.getInt("empId"));
                      return person;
                  }
               }
```

11.2.3.9 Tasklet

Now we will create a class `StoredProcedure.java` that implements the `Tasklet`. This is what will be executed from our tasklet code. On second thoughts, probably the class should have been named more appropriately. Anyways, so here is the class `StoredProcedure.java`

```
package com.javacodegeeks.util;

import java.util.ArrayList;
import java.util.List;

import javax.sql.DataSource;

import org.springframework.batch.core.StepContribution;
import org.springframework.batch.core.scope.context.ChunkContext;
import org.springframework.batch.core.step.tasklet.Tasklet;
import org.springframework.batch.repeat.RepeatStatus;
import org.springframework.jdbc.core.JdbcTemplate;

public class StoredProcedure implements Tasklet{

        private DataSource dataSource;
        private String sql;


        public DataSource getDataSource() {
                return dataSource;
        }
        public void setDataSource(DataSource dataSource) {
                this.dataSource = dataSource;
```

```
        }
        public String getSql() {
                return sql;
        }
        public void setSql(String sql) {
                this.sql = sql;
        }
        public RepeatStatus execute(StepContribution contribution,
                        ChunkContext chunkContext) throws Exception {

                List result=new ArrayList();
                JdbcTemplate myJDBC=new JdbcTemplate(getDataSource());
        result = myJDBC.query(sql, new PersonMapper());
        System.out.println("Number of records effected: "+ result);
        return RepeatStatus.FINISHED;

        }
}
```

11.2.3.10 Job Configuration

Ok, so now we are nearing our goal. We will configure the job that reads data from a CSV file into a database table and then calls the tasklet in `job-config.xml` as follows. `job-config.xml`

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="https://www.springframework.org/schema/beans"
        xmlns:aop="https://www.springframework.org/schema/aop" xmlns:tx="https://www. ↩
            springframework.org/schema/tx"
        xmlns:batch="https://www.springframework.org/schema/batch" xmlns:task="https://www. ↩
            springframework.org/schema/task"
        xmlns:file="https://www.springframework.org/schema/integration/file"
        xmlns:integration="https://www.springframework.org/schema/integration"
        xmlns:p="https://www.springframework.org/schema/p" xmlns:xsi="https://www.w3.org ↩
            /2001/XMLSchema-instance"
        xmlns:context="https://www.springframework.org/schema/context"
        xmlns:util="https://www.springframework.org/schema/util"
        xsi:schemaLocation="https://www.springframework.org/schema/beans
        https://www.springframework.org/schema/beans/spring-beans.xsd
        https://www.springframework.org/schema/integration
        https://www.springframework.org/schema/integration/spring-integration.xsd
        https://www.springframework.org/schema/integration/file
        https://www.springframework.org/schema/integration/file/spring-integration-file.xsd
    https://www.springframework.org/schema/beans
    https://www.springframework.org/schema/beans/spring-beans.xsd
    https://www.springframework.org/schema/batch
    https://www.springframework.org/schema/batch/spring-batch.xsd
    https://www.springframework.org/schema/context
    https://www.springframework.org/schema/context/spring-context.xsd
    https://www.springframework.org/schema/aop
        https://www.springframework.org/schema/aop/spring-aop.xsd
        https://www.springframework.org/schema/tx
        https://www.springframework.org/schema/tx/spring-tx.xsd
        https://www.springframework.org/schema/util
        https://www.springframework.org/schema/util/spring-util.xsd
        https://www.springframework.org/schema/task
    https://www.springframework.org/schema/task/spring-task.xsd">

    <!-- Pojo class used as data model -->
        <bean id="personModel" class="com.javacodegeeks.util.Person" scope="prototype"/>
```

```
        <!-- Define the job -->
      <job id="springBatchCsvToDbJob" xmlns="https://www.springframework.org/schema/batch ↩
          ">
                      <step id="springBatchCsvToDbProcessor" next="callStoredProcedure">
                              <tasklet >
                                      <chunk reader="itemReader" writer="itemWriter"  ↩
                                          commit-interval="10"></chunk>
                              </tasklet>
                      </step>
                      <step id="callStoredProcedure">
                              <tasklet ref="storedProcedureCall"/>
                      </step>
      </job>

      <bean id="storedProcedureCall" class="com.javacodegeeks.util.StoredProcedure">
      <property name="dataSource" ref="dataSource"/>
      <property name="sql" value="${QUERY}"/>
   </bean>

      <!-- Read data from the csv file-->
      <bean id="itemReader" class="org.springframework.batch.item.file.FlatFileItemReader ↩
          ">
              <property name="resource" value="classpath:sampleData/personData.csv"></ ↩
                  property>
              <property name="linesToSkip" value="1"></property>
              <property name="lineMapper">
                      <bean class="org.springframework.batch.item.file.mapping. ↩
                          DefaultLineMapper">
                          <property name="lineTokenizer">
                                  <bean class="org.springframework.batch.item.file. ↩
                                      transform.DelimitedLineTokenizer">
                                          <property name="names" value="firstName, ↩
                                              lastName,address,age,empId"></property>
                                  </bean>
                          </property>
                          <property name="fieldSetMapper">
                                  <bean class="org.springframework.batch.item.file. ↩
                                      mapping.BeanWrapperFieldSetMapper">
                                          <property name="prototypeBeanName" value=" ↩
                                              personModel"></property>
                                  </bean>
                          </property>
                      </bean>
              </property>
   </bean>

   <bean id="itemWriter" class="org.springframework.batch.item.database.JdbcBatchItemWriter ↩
       ">
      <property name="dataSource" ref="dataSource"></property>
      <property name="sql">
              <value>
                      <![CDATA[
                          insert into PERSON_DATA(firstName,lastName,address,age,empId)
                              values (:firstName,:lastName,:address,:age,:empId)
                      ]]>
              </value>
      </property>
      <property name="itemSqlParameterSourceProvider">
              <bean class="org.springframework.batch.item.database. ↩
                  BeanPropertyItemSqlParameterSourceProvider"/>
      </property>
   </bean>
```

```
</beans>
```

11.2.3.11 Context Configuration

Next, we will set up the `context.xml` file that defines the `jobRepository`, `jobLauncher`, `transactionManager` etc.

- Notice how the HSQL database has been set-up in the `dataSource`
- Also, take note of how the initial queries to be executed on the `dataSource` have been specified
- We have also configured the property-placeholder in it so that the values passed in `spring.batch.properties` file is accessible.
- Also, we have simply imported the `job-config.xml` file in it, so that loading just this one file in the application context is good enough

`context.xml`

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

        <import resource="classpath:META-INF/spring/job-config.xml"/>

           <bean
                class="org.springframework.beans.factory.config. ↩
                    PropertyPlaceholderConfigurer">
                <property name="locations">
                        <list>
                                <value>classpath:META-INF/spring.batch.properties
                                </value>
                        </list>
                </property>
                <property name="searchSystemEnvironment" value="true" />
                <property name="systemPropertiesModeName" value=" ↩
                    SYSTEM_PROPERTIES_MODE_OVERRIDE" />
                <property name="ignoreUnresolvablePlaceholders" value="true" />
        </bean>

        <bean id="jobRepository"
        class="org.springframework.batch.core.repository.support.JobRepositoryFactoryBean">
                <property name="dataSource" ref="dataSource" />
                <property name="transactionManager" ref="transactionManager" />
                <property name="databaseType" value="hsql" />
        </bean>

        <bean id="jobLauncher"
        class="org.springframework.batch.core.launch.support.SimpleJobLauncher">
                <property name="jobRepository" ref="jobRepository" />
        </bean>

        <bean id="transactionManager"
        class="org.springframework.batch.support.transaction.ResourcelessTransactionManager ↩
            " />
```

```
        <bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource"
                lazy-init="true" destroy-method="close">
                <property name="driverClassName" value="org.hsqldb.jdbcDriver" />
                <property name="url"
                        value="jdbc:hsqldb:file:src/test/resources/hsqldb/batchcore.db; ↩
                            shutdown=true;" />
                <property name="username" value="sa" />
                <property name="password" value="" />

        </bean>

  <!-- create job-meta tables automatically -->
  <!-- Note: when using db2 or hsql just substitute "mysql" with "db2" or "hsql".
       For example, .../core/schema-drop-db2.sql -->
          <jdbc:initialize-database data-source="dataSource">
                <jdbc:script location="org/springframework/batch/core/schema-drop-hsqldb. ↩
                    sql" />
            <jdbc:script location="org/springframework/batch/core/schema-hsqldb.sql" />
                <jdbc:script location="classpath:hsqldb/initial-query.sql" />
          </jdbc:initialize-database>

</beans>
```

11.2.3.12 Properties File

Add a properties file `spring.batch.properties` under `src/main/resources/META-INF` and put the query we want to be executed as part of the tasklet as a property value as shown here.

`spring.batch.properties`

```
QUERY=select * from PERSON_DATA where age=31
```

11.2.3.13 Run the Application

Now we are all set to fire the execution. In the `Main.java` file, write down the following snippet and run it as a Java application.

`Main.java`

```
package com.javacodegeeks.example;

import org.springframework.batch.core.Job;
import org.springframework.batch.core.JobExecution;
import org.springframework.batch.core.JobParameters;
import org.springframework.batch.core.launch.JobLauncher;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class Main {

        public static void main(String[] args) {
                ApplicationContext ctx = new ClassPathXmlApplicationContext("classpath:META ↩
                    -INF/spring/context.xml");
                Job job = (Job) ctx.getBean("springBatchCsvToDbJob");
                JobLauncher jobLauncher = (JobLauncher) ctx.getBean("jobLauncher");
                try{
                        JobExecution execution = jobLauncher.run(job, new JobParameters());
                        System.out.println(execution.getStatus());
                }catch(Exception e){
```

```
                        e.printStackTrace();
                }
        }
}
```

11.2.3.14 Output

On running the application, we will find the following output.

```
Jun 8, 2015 9:05:37 AM org.springframework.batch.core.launch.support.SimpleJobLauncher run
INFO: Job: [FlowJob: [name=springBatchCsvToDbJob]] launched with the following parameters: ↩
    [{}]
Jun 8, 2015 9:05:37 AM org.springframework.batch.core.job.SimpleStepHandler handleStep
INFO: Executing step: [springBatchCsvToDbProcessor]
Jun 8, 2015 9:05:37 AM org.springframework.batch.core.job.SimpleStepHandler handleStep
INFO: Executing step: [callStoredProcedure]
Number of records effected: [Alex Borneo 101, Wellington, London]
Jun 8, 2015 9:05:37 AM org.springframework.batch.core.launch.support.SimpleJobLauncher run
INFO: Job: [FlowJob: [name=springBatchCsvToDbJob]] completed with the following parameters: ↩
     [{}] and the following status: [COMPLETED]
COMPLETED
```

11.2.4 Download Example

This brings us to the end of this example; hope it was an interesting and useful read. As promised, the example code is available for download below.

Download

You can download the full source code of this example here : **spring.batch.tasklet.example**

Chapter 12

Spring Boot Tutorial for beginners

12.1 Introduction

When I just heard about Spring Boot there were many questions to pop out of my head "What is it? Why do I need it? How different is it to the other features under the same Spring umbrella?" etc. I am sure you would have had similar inquisitions too.

In short, Spring Boot takes care of application infrastructure while you can focus on coding the actual business flesh. Surely fast tracks building of applications. It makes reasonable assumptions of the dependencies and adds them accordingly. It also lets you customize the dependencies according to your requirement.

In the following sections, I am going to site a sample application example using Spring Boot.

12.2 Environment

This tutorial assumes that you have basic understanding of Gradle build framework and also that your Eclipse IDE (Luna) environment is fully setup and configured with:

- Java 1.8
- Gradle 2.9
- Groovy Eclipse Plugin
- Eclipse Buildship Plugin for Eclipse Gradle integration

In addition to the above you need the following to work on this sample application.

- Spring Boot 1.3.2
- Spring Boot Gradle plugin

This tutorial assumes that you have basic understanding of Spring framework.

12.3 Sample Application using Spring Boot

12.3.1 Create and configure a Gradle project in Eclipse IDE

In the Eclipse IDE, click File → New → Other:

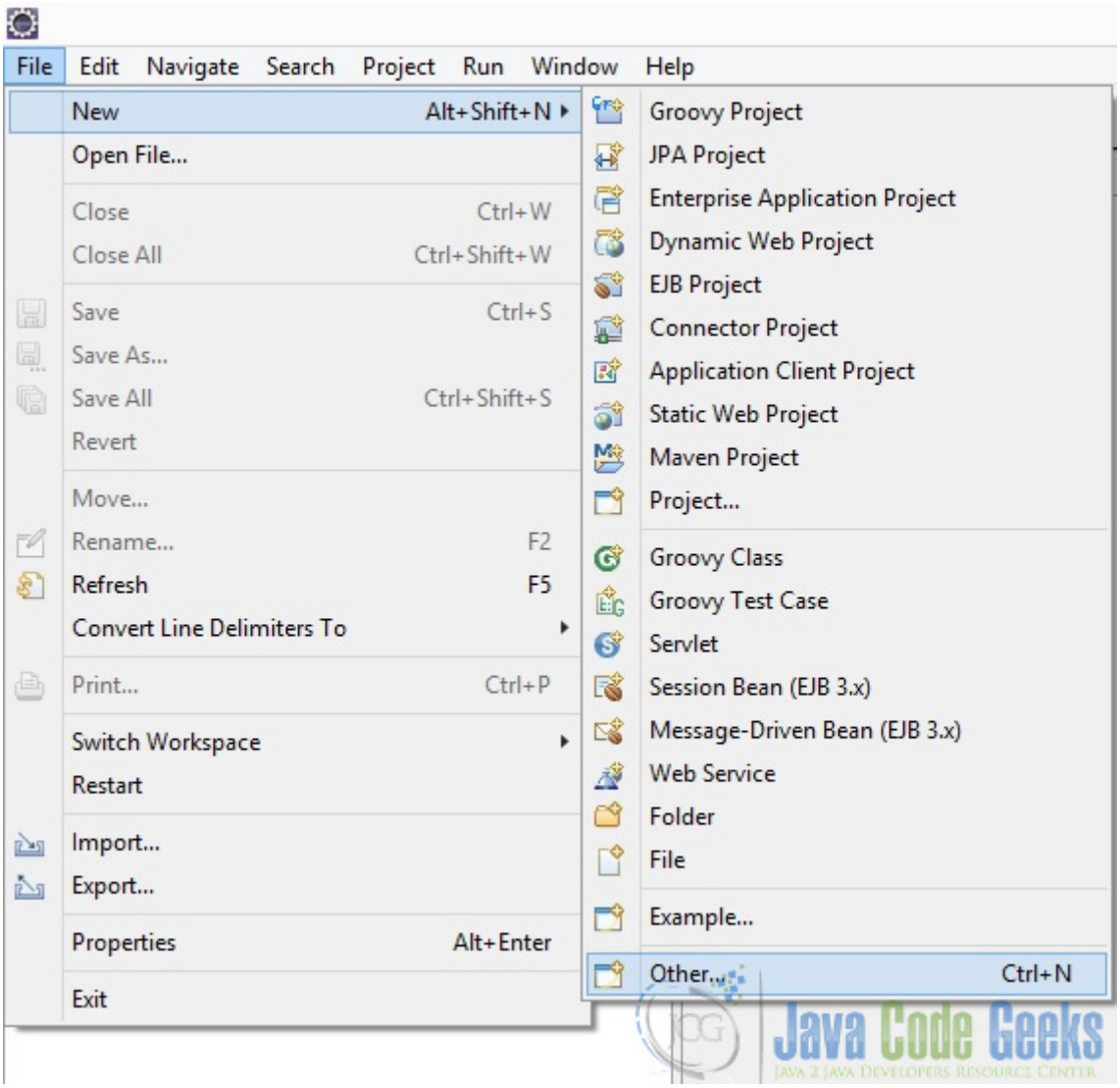

Figure 12.1: Create Gradle Project

Select “Gradle Project”:

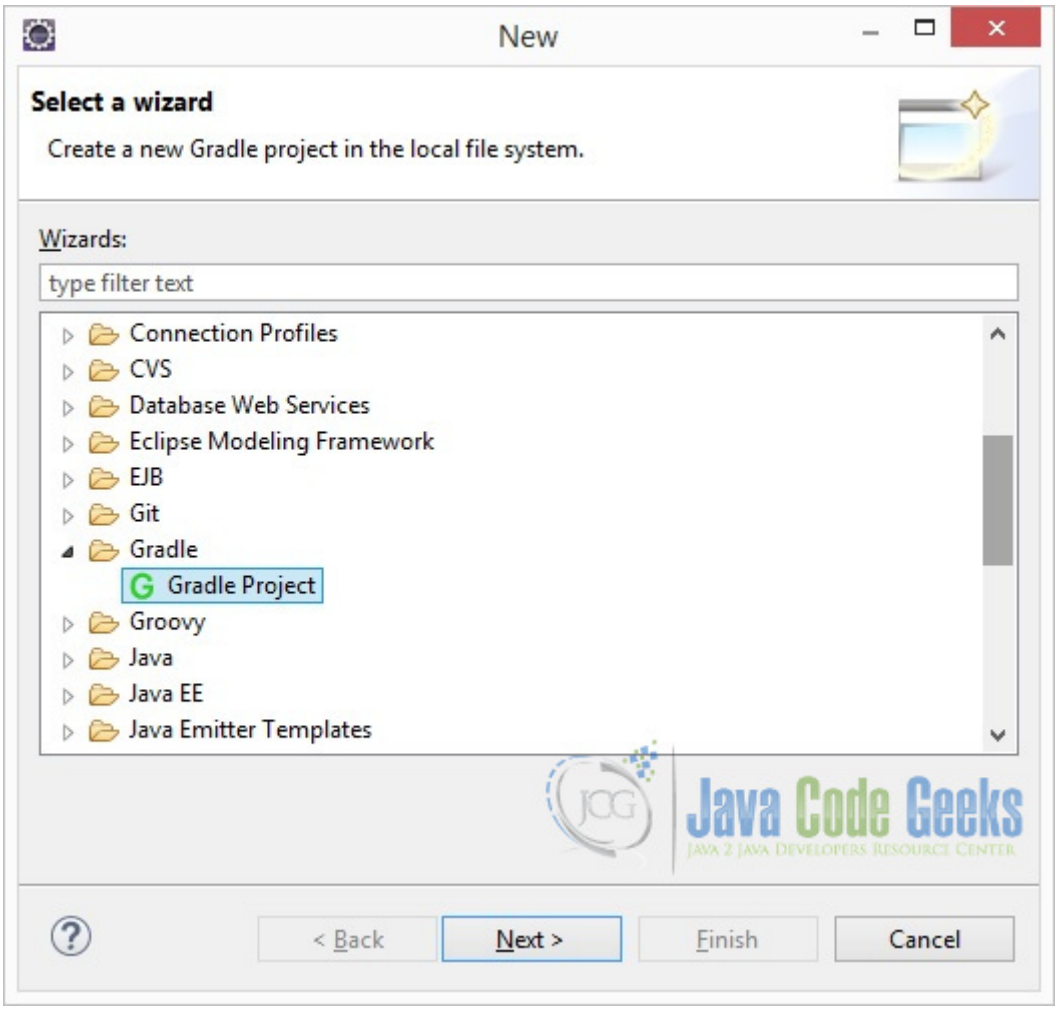

Figure 12.2: Create Gradle Project

Take a moment to read the suggestions in the following screen. Press next.

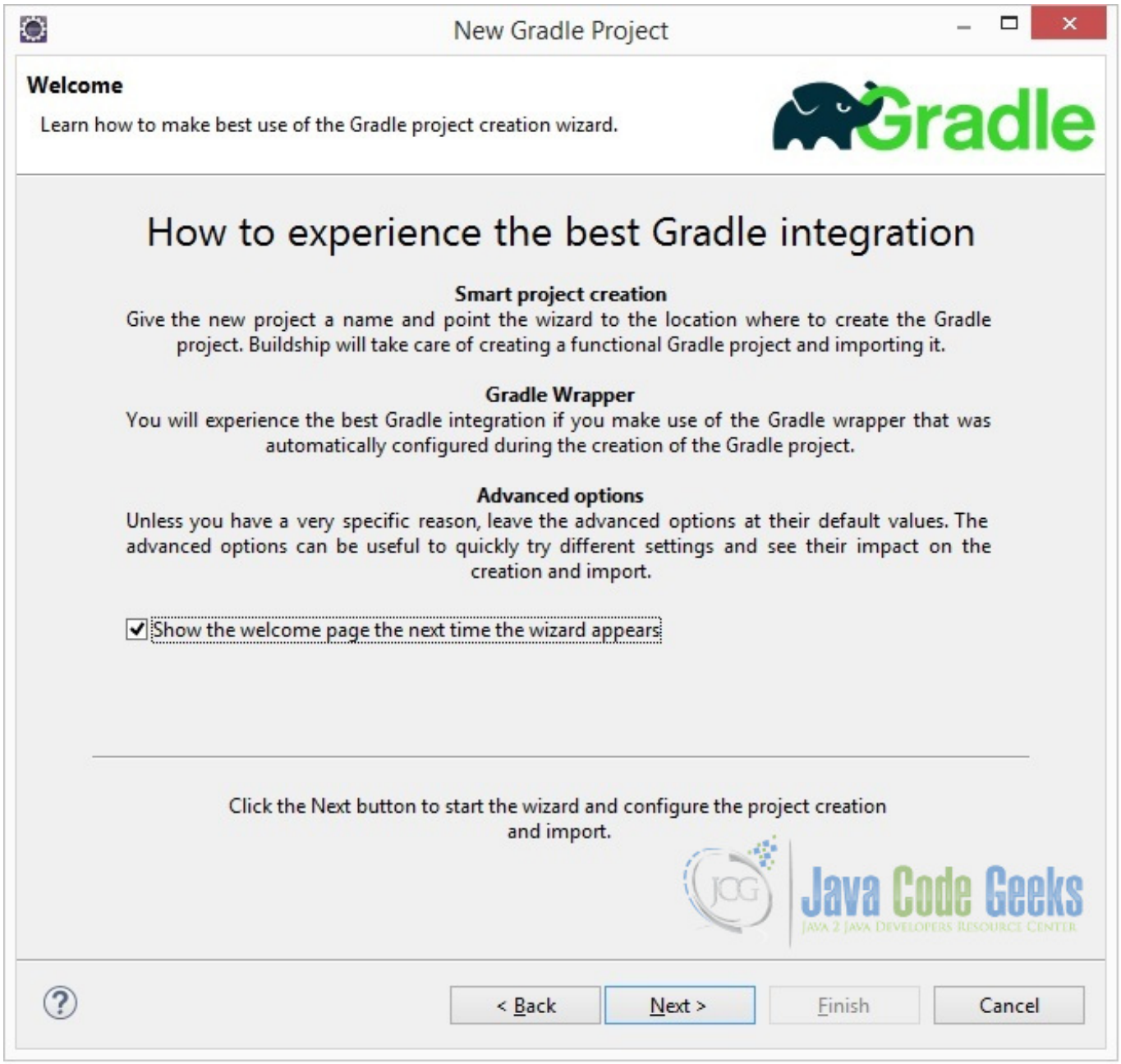

Figure 12.3: Create Gradle Project - Welcome Page

Enter the name of your project.

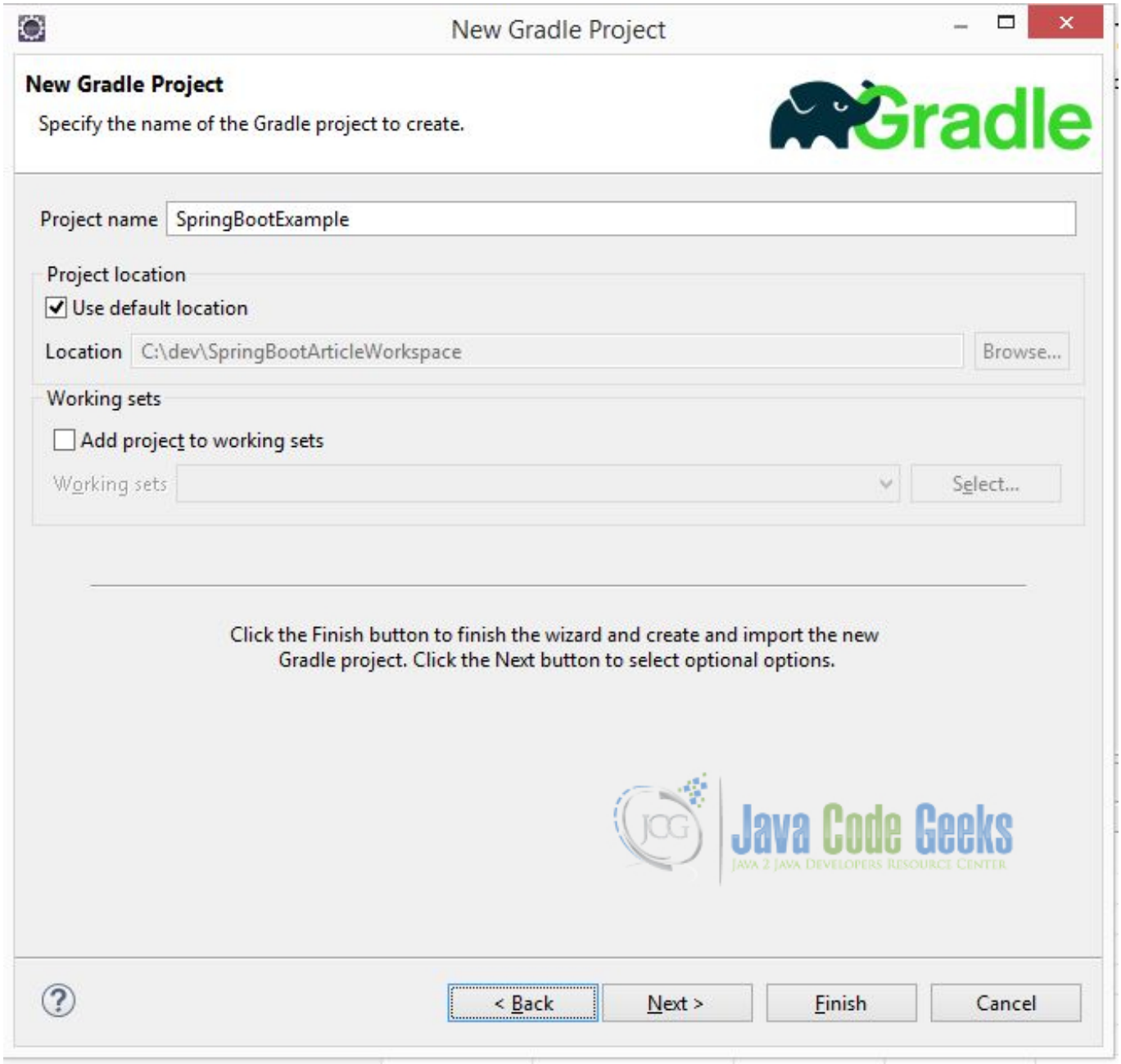

Figure 12.4: Enter Name of Gradle Project

Keep the default and recommended Gradle Wrapper option selected and press next.

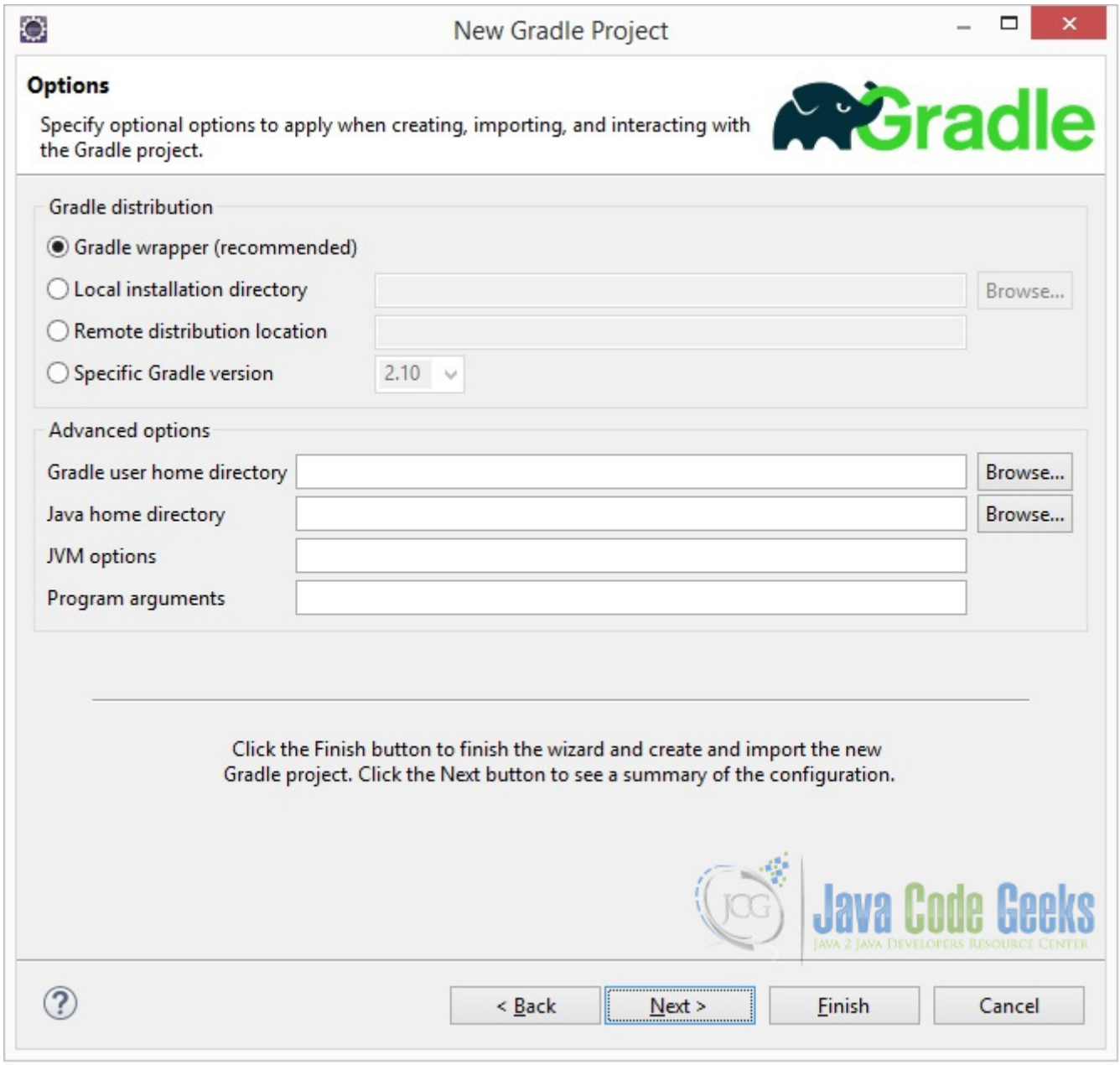

Figure 12.5: Create Gradle Project - Gradle Wrapper

Press finish on the preview screen.

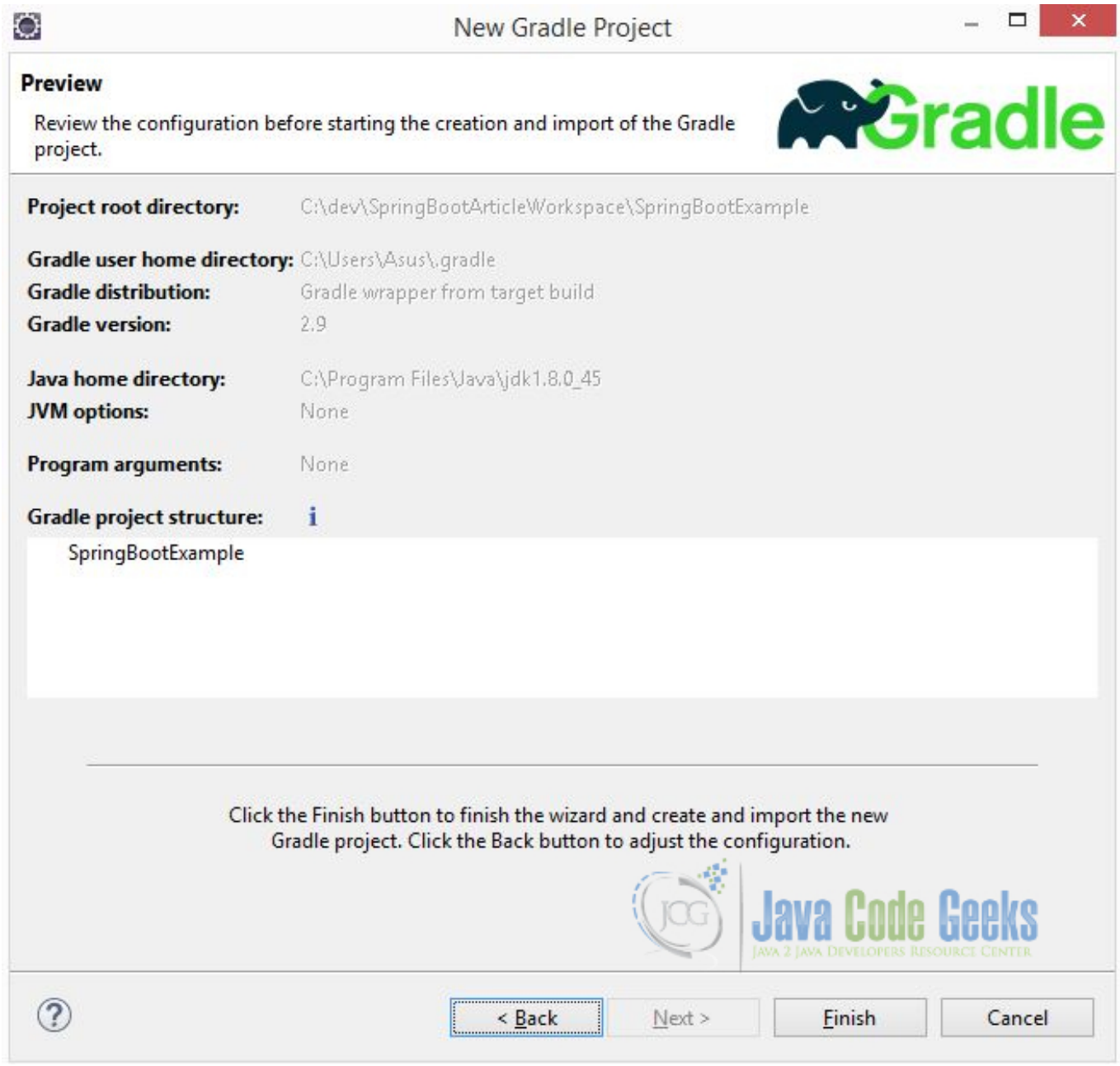

Figure 12.6: Finish Screen

You have successfully created the Gradle project. The following is the project structure of your Gradle project.

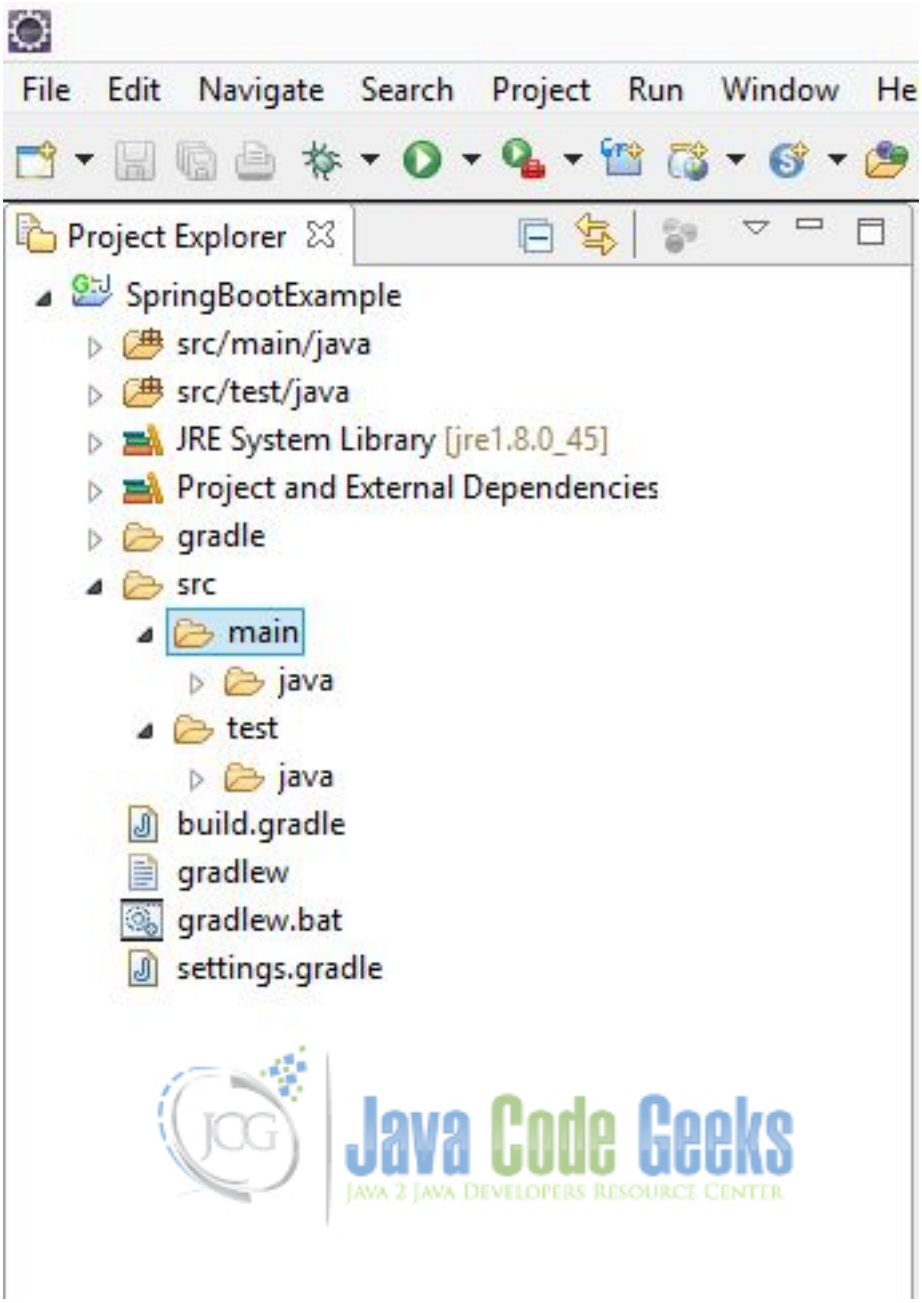

Figure 12.7: Gradle Project Structure

12.3.2 build.gradle

12.3.2.1 Modify build.gradle

In the Eclipse IDE, open the `build.gradle` file that is in the project root directory. Modify the file as shown below.

build.gradle

```
buildscript {
        repositories {
                mavenCentral()
        }
        dependencies {
```

```
                classpath("org.springframework.boot:spring-boot-gradle-plugin:1.3.2.RELEASE ↩
                    ")
        }
}

apply plugin: 'java'
apply plugin: 'eclipse'
apply plugin: 'spring-boot'

jar {
        baseName = 'sample'
        version =  '0.1.0'
}

repositories {
        mavenCentral()
}

sourceCompatibility = 1.8
targetCompatibility = 1.8

dependencies {
        compile("org.springframework.boot:spring-boot-starter-web")

}
```

12.3.2.2 Walk through build.gradle

```
buildscript {
    repositories {
        mavenCentral()
    }
    dependencies {
        classpath("org.springframework.boot:spring-boot-gradle-plugin:1.3.2.RELEASE")
    }
}

apply plugin:"spring-boot"
```

The `buildscript()` method is used to add any external libraries to script's classpath, by passing in a closure that declares the build script classpath using `classpath` configuration.

Any binary plugins that have been published as external jars can be added to the project by adding them to the `classpath` configuration and then applying the plugin.

In our example, `spring-boot-gradle-plugin` is the binary plugin that needs to be added to our project. And `apply plugin:"spring-boot"` is used to apply the plugin.

Spring Boot Gradle plugin is added to project to provide Spring Boot support in Gradle.

Follow the link Spring Boot Gradle plugin to dig further about this plugin.

```
apply plugin: 'java'
apply plugin: 'eclipse'
```

The `java` plugin adds compilation, testing and bundling capabilities to the project. When build using the tasks from the `eclipse` plugin, certain files are added to project to enable it to get imported into Eclipse IDE.

```
jar {
        baseName = 'sample'
        version =  '0.1.0'
}
```

When the project is built with `gradle clean build`, the jar file with name sample-0.1.0.jar is created in the `$PROJECT_ROOT/build/lib` folder of the project.

```
repositories {
        mavenCentral()
}
```

This is to specify the repository where the dependencies will be downloaded from.

```
sourceCompatibility = 1.8
targetCompatibility = 1.8
```

The `sourceCompatibility` is Java version compatibility to use when compiling Java source. Default value is version of the current JVM in use. The `targetCompatibility` is Java version to generate classes for. The default value is sourceCompatibility.

```
dependencies {
        compile("org.springframework.boot:spring-boot-starter-web")
}
```

To specify the required dependency for Spring Boot.

12.3.2.3 Run initial build

At the command prompt run:

```
gradle clean build
```

During build process, Spring Boot Gradle plugin looks for class with `public static void main()` to flag it as runnable class. As we haven't created a class with `public static void main()` yet, the build fails as shown below.

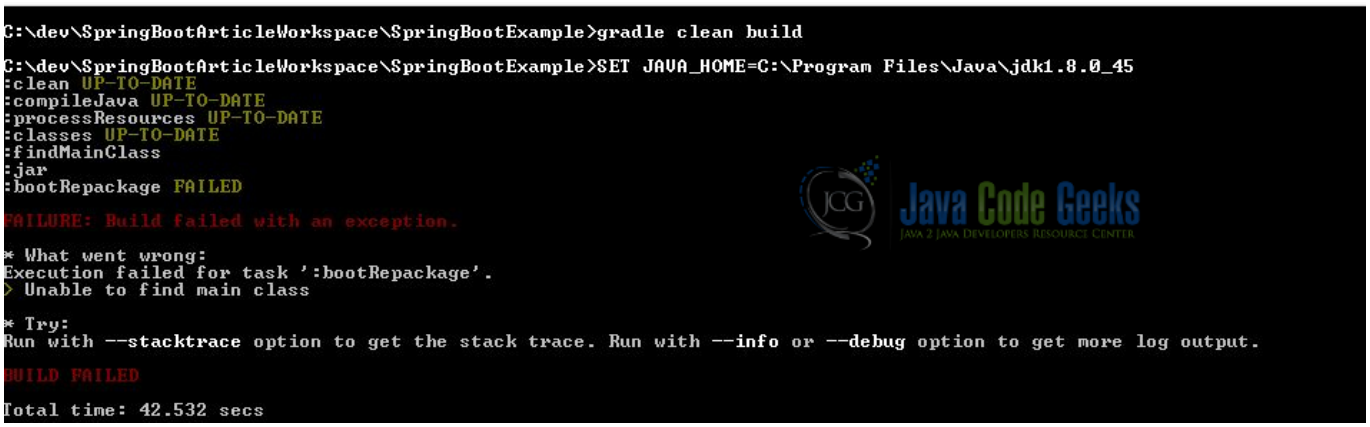

Figure 12.8: Initial Build Failure

12.3.3 Create SampleApplication.java

Let's create a simple runnable class with `public static void main()`. In Eclipse IDE, right click on source folder `src/main/java` and select New → Other.

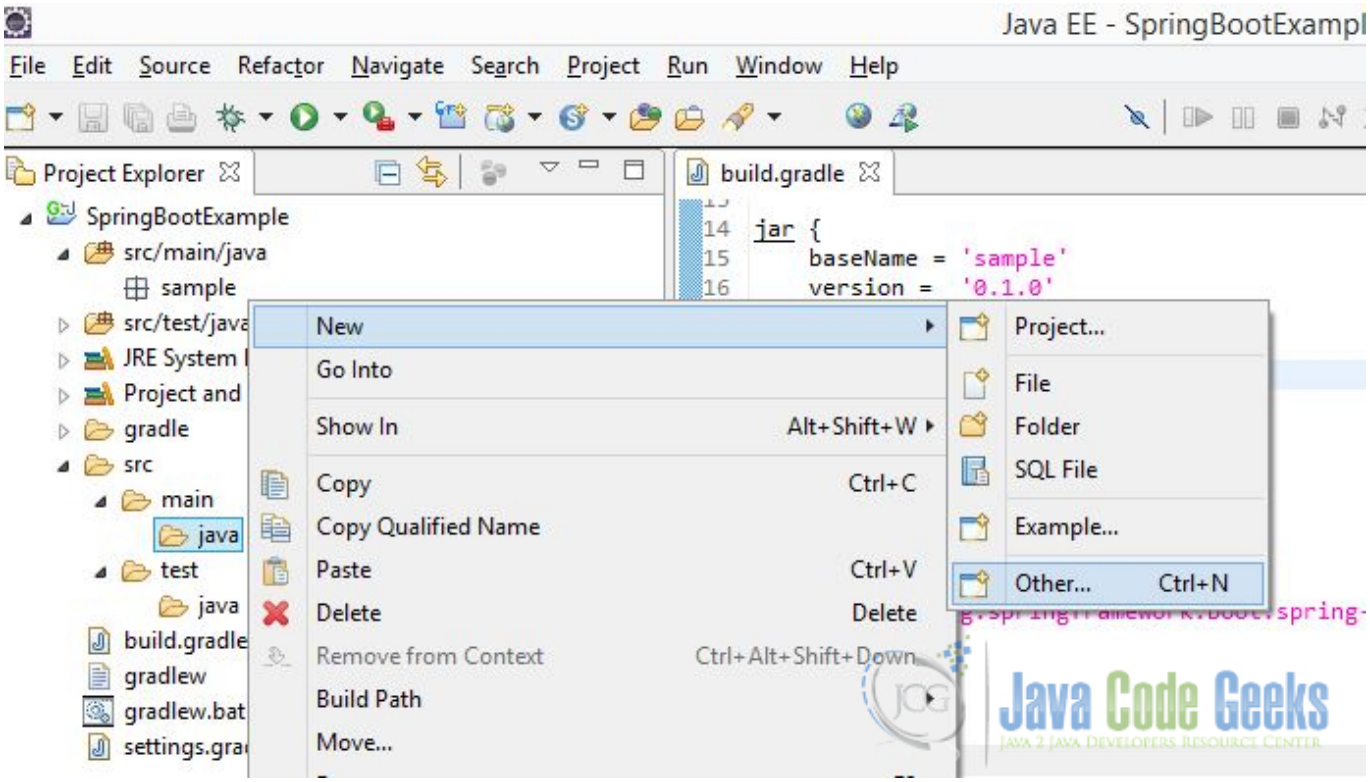

Figure 12.9: Create Sample Application - Step 1

Select "Class" in the resultant window. Click on "Next":

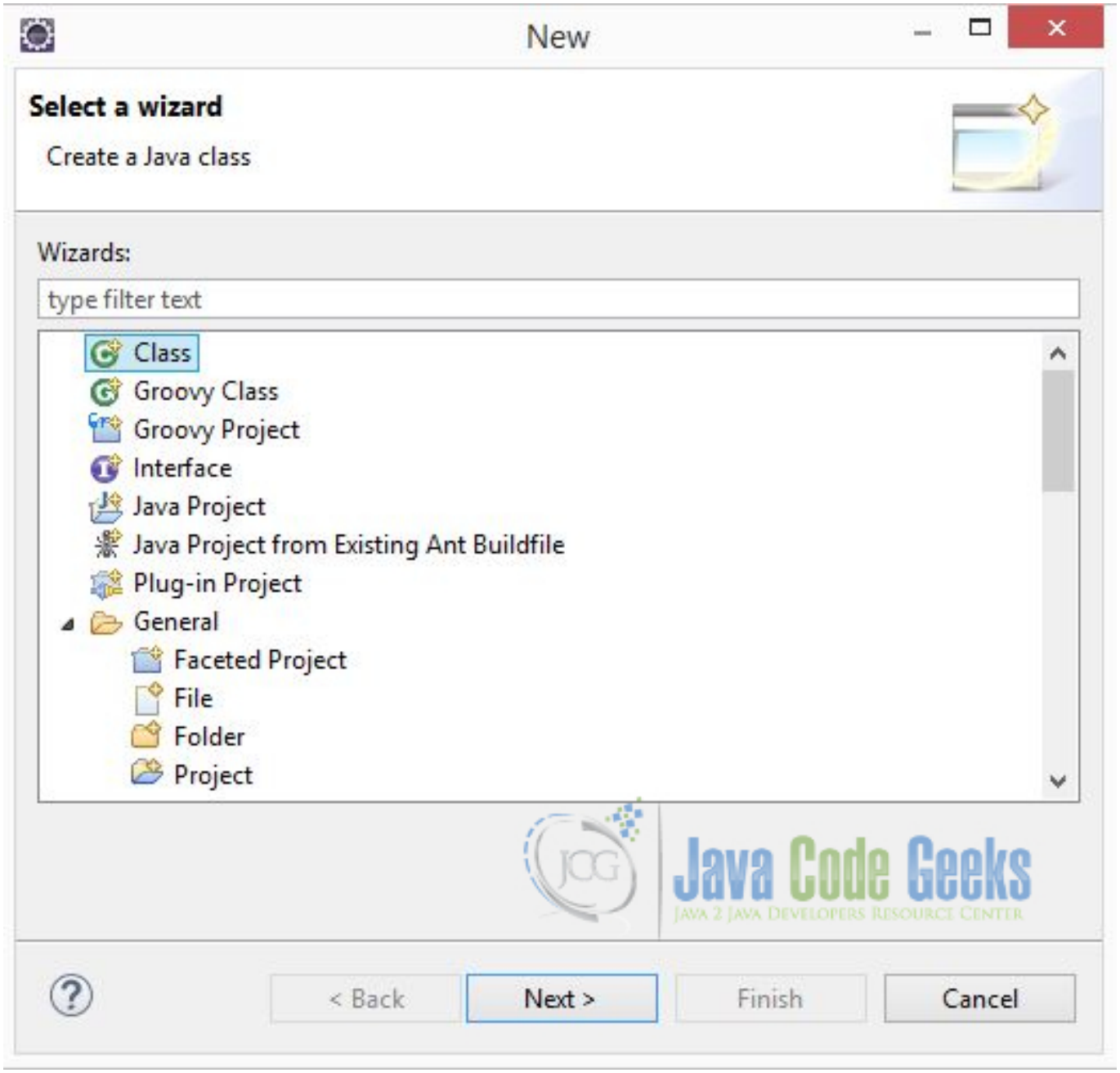

Figure 12.10: Create Sample Application - Step 2

Specify the package name "sample", class name "SampleApplication" and select to create `public static void main(String[] args)` method, as shown in the below picture. Click on "Finish".

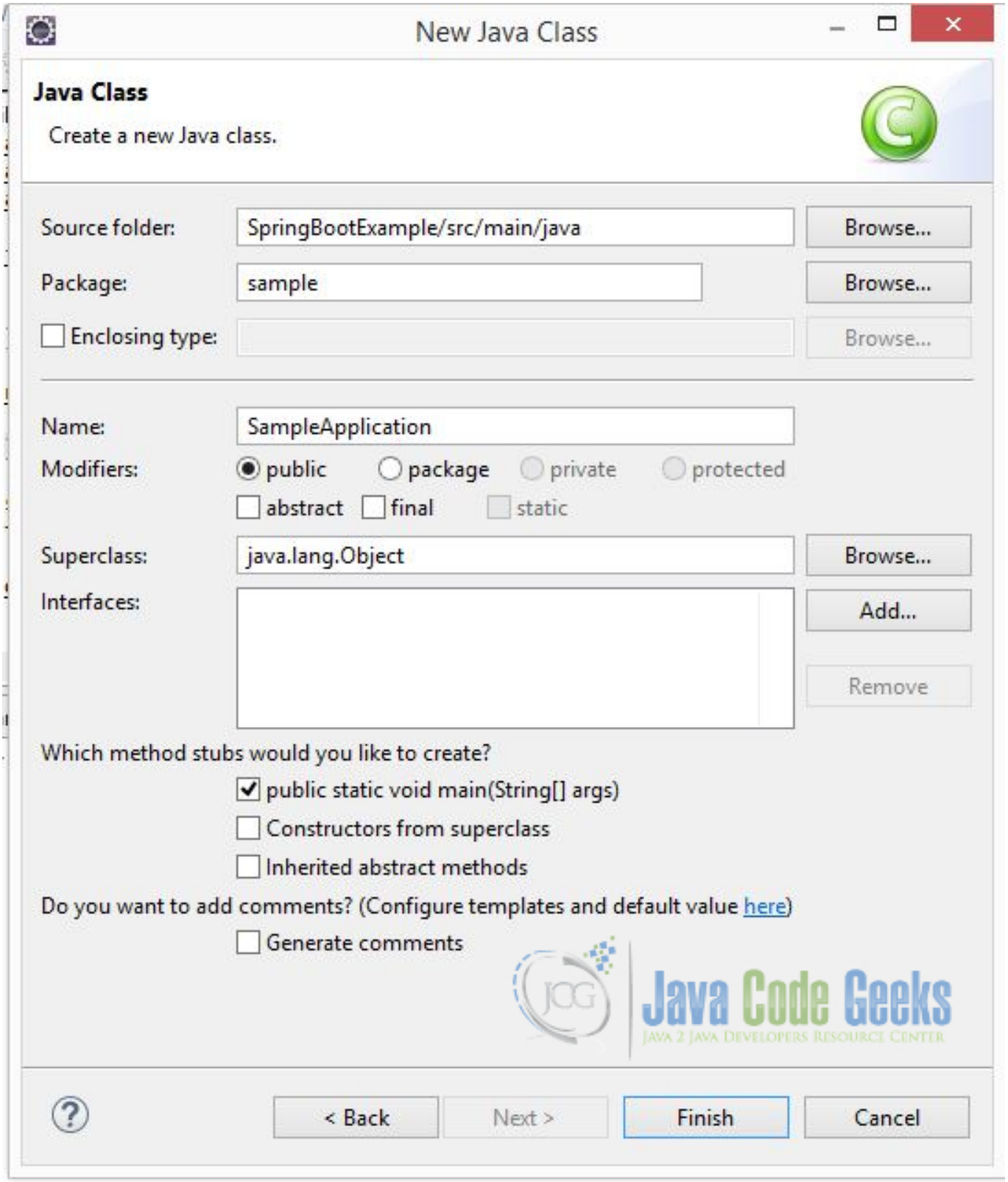

Figure 12.11: Create Sample Application - Step 3

After creating `SampleApplication.java`, the project structure looks as below.

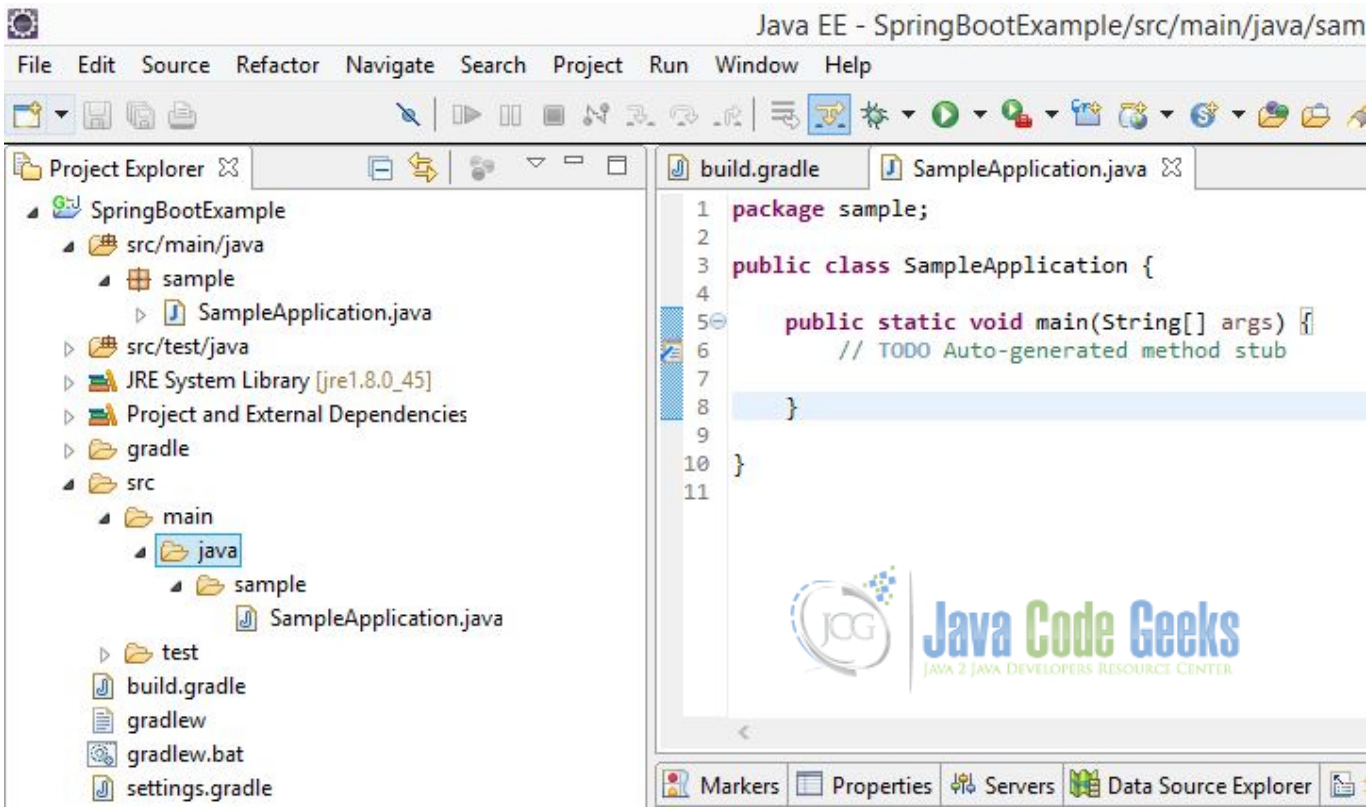

Figure 12.12: SampleApplication.java

At the command prompt run:

```
gradle clean build
```

As shown below the build is now successful.

Figure 12.13: Gradle Initial Build Success

As shown below Spring Boot makes reasonable assumptions of the dependencies and adds them automatically.

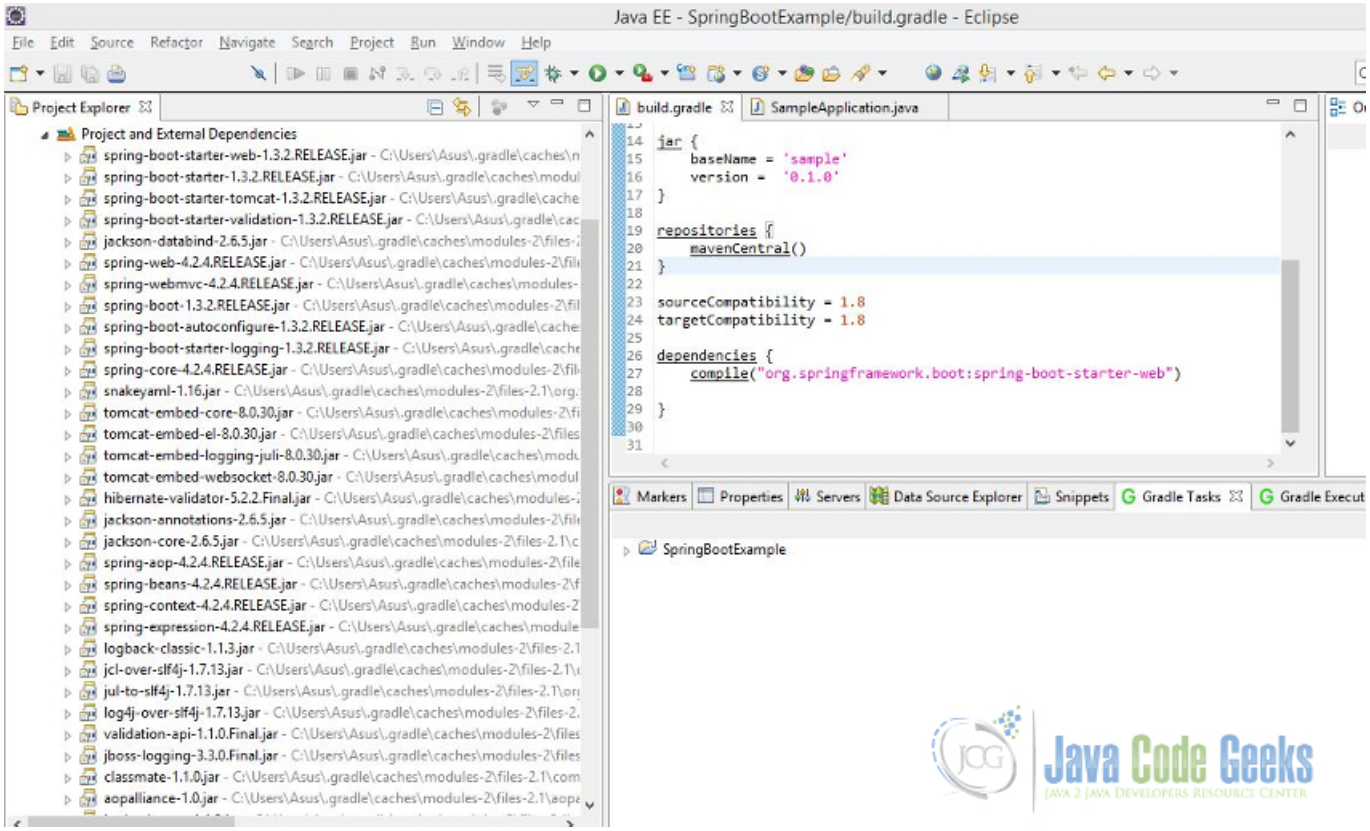

Figure 12.14: Project Dependencies added by Spring Boot.

12.3.4 Create SampleController.java

Lets us now create a simple controller `SampleController.java`

In Eclipse IDE, right click on source folder `src/main/java` and select New → Other.

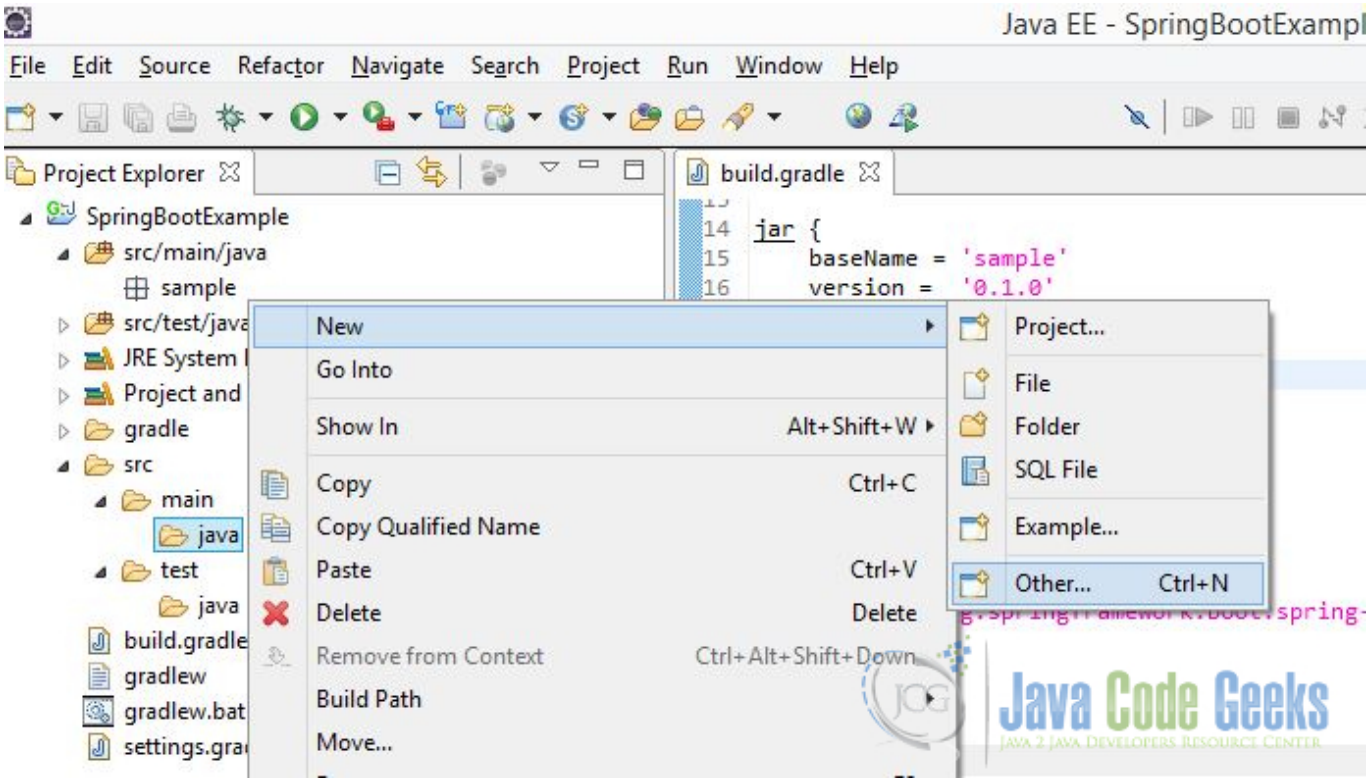

Figure 12.15: Create SampleController.java - Step 1

Select "Class" in the resultant window. Click on "Next":

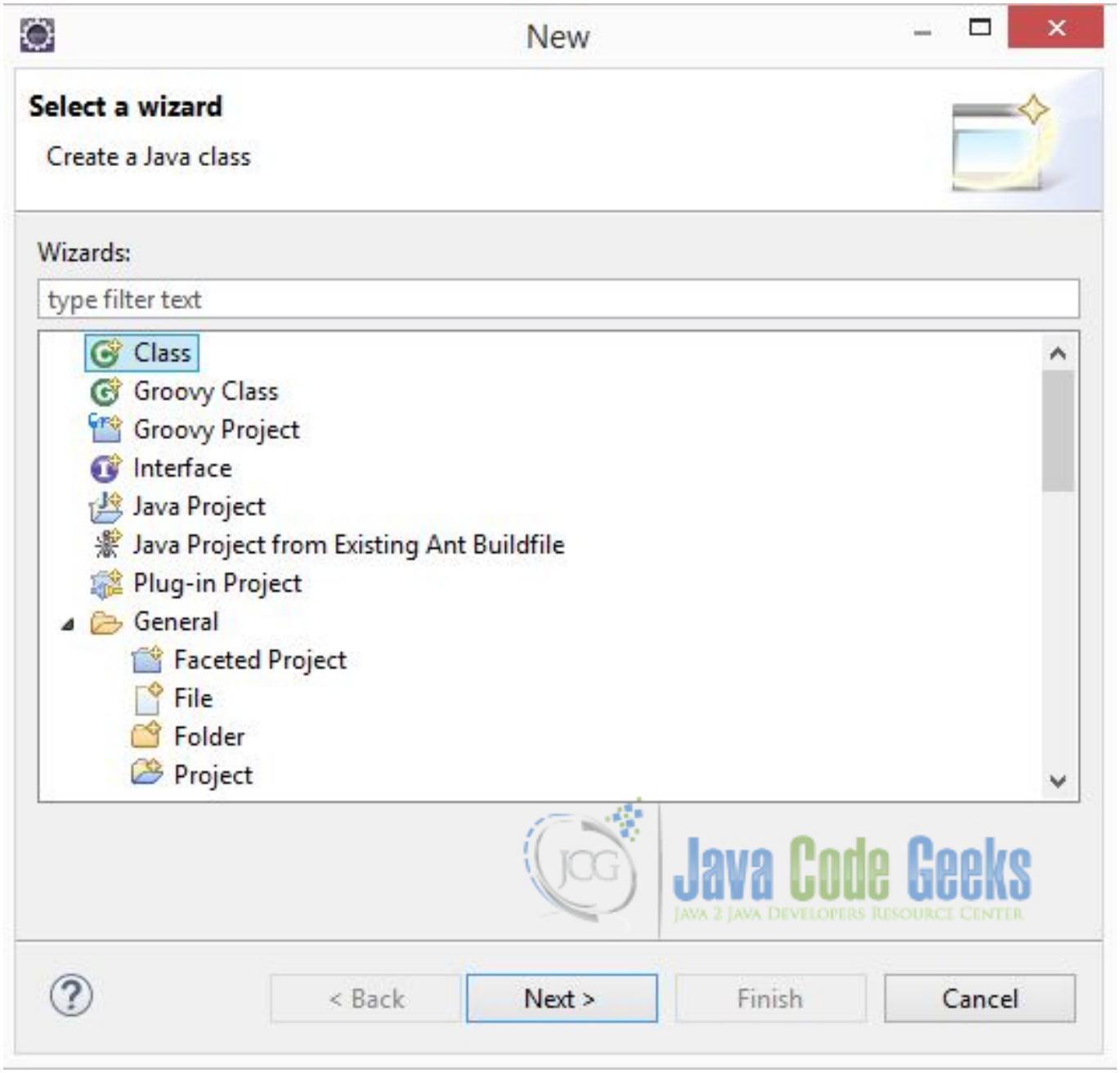

Figure 12.16: Create SampleController.java - Step 2

Specify the package name "sample", class name "SampleController". Click on "Finish".

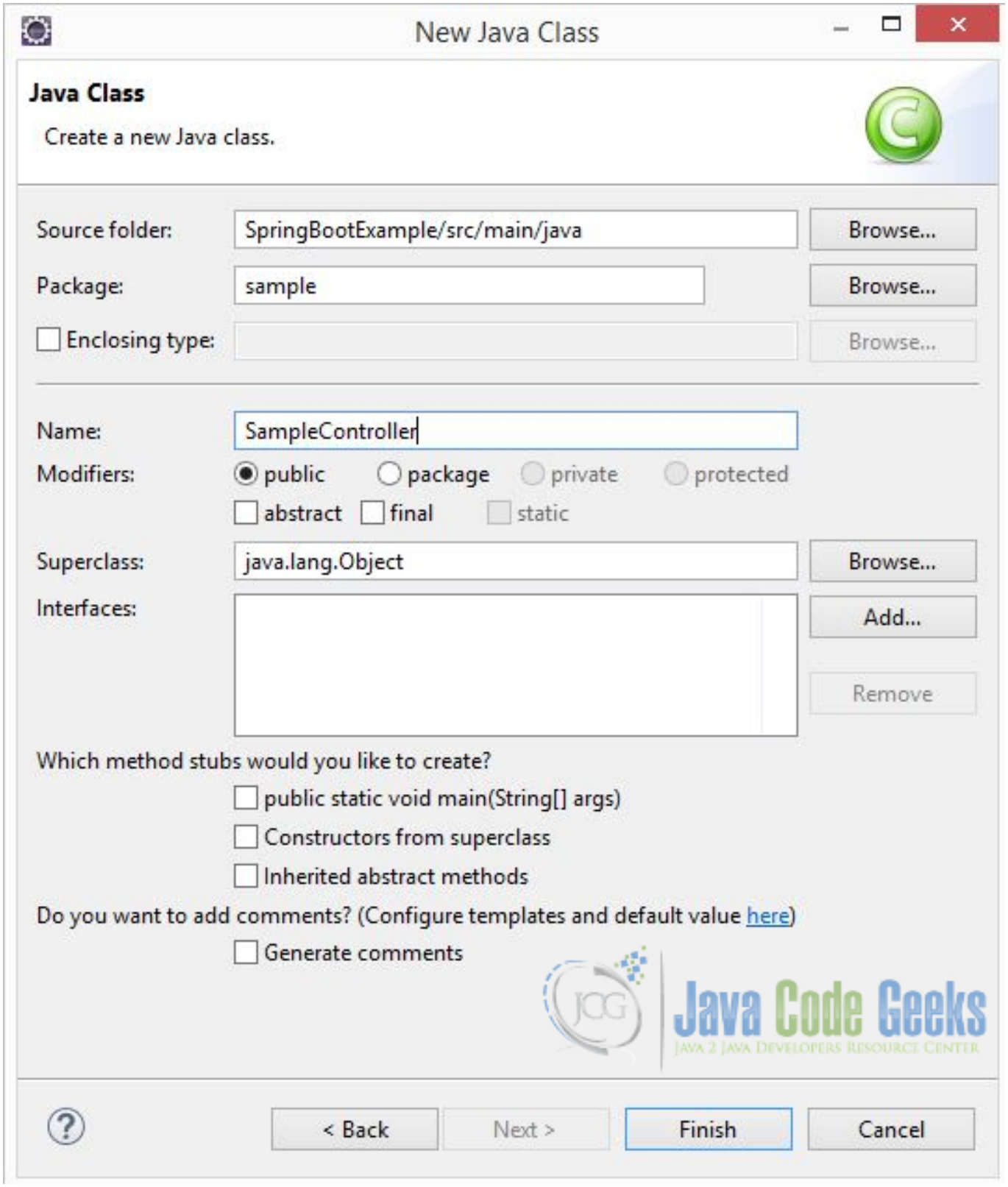

Figure 12.17: Create SampleController.java - Final step

Following is the current snapshot of the project structure.

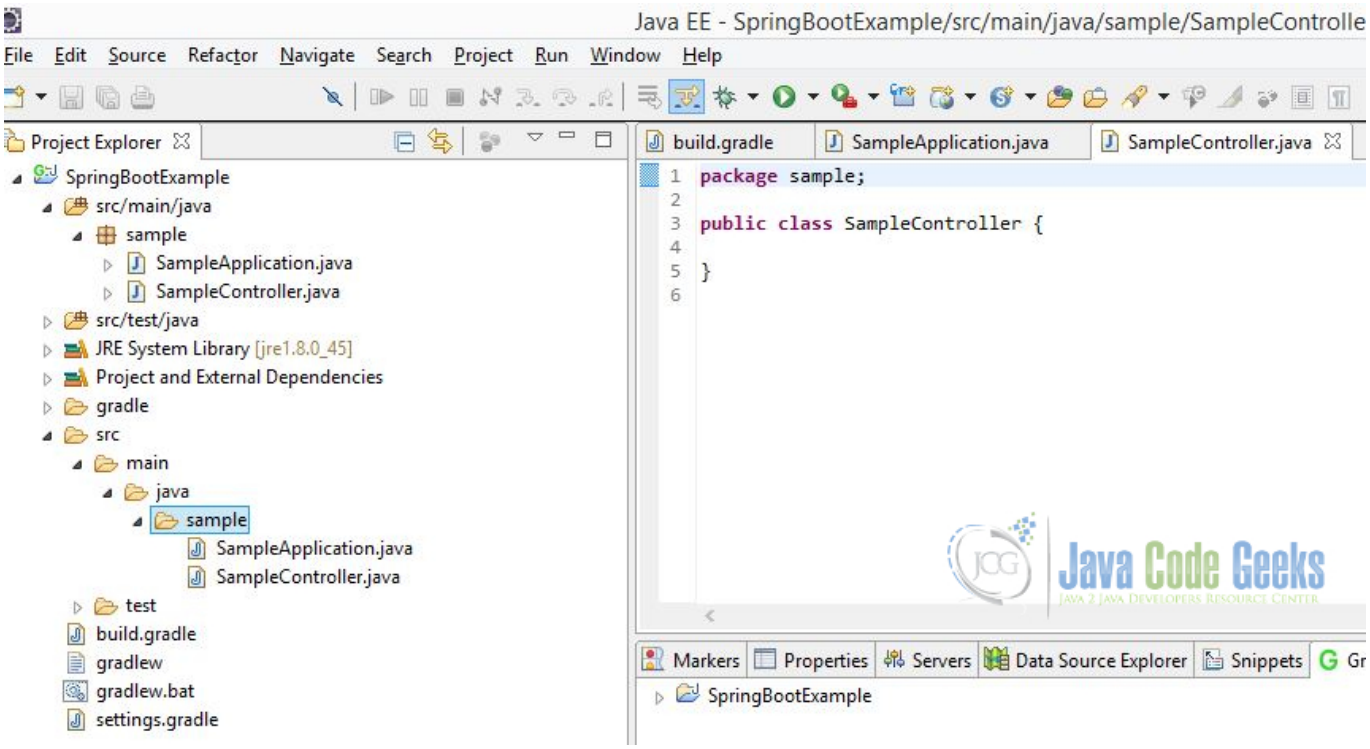

Figure 12.18: Project Structure Snapshot

Modify `SampleController.java` as shown below.

SampleController.java

```
package sample;

import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class SampleController {

        @RequestMapping("/sample")
        public String sampleIt(){
                return "Hello! Welcome to Spring Boot Sample. ";
        }
}
```

```
@RestController
```

The `@RestController` annotation marks the class as controller and adds `@Controller` and `@ResponseBody` annotations.

```
@RequestMapping
```

The `@RequestMapping` annotation ensures that HTTP requests to `/sample` is mapped to the `sampleIt()` method. As you would have already noticed, we didn't have to create any web.xml for the mapping.

12.3.5 SampleApplication.java

12.3.5.1 Modify SampleApplication.java

SampleApplication.java

```
package sample;

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

@SpringBootApplication
public class SampleApplication {

        public static void main(String[] args) {
                ApplicationContext ctx = SpringApplication.run(SampleApplication.class,  ↩
                    args);
                System.out.println(ctx.getDisplayName());

                System.out.println("This is my first Spring Boot Example");

        }

}
```

Let's explore through the `SampleApplication.java`:

```
@SpringBootApplication
```

If you are familiar with Spring framework, many-a-times you would have annotated your main class with `@Configuration`, `@ComponentScan` and `@EnableAutoConfiguration`. `@SpringBootApplication` is equivalent to using these three annotations `@Configuration`, `@ComponentScan` and `@EnableAutoConfiguration` with their default attributes.

To read further about these annotations, visit the links:

- `@Configuration`
- `@EnableAutoConfiguration`
- `@ComponentScan`

```
ApplicationContext ctx = SpringApplication.run(SampleApplication.class, args);
```

`SpringApplication` is used to bootstrap and launch a Spring application from the main method.

12.3.6 Run SampleApplication

At the command prompt:

```
gradle clean build bootRun
```

The output is as follows.

Figure 12.19: Gradle bootRun output

When invoked from browser the output is displayed as below.

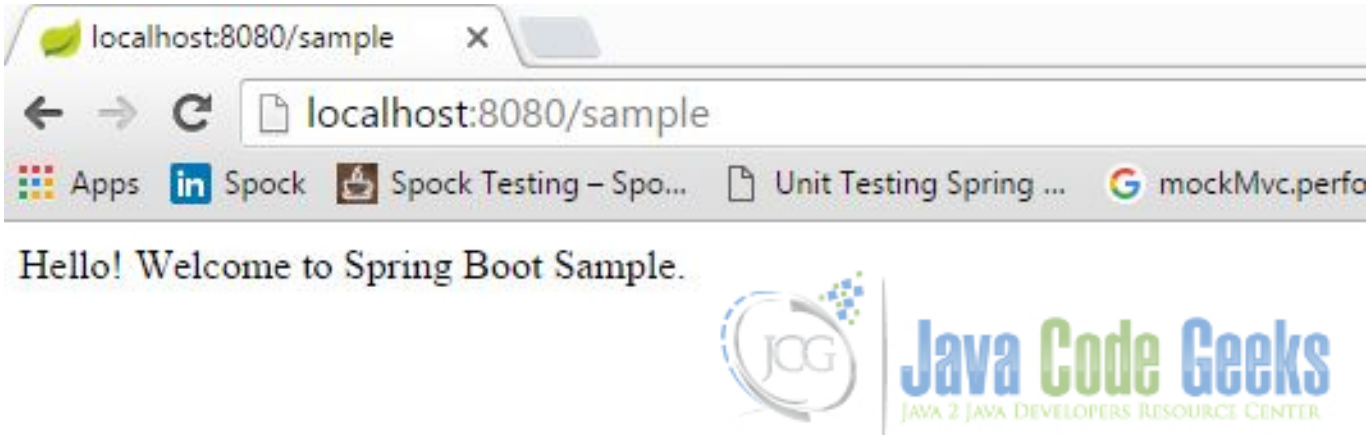

Figure 12.20: Browser Output

12.4 References

12.5 Conclusion

That's all Folks!! Make sure you make yourself a cuppa before jumping into additional reading through the links provided in "References" section.

12.6 Download the Eclipse project

Download

You can download the full source code of this example here: * Spring Boot Tutorial for Beginners*

Chapter 13

Spring Session Tutorial

In this example, we shall demonstrate how we can use Spring Session to improve user experience and maintain continuity of user sessions even in case of server failures.

13.1 Introduction

Spring Session is another very important Spring project that eases our task of HttpSession Management. It offers out of the box support for various `Session` related services like Multiple Browser Logins, maintaining user session state through server crashes i.e. Sticky Session etc.

Spring Session uses a filter, `org.springframework.web.filter.DelegatingFilterProxy`, which accepts the `HttpRequest` and constructs and injects its own Request object down the hierarchy. This way it gains control to the way new sessions are created, since the session object is attached to the `HttpRequest` Object.

The session information is stored in a database, Redis NoSQL database, in our case. As a result, even when the server crashes, the session data is maintained on the server. Redis is a key-value based NoSQL Database which can be easily associated with Spring Data as demonstrated here. The key for storing session is the `sessionId` and the value is the data associated with the user which is present in the session.

Let's take an example and see how it works:

13.2 Project Set-Up

Let's start by first setting up the project in Eclipse or any other you have in mind. We shall use Maven to setup our project. Open Eclipse and create a simple Maven project and check the skip archetype selection checkbox on the dialogue box that appears. Replace the content of the existing `pom.xml` with the one provided below:

pom.xml

```
<project xmlns="https://maven.apache.org/POM/4.0.0" xmlns:xsi="https://www.w3.org/2001/ ↩
    XMLSchema-instance"
        xsi:schemaLocation="https://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd ↩
            /maven-4.0.0.xsd">
        <modelVersion>4.0.0</modelVersion>
        <groupId>com.jcg.examples.springSessionExample</groupId>
        <artifactId>SpringSessionExample</artifactId>
        <version>0.0.1-SNAPSHOT</version>
        <packaging>war</packaging>

        <dependencies>
                <dependency>
                        <groupId>org.springframework.session</groupId>
```

```
                        <artifactId>spring-session</artifactId>
                        <version>1.2.0.RC3</version>
                </dependency>
                <dependency>
                        <groupId>org.springframework.data</groupId>
                        <artifactId>spring-data-redis</artifactId>
                        <version>1.7.1.RELEASE</version>
                </dependency>
                <dependency>
                        <groupId>org.springframework.security</groupId>
                        <artifactId>spring-security-web</artifactId>
                        <version>4.0.3.RELEASE</version>
                </dependency>
                <dependency>
                        <groupId>org.apache.commons</groupId>
                        <artifactId>commons-pool2</artifactId>
                        <version>2.4.1</version>
                </dependency>
                <dependency>
                        <groupId>redis.clients</groupId>
                        <artifactId>jedis</artifactId>
                        <version>2.7.3</version>
                </dependency>


        </dependencies>
        <repositories>
                <repository>
                        <id>spring-milestones</id>
                        <name>Spring Milestones</name>
                        <url>https://repo.spring.io/libs-milestone</url>
                        <snapshots>
                                <enabled>false</enabled>
                        </snapshots>
                </repository>
        </repositories>

        <build>
                <sourceDirectory>src</sourceDirectory>
                <plugins>
                        <plugin>
                                <artifactId>maven-compiler-plugin</artifactId>
                                <version>3.3</version>
                                <configuration>
                                        <source>1.8</source>
                                        <target>1.8</target>
                                </configuration>
                        </plugin>
                        <plugin>
                                <artifactId>maven-war-plugin</artifactId>
                                <version>2.6</version>
                                <configuration>
                                        <warSourceDirectory>WebContent</warSourceDirectory>
                                        <failOnMissingWebXml>false</failOnMissingWebXml>
                                </configuration>
                        </plugin>
                </plugins>
        </build>
</project>
```

This will import the required JAR dependencies in the project. We can now start with the actual implementation of the `Spring`

`Session` in our project.

13.3 Implementation

13.3.1 Sticky Session

We start the implementation by first configuring the `springSessionRepositoryFilter` filter which will inject the Spring Request object in stead of the original HttpRequest object. The `web.xml` looks like :

web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="https://www.w3.org/2001/XMLSchema-instance"
        xmlns="https://java.sun.com/xml/ns/javaee"
        xsi:schemaLocation="https://java.sun.com/xml/ns/javaee https://java.sun.com/xml/ns/ ↩
            javaee/web-app_2_5.xsd"
        id="WebApp_ID" version="2.5">
        <display-name>SpringMVCloginExample</display-name>
        <filter>
                <filter-name>springSessionRepositoryFilter</filter-name>
                <filter-class>org.springframework.web.filter.DelegatingFilterProxy</filter- ↩
                    class>
        </filter>
        <filter-mapping>
                <filter-name>springSessionRepositoryFilter</filter-name>
                <url-pattern>/*</url-pattern>
        </filter-mapping>
        <context-param>
                <param-name>contextConfigLocation</param-name>
                <param-value>classpath://spring-config.xml</param-value>
        </context-param>

        <listener>
                <listener-class>org.springframework.web.context.ContextLoaderListener</ ↩
                    listener-class>
        </listener>
        <session-config>
                <session-timeout>1</session-timeout>
        </session-config>
</web-app>
```

Next, we need to configure the spring container so that it can inject the `DAOs` for the `Redis Repositories`.

spring-config.xml

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

        <context:component-scan base-package="com.jcg.examples" />
        <bean
class="org.springframework.session.data.redis.config.annotation.web.http. ↩
    RedisHttpSessionConfiguration" />
```

```
        <bean
                class="org.springframework.data.redis.connection.jedis. ↩
                    JedisConnectionFactory" />

</beans>
```

Now all the configuration is in place for the Spring Container. For ease of understanding, I have used plain J2EE with Servlets and JSP's. The requests from the browser are directed to the Servlet : `CustomServlet` class below. The servlet simply adds the username if present in the request to the `HttpSession`.

CustomServlet.java

```
package com.jcg.examples.servlet;

import java.io.IOException;

import javax.servlet.RequestDispatcher;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;

import com.jcg.examples.service.MultiLoginService;


@WebServlet("/customServlet")
public class CustomServlet extends HttpServlet
{
        private static final long serialVersionUID = 1L;

        protected void doGet(HttpServletRequest request, HttpServletResponse response)  ↩
            throws ServletException, IOException
        {
                doPost(request, response);
        }

        protected void doPost(HttpServletRequest request, HttpServletResponse response)  ↩
            throws ServletException, IOException
        {
                HttpSession session = request.getSession();

                String userName = request.getParameter("userName");

                if(userName != null)
                {
                        session.setAttribute("Username", request.getParameter("userName"));
                }

                RequestDispatcher rd = request.getRequestDispatcher("welcome.jsp");

                rd.forward(request, response);
        }
}
```

Here's the simple `welcome.jsp` just to display the `username` passed via the browser. If no username is passed, it simply prints `Hello World!`

welcome.jsp

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
        pageEncoding="ISO-8859-1"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "https://www.w3.org/TR/html4 ↩
    /loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Welcome</title>
</head>
<body>
        <%
                        String userName = (String) session.getAttribute("Username");
                        String additionalURL = (String) request.getAttribute("alias");
                        if (userName != null)
                        {
                                        out.write("Hello" + userName);
                        }
                        else
                        {
                                        out.write("Hello World!");
                        }
                        if (additionalURL == null)
                        {
                                        additionalURL = "";
                        }
        %>

</body>
</html>
```

How it Works : When a new request comes from the browser, the `springSessionRepositoryFilter` intercepts it. It substitutes the `HttpRequest` object with its own implementation which is created using the original Request. This new `Request` object also holds reference to the substituted `HttpSession` Wrapper instead of the plain `javax.servlet.http.HttpSession` object.

The changes made to this new session object like addition or removal of attributes is persisted into the Redis Database Server without the developer writing any code for this. Since the session is data is persisted in the database instead of the Application server memory, the session data is available to other Application servers as well.

As a result, even when any of the node fails another server from the node group can take-up without any interruption noticed by the end-user(sticky-session).

For the Spring session to be able to persist the data, it is essential that the Redis Server should be running.

Let's have a look at demonstrating what we have learned so far in this example:

To test our Spring Session demo, we have two Tomcat servers with our application deployed at both the servers. Both the applications are pointing to the same `Redis` server which stores the user session information. The thing the readers should note here is that, `Redis` server stores data in the form of key-value pairs. The key being the `Jsession-Id` and the values being whatever we want to store in the user session.

When we first hit one of the server running on port `8084`:

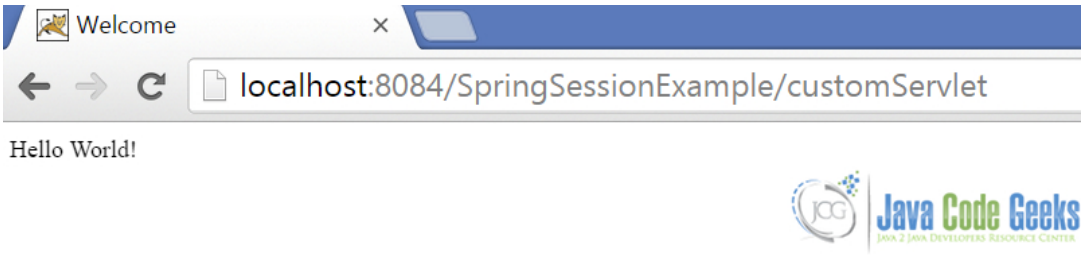

Figure 13.1: Without Request Parameters

Passing the `userName` parameter via the URL:

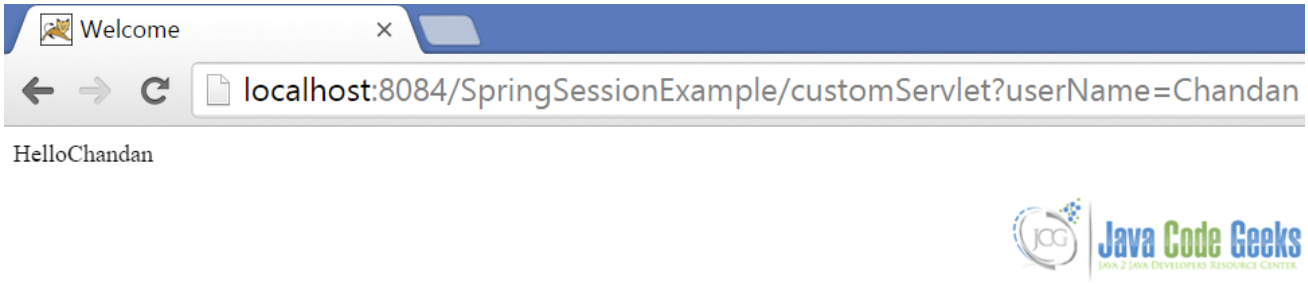

Figure 13.2: Request Parameter passed in First Server

Now we hit the other server which also has our application deployed on port `8088`:

Figure 13.3: UserName is present on other server

As you can see in the image above even though we have not passed the username in the url, the server is able to display the username associated with that session.

Here's the session information stored in the Redis Server:

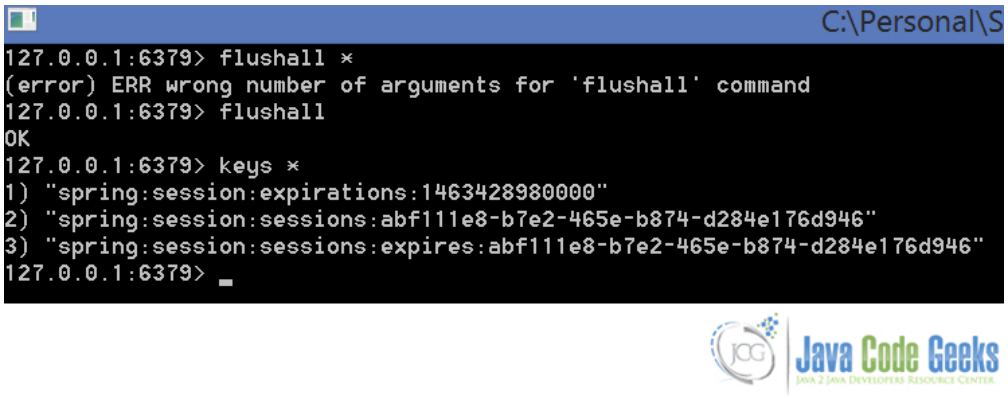

Figure 13.4: Data in Redis Server

This is all about maintaining the session when the application server fails.

13.3.2 Single Sign On

Another feature about Spring Session is that it can be used for multiple logins from the same browser. This is particularly useful when we are building multiple applications that permit the same credentials set. This functionality is called Single `Sign-On`.

Let's modify our code so that we can implement this single sign on functionality we just discussed:

In the `Servlet`, we need to add a call to the `MultiLoginService` class, so that appropriate session aliases are assigned to them.

CustomServlet.java

```
package com.jcg.examples.servlet;


import java.io.IOException;

import javax.servlet.RequestDispatcher;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;

import com.jcg.examples.service.MultiLoginService;


@WebServlet("/customServlet")
public class CustomServlet extends HttpServlet
{
                private static final long serialVersionUID = 1L;

                protected void doGet(HttpServletRequest request, HttpServletResponse  ↩
                    response) throws ServletException, IOException
                {
                                doPost(request, response);
                }

                protected void doPost(HttpServletRequest request, HttpServletResponse  ↩
                    response) throws ServletException, IOException
                {
```

```
                                HttpSession session = request.getSession();

                                String userName = request.getParameter("userName");

                                if(userName != null && !"".equals(userName))
                                {
                                                session.setAttribute("Username", request. ↩
                                                    getParameter("userName"));
                                }

                                MultiLoginService.createMultiLogin(request);

                                RequestDispatcher rd = request.getRequestDispatcher(" ↩
                                    welcome.jsp");

                                rd.forward(request, response);
                }

}
```

Session alias is nothing but a random string that we use to keep a mapping of associated session Ids. We can get this alias by calling the `sessionManager.getNewSessionAlias` method as shown in line 40 of the MultiLoginService class below.

MultiLoginService.java

```
package com.jcg.examples.service;

import java.util.Map;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpSession;

import org.springframework.session.Session;
import org.springframework.session.SessionRepository;
import org.springframework.session.web.http.HttpSessionManager;

public class MultiLoginService
{
                public static void createMultiLogin(HttpServletRequest httpRequest)
                {
                                HttpSessionManager sessionManager =(HttpSessionManager)  ↩
                                    httpRequest.getAttribute(HttpSessionManager.class. ↩
                                    getName());
                                String alias = httpRequest.getParameter("_s");
                                @SuppressWarnings("unchecked")
                                SessionRepository<Session> sessionRepository =
                (SessionRepository<Session>) httpRequest.getAttribute(SessionRepository. ↩
                    class.getName());


                                for(Map.Entry<String, String> entry : sessionManager. ↩
                                    getSessionIds(httpRequest).entrySet()) {
            String aliasId = entry.getKey();
            String sessionId = entry.getValue();

            Session storedSession = sessionRepository.getSession(sessionId);
            HttpSession httpSession = httpRequest.getSession();
            if(storedSession != null && storedSession.getAttribute("Username") != null &&  ↩
                httpSession.getAttribute("Username") == null )
            {
                            httpSession.setAttribute("Username", storedSession.getAttribute(" ↩
                                Username"));
```

```
            }


            System.out.println(aliasId +" : "+sessionId);
                                }

                                if(alias == null || "".equals(alias.trim()))
                                {
                                                alias = sessionManager.getNewSessionAlias( ↩
                                                    httpRequest);
                                }
                                httpRequest.setAttribute("alias",alias);
                }
}
```

The alias is named `_s` by default. This alias needs to be present in every request for the application to decide the correct session mapping. In the absence of this alias, the application maps the incoming request session to alias with value as `_s=0`.

Below is the `jsp` that holds the `_s` variable in a hidden field and the value is submitted alongwith every request.

welcome.jsp

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
        pageEncoding="ISO-8859-1"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "https://www.w3.org/TR/html4 ↩
    /loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Welcome</title>
</head>
<body>
        <%
                        String userName = (String) session.getAttribute("Username");
                        String additionalURL = (String) request.getAttribute("alias");
                        if (userName != null)
                        {
                                        out.write("Hello " + userName+ "!");
                        }
                        else
                        {
                                        out.write("Hello World!");
                        }
                        if (additionalURL == null)
                        {
                                        additionalURL = "";
                        }
        %>
        <form method="post" action="<%=request.getContextPath()%>/customServlet">
                <input type="text" name = "userName" >
                <br/>
                <input type="submit" value="View My Name"/>
                <input type="hidden" name="_s" value="<%=additionalURL%>"/>

        </form>



</body>
</html>
```

Let's test out this functionality:

Initial Page:

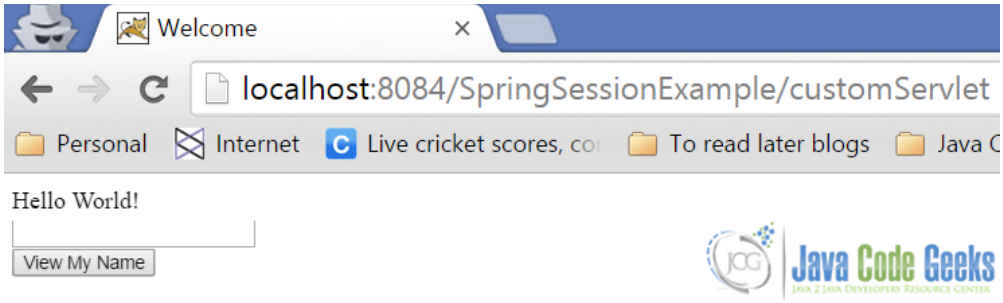

Figure 13.5: Initial Page without Parameters

Upon entering the text : `Chandan` and clicking on the `View My Name` Button.

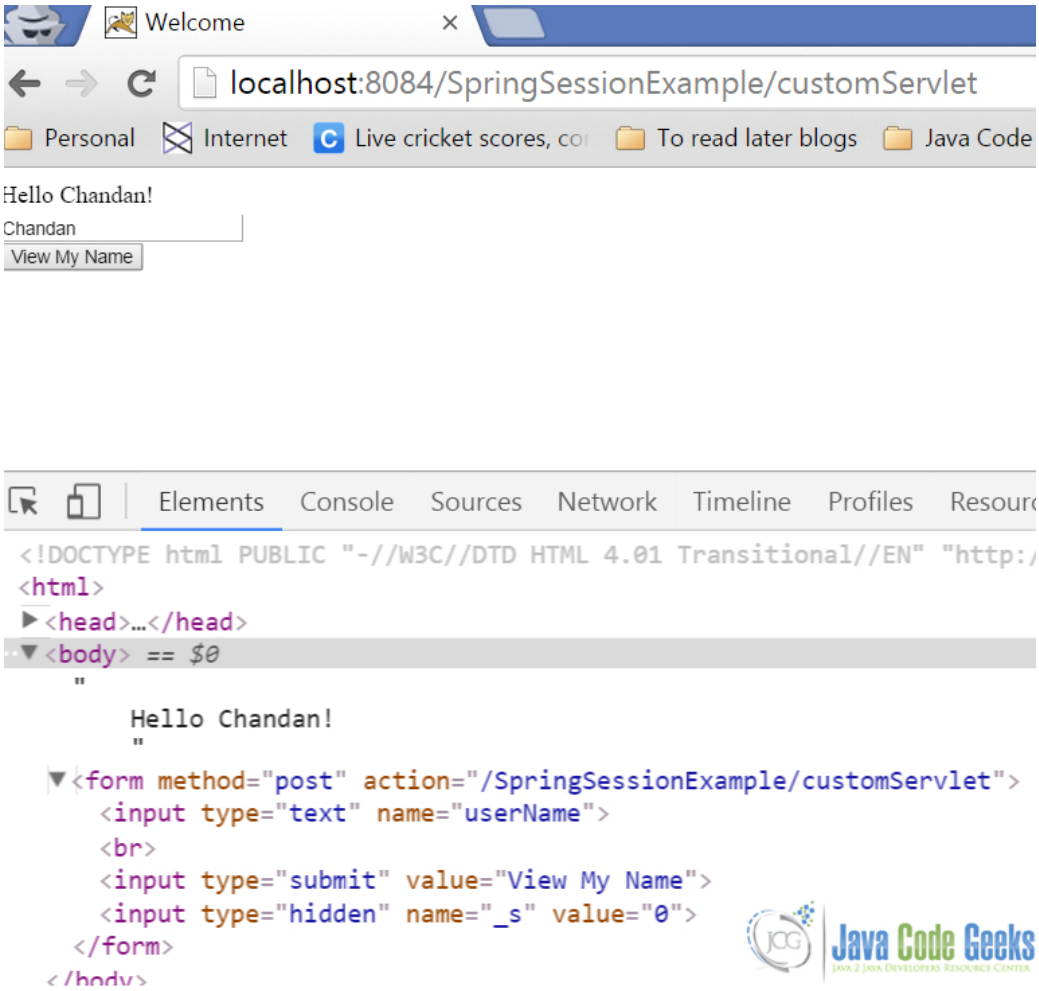

Figure 13.6: Session One

Same session in another browser tab but with different parameter :

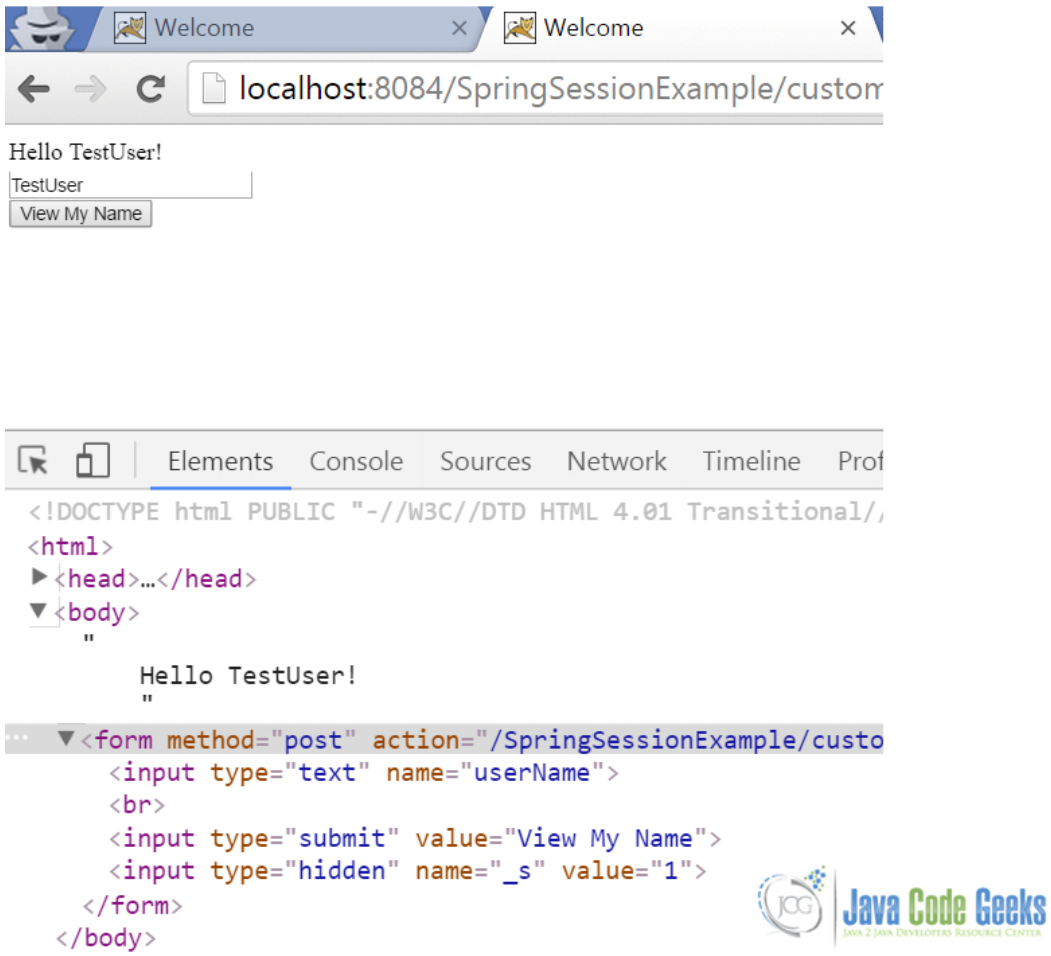

Figure 13.7: Session Two in another tab

The sessions are differentiated by their aliases. Here's how the browser stores them:

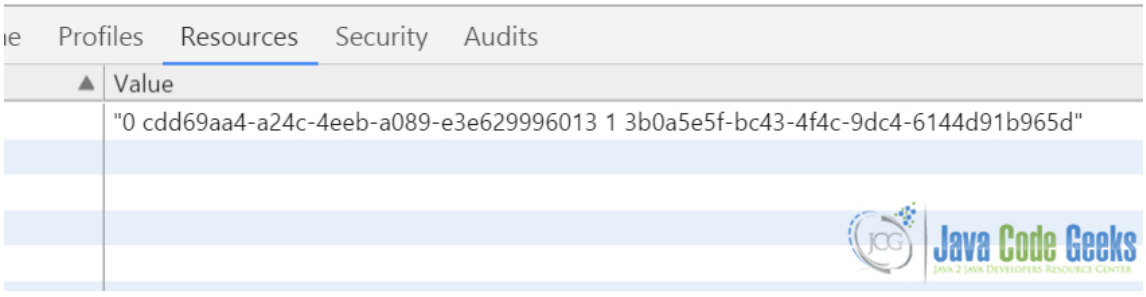

Figure 13.8: Cookies stored in Browser

Here's the how the output looks in the console, displaying the sessionIds map, with alias being the key and the session id being the value.

```
0 : cdd69aa4-a24c-4eeb-a089-e3e629996013
1 : 3b0a5e5f-bc43-4f4c-9dc4-6144d91b965d
```

13.4 Download The Source Code

Here, we demonstrated how we can use spring session to manage `HttpSession` for an uninterrupted user experience.

Download

You can download the source code of this example here: **SpringSessionExample.zip**

Chapter 14

Spring Web Flow Tutoriall

In of our previous examples, we have demonstrated how Spring MVC can be configured. In this example, we will demonstrate what is Spring Web-Flow and what are its benefits and how to configure it in a web-application.

14.1 Introduction

Spring MVC is a powerful framework that allows the user to configure and manage the flow of web-application in any possible way. However, sometimes the scenario may require be to have a more tight control over the flow of the application or to manage the possible ways to navigate through the application.

Spring Web-Flow helps in this kind of scenario by clearly defining the views and the transition between them. Web-Flow is itself based on top of `Spring MVC` and hence provides all the goodies of Spring MVC plus the added control over the transitions. Let's look at how we can configure the Web-Flow for our applications:

14.2 Project Set-Up

Let's start by first setting up the project in Eclipse or any other you have in mind. We shall use Maven to setup our project. Open Eclipse and create a simple Maven project and check the skip archetype selection checkbox on the dialogue box that appears. Replace the content of the existing `pom.xml` with the one provided below:

pom.xml

```
<project xmlns="https://maven.apache.org/POM/4.0.0" xmlns:xsi="https://www.w3.org/2001/ ↩
    XMLSchema-instance"
  xsi:schemaLocation="https://maven.apache.org/POM/4.0.0 https://maven.apache.org/maven- ↩
      v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.jcg.examples.springWebFlowExample</groupId>
  <artifactId>SpringWebFlowExample</artifactId>
  <packaging>war</packaging>
  <version>0.0.1-SNAPSHOT</version>
  <name>SpringWebFlowExample</name>
  <url>https://maven.apache.org</url>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
     <dependency>
```

```
        <groupId>org.springframework.webflow</groupId>
        <artifactId>spring-webflow</artifactId>
        <version>2.4.2.RELEASE</version>
    </dependency>
  </dependencies>
  <build>
    <finalName>SpringWebFlowExample</finalName>
  </build>
</project>
```

This will import the required JAR dependencies in the project. We can now start with the actual implementation of the Spring Web-Flow in our project.

Our application will be a simple login based application. Upon hitting the URL for the first time, the user will directed to a login page.

The user enters his credentials and clicks on the login button. If the password is correct the view transitions to success view or else the user is directed back to the login screen.

While this is a very basic scenario for the beginner users to understand, Spring Web-Flow is capable of handling many more complex scenarios.

14.3 Implementation

The implementation starts with the basic PoJo for login purposes which will hold the username and password.

LoginBean.java

```
package com.jcg.examples.bean;
import java.io.Serializable;

public class LoginBean implements Serializable
{
                /**
                 *
                 */
                private static final long serialVersionUID = 1L;

                private String userName;

                private String password;

                public String getUserName()
                {
                                return userName;
                }

                public void setUserName(String userName)
                {
                                this.userName = userName;
                }

                public String getPassword()
                {
                                return password;
                }

                public void setPassword(String password)
                {
                                this.password = password;
                }
```

```
                @Override
                public String toString()
                {
                        return "LoginBean [userName=" + userName + ", password=" +  ↩
                            password + "]";
                }

}
```

Next is the `Service` file which will authenticate the user. Based on the output of its `validateUser` method, web-flow will decide the view to be rendered. The Service class is marked with annotation to be picked up at run time by the Spring Bean Factory. For the sake of brevity, I have hard-coded the credentials in the source file itself.

LoginService.java

```
package com.jcg.examples.service;

import org.springframework.stereotype.Service;

import com.jcg.examples.bean.LoginBean;

@Service
public class LoginService
{
                public String validateUser(LoginBean loginBean)
                {
                        String userName = loginBean.getUserName();
                        String password = loginBean.getPassword();
                        if(userName.equals("Chandan") && password.equals(" ↩
                            TestPassword"))
                        {
                                return "true";
                        }
                        else
                        {
                                return "false";
                        }
                }

}
```

Now, we need to implement the define the flow. Flow is basically a cycle of events that will lead to completion of a single task in the context of the application. This cycle or flow will include multiple events and the user maybe made to navigate to and fro between various views, depending upon the choice he makes. Let's have a look at the xml that we have to use for our application for flow navigation:

book-search-flow.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<flow xmlns="https://www.springframework.org/schema/webflow"
        xmlns:xsi="https://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="https://www.springframework.org/schema/webflow
https://www.springframework.org/schema/webflow/spring-webflow-2.4.xsd">

        <var name="loginBean" class="com.jcg.examples.bean.LoginBean" />

        <view-state id="displayLoginView" view="jsp/login.jsp" model="loginBean">
                <transition on="performLogin" to="performLoginAction" />
        </view-state>

        <action-state id="performLoginAction">
```

```
                <evaluate expression="loginService.validateUser(loginBean)" />

                <transition on="true" to="displaySuccess" />
                <transition on="false" to="displayError" />

        </action-state>

        <view-state id="displaySuccess" view="jsp/success.jsp" model="loginBean"/>

        <view-state id="displayError" view="jsp/failure.jsp" />
</flow>
```

The first view in the flow becomes the default view and hence, is shown when the URL for that particular `Flow` is hit for the first time. Once the user submits the flow moves to `action` tag to determine dynamically which view should be rendered. The `action directive` in turn uses the backing Service Bean we created earlier.

Also the view may have a backing Bean as we have in Spring MVC, which is defined by the `model` attribute. The view contain two important variables which tell the container, the event that has occurred and the current state of the application. These variables are `_eventId` and `_flowExecutionKey`. When coding for the view, the developer should not forget to include these variables in the view code.

Now that the flow is ready we need to hook it up somewhere in the system, so that it can be picked up by the Spring Container.

`flow-definition.xml` file defines a `Flow-Executor` and a `Flow-Registry`. As the name indicates, the `Flow-Executor`, actually orchestrates the flow while it refers to `Flow-Registry` to determine the next action to be taken for the flow.

`FlowHandlerMapping` is responsible for creating the appropriate URLs for all the flows defined in the application. `FlowHandlerAdapter` encapsulates the actual flow and delegates the specific flows to be handled by the Spring Flow Controllers. We will include this file in the main spring configuration sheet so that our web-flow gets hooked into the main Spring Container and the requests are directed to the Flow Controllers.

flow-definition.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="https://www.springframework.org/schema/beans"
        xmlns:xsi="https://www.w3.org/2001/XMLSchema-instance" xmlns:flow="https://www. ↩
            springframework.org/schema/webflow-config"
        xsi:schemaLocation="https://www.springframework.org/schema/webflow-config
https://www.springframework.org/schema/webflow-config/spring-webflow-config-2.4.xsd
https://www.springframework.org/schema/beans
https://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

        <bean class="org.springframework.webflow.mvc.servlet.FlowHandlerMapping">
                <property name="flowRegistry" ref="bookSearchFlowRegistry" />
        </bean>

        <bean class="org.springframework.webflow.mvc.servlet.FlowHandlerAdapter">
                <property name="flowExecutor" ref="bookSearchFlowExecutor" />
        </bean>

        <flow:flow-executor id="bookSearchFlowExecutor" flow-registry=" ↩
            bookSearchFlowRegistry" />

        <flow:flow-registry id="bookSearchFlowRegistry">
                <flow:flow-location id="bookSearchFlow" path="/flows/book-search-flow.xml"  ↩
                    />
        </flow:flow-registry>

</beans>
```

`spring-config.xml` contains the basic information for the spring container for tasks like rendering the views, bean declarations, annotation scanning etc. It also includes the `flow-definition.xml` file for the container to load its contents.

spring-config.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="https://www.springframework.org/schema/beans"
        xmlns:context="https://www.springframework.org/schema/context"
        xmlns:xsi="https://www.w3.org/2001/XMLSchema-instance" xmlns:flow="https://www. ↩
            springframework.org/schema/webflow-config"
        xsi:schemaLocation="
        https://www.springframework.org/schema/webflow-config
        https://www.springframework.org/schema/webflow-config/spring-webflow-config-2.4.xsd
   https://www.springframework.org/schema/beans
   https://www.springframework.org/schema/beans/spring-beans-3.0.xsd
   https://www.springframework.org/schema/context
   https://www.springframework.org/schema/context/spring-context-3.0.xsd">

        <context:component-scan base-package="com.jcg.examples" />

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

        <import resource="flow-definition.xml"/>

</beans>
```

The `web.xml` is similar to any spring mvc application. It starts the Spring container with the above xml and directs all the reuqests to the `DispatcherServlet`.

web.xml

```
<!DOCTYPE web-app PUBLIC
 "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
 "https://java.sun.com/dtd/web-app_2_3.dtd" >

<web-app>
  <display-name>Spring-Flow Web-Application Example</display-name>

  <servlet>
                <servlet-name>springFlowApplication</servlet-name>
                <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet- ↩
                    class>
                <init-param>
                        <param-name>contextConfigLocation</param-name>
                        <param-value>classpath://spring-config.xml</param-value>
                </init-param>
                <load-on-startup>1</load-on-startup>
        </servlet>

         <servlet-mapping>
      <servlet-name>springFlowApplication</servlet-name>
      <url-pattern>/</url-pattern>
   </servlet-mapping>
</web-app>
```

Here's the default view of our Flow which is backed by a Spring bean. The name of the input boxes are same as the property names in the backing PoJo. In case the developer wants to name them separately, he may use the Spring tag library and the `path` attribute.

login.jsp

```
<%@ page isELIgnored="false"%>
```

```
<html>
<body>
        <h2>Please Login</h2>

        <form method="post" action="${flowExecutionUrl}">

                <input type="hidden" name="_eventId" value="performLogin">
                <input type="hidden" name="_flowExecutionKey" value="${flowExecutionKey}" / ↩
                    >

                <input type="text" name="userName" maxlength="40"><br>
                <input type="password" name="password" maxlength="40">
                <input type="submit" value="Login" />

        </form>

</body>
</html>
```

This is the view rendered upon successful authentication.

success.jsp

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
    <%@ page isELIgnored ="false" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "https://www.w3.org/TR/html4 ↩
    /loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Login Successful</title>
</head>
<body>
Welcome ${loginBean.userName}!!
</body>
</html>
```

When entering the wrong credentials, the user is notified via this view:

failure.jsp

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
    <%@ page isELIgnored ="false" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "https://www.w3.org/TR/html4 ↩
    /loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Login Successful</title>
</head>
<body>
Invalid username or password. Please try again!
</body>
</html>
```

Now, let's deploy and run the code. I have used Apache Tomcat 7 for this example. Here's the output for the first page:

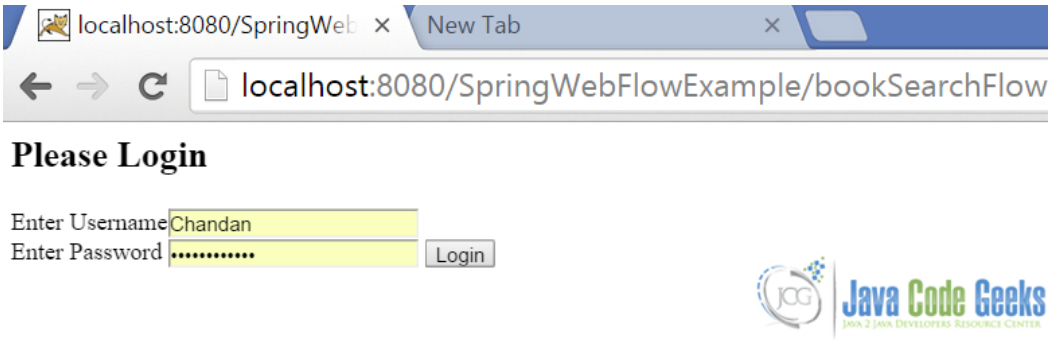

Figure 14.1: Login page

Upon successful authentication :

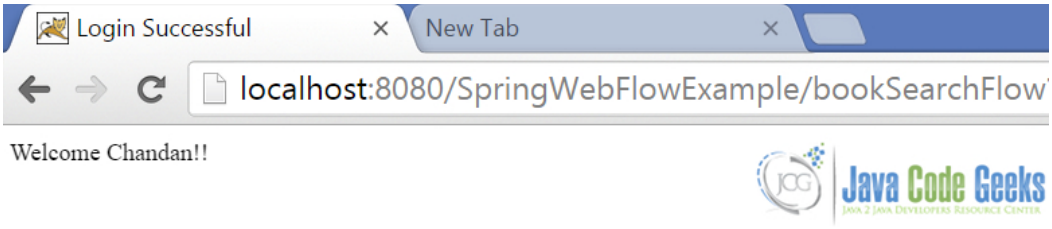

Figure 14.2: Successful login flow

Upon entering Invalid Credentials:

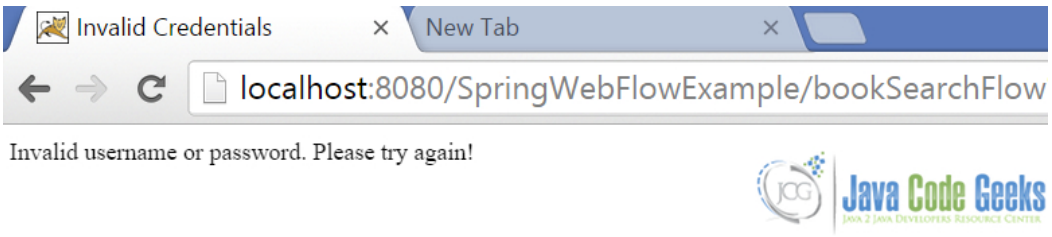

Figure 14.3: Failed Login Flow

14.4 Download The Source Code

In this example, we demonstrated how our view flows could be defined clearly and hence, managed easily and minutely using Spring Web-Flow.

Download

You can download the source code of this example here: **SpringWebFlowExample.zip**