



Module : Spring

Module Overview

In this module, students will be able to learn about the design patterns in java such as creational, structural, and behavioral design patterns.

Module Objective

After completion of this session, learners will be able to:

- understand the different design patterns that are the solutions for solving a specific problem/task.
- implement the different design patterns

Spring

Java EE

JEE or Jakarta Enterprise Edition, previously known as Java Platform, Enterprise Edition (Java EE) is a Java platform by Oracle. It offers a set of specifications, extending the features offered within Java Standard edition, SE, by including enterprise features for application development such as distributed computing, web services and Java microservices.

The core component of Java EE includes Enterprise Java Beans, EJBs, Java Server Pages, JSP, and Java Database Connectivity, JDBC. Java EE applications run on microservices or application servers that handle all the fundamental things such as transactions, security, scalability, concurrency and component management. These are the most essential things an application must handle properly and smoothly.

Limitations of JEE

It has a bit complex application development environment, which is difficult to understand for beginners.

The final cost of a project including development, deployment, and application development may end up being expensive.

Spring Framework

Spring is the application development framework for Java EE. It is an open-source Java platform that provides supports for developing robust and large-scale Java applications. It also offers tons of extensions that are used



for building all sorts of large-scale applications on top of the Java EE platform.

It is a lightweight framework that enables developers to develop enterprise-class applications using Plain Old Java Object, POJO. Although the Spring framework does not have any specific programming model, it has become popular among Java developers as an addition to the Java platform. It has a huge community of Java developers who are working and contributing to introducing more extensions and improving existing features offered by the Spring framework.

Advantages of Spring Framework

1) Predefined Templates

Spring framework provides templates for JDBC, Hibernate, JPA etc. technologies. So, there is no need to write too much code. It hides the basic steps of these technologies.

Let's take the example of JdbcTemplate, you don't need to write the code for exception handling, creating connection, creating statement, committing transaction, closing connection etc. You need to write the code of executing query only. Thus, it saves a lot of JDBC code.

2) Loose Coupling

The Spring applications are loosely coupled because of dependency injection.

3) Easy to test

The Dependency Injection makes easier to test the application. The EJB or Struts application require server to run the application but Spring framework doesn't require server.

4) Lightweight

Spring framework is lightweight because of its POJO implementation. The Spring Framework doesn't force the programmer to inherit any class or implement any interface. That is why it is said non-invasive.

5) Fast Development

The Dependency Injection feature of Spring Framework and its support to various frameworks makes the easy development of JavaEE application.

6) Powerful abstraction

It provides powerful abstraction to JavaEE specifications such as JMS, JDBC, JPA and JTA.

7) Declarative support

It provides declarative support for caching, validation, transactions and formatting.

- It is relatively complex to build on Spring as it lacks a clear focus.
- For a beginner Java developer, learning Spring framework might be challenging.

Disadvantages of Spring:

- You must have some knowledge of XML as a lot of XML is used in Spring.
- Clear guidelines on several topics are not present in Spring documentation.
- It takes a lot of time and effort in initial configurations.



Key differences between Java EE vs Spring

Java EE vs Spring discussion is usually based on the comparison of some very specific features. Both are quite popular choices among Java developers but they both have some prominent differences in their offerings in terms of features, services and cost.

Following are some of the major differences that spark the Java EE vs Spring debate:

Different architectures:

Java EE is based on a three-tier architecture. The first is the presentation tier that contains the user interface and focuses on delivering a fluid and stable experience. The second is the application tier that contains the main business logic of the application and the third is the data-tier, which comprises of the database and other data needs.

Spring, on the other hand, is based on a layered architecture that includes different modules. Each module delivers different features for the applications and all these modules are built on top of their core containers.

Languages supported:

Java EE uses a high-level object-oriented programming language. Java is used in Java EE that checks each described feature of a programming language.

Spring does not have any specific programming model. Developers have the option to use Java as well as Kotlin in Spring.

Structures:

Java EE can be used to develop either web-based or non-web-based structures, whereas Spring offers a variety of structures for your application ranging from Microservices, cloud, serverless event driver, web applications, etc.

Performance and speed:

Speed is something that is always brought up in Java EE vs Spring. In terms of speed, Java EE is the clear winner as Spring is a bit slower than Java EE in terms of performance and usability.

Cost of projects:

Cost is one of the primary differences between Java EE vs Spring. Java EE or Jakarta EE comes with an Oracle-based license, whereas Spring has an open-source license. It is completely free to use.



Standardization:

Java EE is a Sun/Oracle standard/specification. Java EE incorporates things like Object-Relational Mapping, Security, Web Applications, database availability, and exchanges.

Spring, then again, is not a standard. It is just a framework for Java EE itself. It is a structure offering lots of functions on the Java EE details, but in its frame.

How to download and install Spring framework

1. For Downloading Spring Repository you need to visit <https://repo.spring.io/release/org/springframework/spring/>. In this website, you will find different spring framework releases.

The screenshot shows a web browser window displaying the 'Index of release/org/springframework/spring' page. The page lists various Spring framework releases with columns for Name, Last Modified, Size, and Download Link. The releases are organized by version, starting from 1.0.0 and going up to 2.0.2. The latest release, 2.0.2, is highlighted in blue.

| Name | Last Modified | Size | Download Link |
|-----------|-------------------------|------|---------------|
| 1.0.0/ | 06-03-22 09:15:51 +0530 | | |
| 1.0.0-m4/ | 06-03-22 08:50:57 +0530 | | |
| 1.0-rc1/ | 06-03-22 17:48:24 +0530 | | |
| 1.0.1/ | 06-03-22 17:48:26 +0530 | | |
| 1.1/ | 06-03-22 12:41:32 +0530 | | |
| 1.1-rc1/ | 06-03-22 17:48:30 +0530 | | |
| 1.1-rc2/ | 06-03-22 14:50:31 +0530 | | |
| 1.1.1/ | 06-03-22 17:48:34 +0530 | | |
| 1.1.2/ | 06-03-22 17:48:36 +0530 | | |
| 1.1.3/ | 06-03-22 17:48:38 +0530 | | |
| 1.1.4/ | 06-03-22 17:48:40 +0530 | | |
| 1.1.5/ | 06-03-22 17:48:42 +0530 | | |
| 1.2/ | 06-03-22 17:48:45 +0530 | | |
| 1.2-rc1/ | 06-03-22 17:48:47 +0530 | | |
| 1.2-rc2/ | 06-03-22 17:48:49 +0530 | | |
| 1.2.1/ | 06-03-22 17:48:51 +0530 | | |
| 1.2.2/ | 06-03-22 17:48:54 +0530 | | |
| 1.2.3/ | 06-03-22 17:48:56 +0530 | | |
| 1.2.4/ | 06-03-22 11:33:17 +0530 | | |
| 1.2.5/ | 06-03-22 17:49:00 +0530 | | |
| 1.2.6/ | 06-03-22 17:49:03 +0530 | | |
| 1.2.7/ | 06-03-22 17:49:05 +0530 | | |
| 1.2.8/ | 06-03-22 17:49:07 +0530 | | |
| 1.2.9/ | 06-03-22 17:49:10 +0530 | | |
| 2.0/ | 06-03-22 17:49:13 +0530 | | |
| 2.0-m1/ | 06-03-22 17:49:16 +0530 | | |
| 2.0-m2/ | 06-03-22 17:49:18 +0530 | | |
| 2.0-m3/ | 06-03-22 17:49:20 +0530 | | |
| 2.0-m4/ | 06-03-22 17:49:21 +0530 | | |
| 2.0-m5/ | 06-03-22 17:49:23 +0530 | | |
| 2.0-rc1/ | 06-03-22 17:49:25 +0530 | | |
| 2.0-rc2/ | 06-03-22 17:49:27 +0530 | | |
| 2.0-rc3/ | 06-03-22 01:47:51 +0530 | | |
| 2.0.1/ | 06-03-22 17:49:31 +0530 | | |
| 2.0.2/ | 06-03-22 01:58:21 +0530 | | |

You have to click on latest framework release. Here you will find five files which are:



The screenshot shows a web browser window with the address bar displaying 'repo.spring.io/ui/native/release/org/springframework/spring/5.3.9/'. The page title is 'Index of release/org/springframework/spring/5.3.9'. Below the title is a table with four columns: Name, Last Modified, Size, and Download Link. The table lists several files for the Spring 5.3.9 release, including dist.zip, docs.zip, schema.zip, pom, and pom.asc. The 'Last Modified' column shows dates and times for each file. The 'Size' column shows the file sizes in MB or KB. The 'Download Link' column provides links to download each file. At the bottom of the table, there is a note: 'Artifactory: Online Server at localhost Port 8081'.

| Name | Last Modified | Size | Download Link |
|---|-------------------------|---------|---|
| spring-5.3.9-dist.zip | 14-07-21 12:19:17 +0530 | 78.6 MB | spring-5.3.9-dist.zip |
| spring-5.3.9-docs.zip | 14-07-21 12:19:15 +0530 | 35.2 MB | spring-5.3.9-docs.zip |
| spring-5.3.9-schema.zip | 14-07-21 12:19:13 +0530 | 61.3 KB | spring-5.3.9-schema.zip |
| spring-5.3.9.pom | 14-07-21 12:19:06 +0530 | 1.4 KB | spring-5.3.9.pom |
| spring-5.3.9.pom.asc | 14-07-21 12:47:35 +0530 | 488.0 B | spring-5.3.9.pom.asc |

Artifactory: Online Server at localhost Port 8081

Now you have to click on “spring-framework-5.3.9-dist.zip” so that it will download.

- For Installing, you need to extract the “spring-framework-5.3.9-dist.zip” file in your C/D drive. Now you are able to run your application in spring framework.

Features of the Spring Framework

The features of the Spring framework such as IoC, AOP, and transaction management, make it unique among the list of frameworks. Some of the most important features of the Spring framework are as follows:

- **IoC container:**
It refers to the core container that uses the DI or IoC pattern which implicitly provides an object reference in a class during runtime. This pattern acts as an alternative to the service locator pattern. The IoC container contains assembler code that handles the configuration management of application objects. The Spring framework provides two packages, namely `org.springframework.beans` and `org.springframework.context` which helps in providing the functionality of the IoC container.
- **Data access framework:**
This framework allows the developers to use persistence APIs, such as JDBC and Hibernate, for storing persistence data in database. It helps in solving various problems of the developer, such as how to interact with a database connection, how to make sure that the connection is closed, how to deal with exceptions, and how to implement transaction management. It also enables the developers to easily write code to access the persistence data throughout the application.
- **Spring MVC framework:**
Allows you to build Web applications based on MVC architecture. All the requests made by a user first go through the controller and are then dispatched to different views, that is, to different JSP pages or Servlets. The form handling and form validating features of the Spring MVC framework can be easily integrated with all popular view technologies such as JSP, Jasper Report, FreeMarker, and Velocity.
- **Transaction management:**
Helps in handling transaction management of an application without affecting its code. This framework provides Java Transaction API (JTA) for global transactions managed by an application server and local



transactions managed by using the JDBC Hibernate, Java Data Objects (JDO), or other data access APIs. It enables the developer to model a wide range of transactions on the basis of Spring's declarative and programmatic transaction management.

➤ **Spring Web Service:**

Generates Web service endpoints and definitions based on Java classes, but it is difficult to manage them in an application. To solve this problem, Spring Web Service provides layered-based approaches that are separately managed by Extensible Markup Language (XML) parsing (the technique of reading and manipulating XML).

Spring provides effective mapping for transmitting incoming XML message request to an object and the developer to easily distribute XML message (object) between two machines.

➤ **JDBC abstraction layer:**

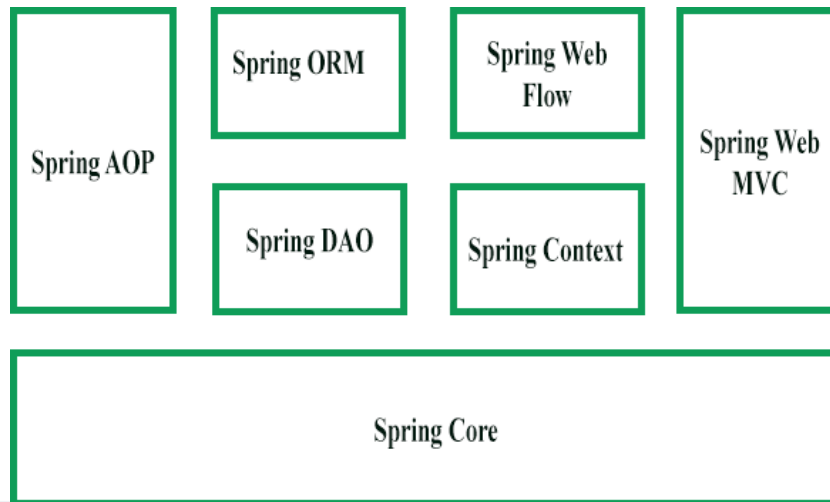
Helps the users in handling errors in an easy and efficient manner. The JDBC programming code can be reduced when this abstraction layer is implemented in a Web application. This layer handles exceptions such as DriverNotFound. All SQLExceptions are translated into the DataAccessException class. Spring's data access exception is not JDBC specific and hence Data Access Objects (DAO) are not bound to JDBC only.

➤ **Spring TestContext framework:**

Provides facilities of unit and integration testing for the Spring applications. Moreover, the Spring TestContext framework provides specific integration testing functionalities such as context management and caching DI of test fixtures, and transactional test management with default rollback semantics.

Evolution of Spring Framework

The Spring framework consists of seven modules which are shown in the Figure below.



These modules provide different platforms to develop different enterprise applications; for example, one can use Spring Web MVC module for developing MVC-based applications.

➤ **Spring Core Module:**

The Spring Core module, which is the core component of the Spring framework, provides the IoC container. There are two types of implementations of the Spring container, namely, bean factory and application context. Bean factory is defined using the `org.springframework.beans.factory.BeanFactory` interface and acts as a container for beans. The Bean factory container allows you to decouple the configuration and specification of dependencies from program logic.

In the Spring framework, the Bean factory acts as a central IoC container that is responsible for instantiating application objects. It also configures and assembles the dependencies between these objects. There are numerous implementations of the BeanFactory interface. The `XmlBeanFactory` class is the most common implementation of the BeanFactory interface.

➤ **Spring AOP Module:**

Similar to Object-Oriented Programming (OOP), which breaks down the applications into hierarchy of objects, AOP breaks down the programs into aspects or concerns. Spring AOP module allows you to implement concerns or aspects in a Spring application. In Spring AOP, the aspects are the regular Spring beans or regular classes annotated with `@Aspect` annotation.

These aspects help in transaction management and logging and failure monitoring of an application.

For example, transaction management is required in bank operations such as transferring an amount from one account to another. Spring AOP module provides a transaction management abstraction layer that can be applied to transaction APIs.



➤ **Spring ORM Module:**

The Spring ORM module is used for accessing data from databases in an application. It provides APIs for manipulating databases with JDO, Hibernate, and iBatis. Spring ORM supports DAO, which provides a convenient way to build the following DAOs-based ORM solutions:

- Simple declarative transaction management
- Transparent exception handling
- Thread-safe, lightweight template classes
- DAO support classes
- Resource management

➤ **Spring Web MVC Module:**

The Web MVC module of Spring implements the MVC architecture for creating Web applications. It separates the code of model and view components of a Web application. In Spring MVC, when a request is generated from the browser, it first goes to the DispatcherServlet class (Front Controller), which dispatches the request to a controller (SimpleFormController class or AbstractWizardFormController class) using a set of handler mappings.

The controller extracts and processes the information embedded in a request and sends the result to the DispatcherServlet class in the form of the model object. Finally, the DispatcherServlet class uses ViewResolver classes to send the results to a view, which displays these results to the users.

➤ **Spring Web Flow Module:**

The Spring Web Flow module is an extension of the Spring Web MVC module. Spring Web MVC framework provides form controllers, such as class SimpleFormController and AbstractWizardFormController class, to implement predefined workflow. The Spring Web Flow helps in defining XML file or Java Class that manages the workflow between different pages of a Web application. The Spring Web Flow is distributed separately and can be downloaded through <http://www.springframework.org> website.

The following are the advantages of Spring Web Flow:

- The flow between different UIs of the application is clearly provided by defining Web flow in XML file.
- Web flow definitions help you to virtually split an application in different modules and reuse these modules in multiple situations.
- Spring Web Flow lifecycle can be managed automatically

➤ **Spring Web DAO Module:**

The DAO package in the Spring framework provides DAO support by using data access technologies such as JDBC, Hibernate, or JDO. This module introduces a JDBC abstraction layer by eliminating the need for providing tedious JDBC coding. It also provides programmatic as well as declarative transaction management classes.

Spring DAO package supports heterogeneous Java Database Connectivity and O/R mapping, which helps Spring work with several data access technologies. For easy and quick access to database resources, the Spring framework provides abstract DAO base classes. Multiple implementations are available for each



data access technology supported by the Spring framework.

For example, in JDBC, the JdbcDaoSupport class and its methods are used to access the DataSource instance and a preconfigured JdbcTemplate instance. You need to simply extend the JdbcDaoSupport class and provide a mapping to the actual DataSource instance in an application context configuration to access a DAO-based application.

➤ **Spring Application Context Module:**

The Spring Application context module is based on the Core module. Application context `org.springframework.context.ApplicationContext` is an interface of `BeanFactory`. This module derives its feature from the `org.springframework.beans` package and also supports functionalities such as internationalization (I18N), validation, event propagation, and resource loading.

The Application context implements `MessageSource` interface and provides the messaging functionality to an application.

Now let us practise Spring program:

You need to create spring configuration file and declare all of your beans here. I am naming the configuration file as `applicationContext.xml` file.

```
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">
  <bean id="springTest" class="com.java2novice.beans.SpringFirstTest" />
</beans>
```

Here is the Spring Demo class to run the spring bean:



```
package com.java2novice.test;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import com.java2novice.beans.SpringFirstTest;

public class SpringDemo {
    public static void main(String a[]){
        String confFile = "applicationContext.xml";
        ApplicationContext context = new
        ClassPathXmlApplicationContext(confFile);
        SpringFirstTest sft = (SpringFirstTest) context.getBean("springTest");
        sft.testMe();
    } }
```

Output:

I am listening...

Coupling in Java

In object-oriented design, Coupling refers to the degree of direct knowledge that one element has of another. In other words, how often do changes in class A force related changes in class B.

There are two types of coupling:

- **Tight coupling:** In general, Tight coupling means the two classes often change together. In other words, if A knows more than it should about the way in which B was implemented, then A and B are tightly coupled.

Loose Coupling: One of the most key aspects of a Java project is loose coupling. The loose coupling in Java shows how to achieve loose coupling in Java projects or programs. The more loosely coupled structures present in the project or program, the better it is. In loose coupling, a method or class is almost independent, and they have less depended on each other. In other words, the more knowledge one class or method has about another class or method, the more tightly coupled structure is developed. If the classes or methods know less about each other, the more loosely coupled structure comes into existence.

Beans

In Spring, the objects that form the backbone of your application and that are managed by the Spring IoC



containers are called beans. A bean is an object that is instantiated, assembled, and otherwise managed by a Spring IoC container.

Inversion of Control

Simply put, Inversion of Control (IoC) is a process in which an object defines its dependencies without creating them. This object delegates the job of constructing such dependencies to an IoC container.

Let's start with the declaration of a couple of domain classes before diving into IoC.

Domain Classes

Assume we have a class declaration:

```
public class Company {  
    private Address address;  
  
    public Company(Address address) {  
        this.address = address;  
    }  
  
    // getter, setter and other properties  
}
```

This class needs a collaborator of type Address:

```
public class Address {  
    private String street;  
    private int number;  
  
    public Address(String street, int number) {  
        this.street = street;  
        this.number = number;  
    }  
  
    // getters and setters  
}
```

Normally, we create objects with their classes' constructors:

```
Address address = new Address("High Street", 1000);
Company company = new Company(address);
```

There's nothing wrong with this approach, but wouldn't it be nice to manage the dependencies in a better way?

Imagine an application with dozens or even hundreds of classes. Sometimes we want to share a single instance of a class across the whole application, other times we need a separate object for each use case, and so on.

Managing such a number of objects can be very tedious. This is where inversion of control comes to the rescue. Instead of constructing dependencies by itself, an object can retrieve its dependencies from an IoC container. All we need to do is to provide the container with appropriate configuration metadata.

Bean Configuration

First off, let's decorate the Company class with the @Component annotation:

```
@Component
public class Company {
    // this body is the same as before
}
```

Here's a configuration class supplying bean metadata to an IoC container:

```
@Configuration
@ComponentScan(basePackageClasses = Company.class)
public class Config {
    @Bean
    public Address getAddress() {
        return new Address("High Street", 1000);
    }
}
```

The configuration class produces a bean of type Address. It also carries the @ComponentScan annotation, which instructs the container to look for beans in the package containing the Company class.

When a Spring IoC container constructs objects of those types, all the objects are called Spring beans, as they are managed by the IoC container.



Design Patterns

Design patterns are an essential part of software development. These solutions not only solve recurring problems but also help developers understand the design of a framework by recognizing common patterns.

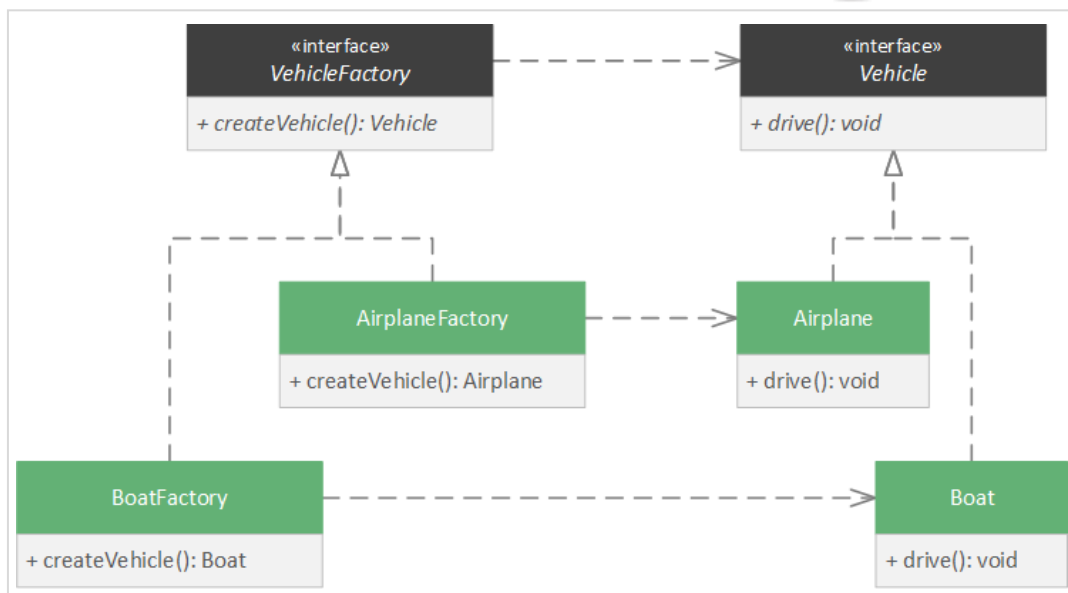
In this tutorial, we'll look at four of the most common design patterns used in the Spring Framework:

- Singleton pattern
- Factory Method pattern
- Proxy pattern
- Template pattern

Factory pattern

The factory method pattern entails a factory class with an abstract method for creating the desired object. Often, we want to create different objects based on a particular context.

For example, our application may require a vehicle object. In a nautical environment, we want to create boats, but in an aerospace environment, we want to create airplanes:



Spring uses this technique at the root of its Dependency Injection (DI) framework. Fundamentally, Spring treats a bean container as a factory that produces beans. Thus, Spring defines the **BeanFactory** interface as an abstraction of a bean container:



```
public interface BeanFactory {

    getBean(Class<T> requiredType);
    getBean(Class<T> requiredType, Object... args);
    getBean(String name);

    // ...
}
```

Configuration metadata

Spring configuration metadata needs to be created to tell Spring container how to initiate, configure, wire and assemble the application specific objects.

Since Spring's first release in 2002 to the latest release, spring has provided three ways of configurations:

- **XML-based Configuration:** In Spring Framework, the dependencies and the services needed by beans are specified in configuration files which are in XML format. These configuration files usually contain a lot of bean definitions and application-specific configuration options. They generally start with a bean tag. For example:

```
<bean id="studentbean" class="org.edubridge.firstSpring.StudentBean">
<property name="name" value="Edubridge"></property>
</bean>
```

- **Annotation-based configuration:** Instead of using XML to describe a bean wiring, you can configure the bean into the component class itself by using annotations on the relevant class, method, or field declaration. By default, annotation wiring is not turned on in the Spring container. So, you need to enable it in your Spring configuration file before using it. For example:

```
<beans>
<context:annotation-config/>
<!-- bean definitions go here -->
</beans>
```

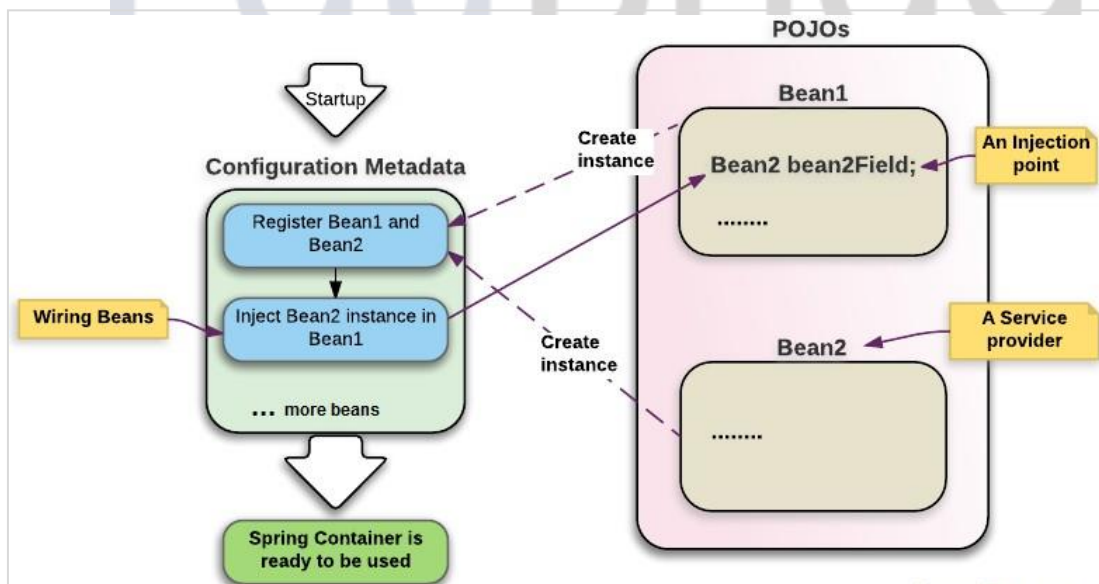
- **Java-based configuration (JavaConfig):** A pure-Java means of configuring container was provided. We don't need any XML with this method of configuration. JavaConfig provides a truly object-oriented mechanism for dependency injection, meaning we can take full advantage of reusability, inheritance and polymorphism in the configuration code. Application developer has complete

control over instantiation and dependency injection here. The key features in Spring Framework's new Java-configuration support are @Configuration annotated classes and @Bean annotated methods.

1. @Bean annotation plays the same role as the <bean/> element.
2. @Configuration classes allow define inter-bean dependencies by simply calling other @Bean methods in the same class.

For example:

```
@Configuration
public class StudentConfig
{
    @Bean
    public StudentBean myStudent()
    { return new StudentBean(); }
}
```



Regardless of what method we use, we mainly have to use configuration metadata at three places:

Beans: The objects managed by Spring Container. They are registered to the Spring container by the use of some metadata.

Injection Points: The places where dependencies have to be injected. The Injection Points typically are fields/setters/constructors in a Spring bean class. Spring framework populates/inserts the injection points with the required instances of other beans. That happens during the bean loading time.



The Configuration: This can be a Java class annotated with @Configuration or it can be XML if we are using old way of configuration. This is where we wire the injection points with dependencies.

Dependency Injection in Spring

Dependency Injection (DI) is a design pattern that removes the dependency from the programming code so that it can be easy to manage and test the application. Dependency Injection makes our programming code loosely coupled.

Every Java-based application has a few objects that work together to present what the end-user sees as a working application. When writing a complex Java application, application classes should be as independent as possible of other Java classes to increase the possibility to reuse these classes and to test them independently of other classes while unit testing. Dependency Injection (or sometime called wiring) helps in gluing these classes together and at the same time keeping them independent.

Consider you have an application which has a text editor component and you want to provide a spell check. Your standard code would look something like this –

```
public class TextEditor {  
    private SpellChecker spellChecker;  
  
    public TextEditor() {  
        spellChecker = new SpellChecker();  
    }  
}
```

What we've done here is, create a dependency between the TextEditor and the SpellChecker. In an inversion of control scenario, we would instead do something like this –

```
public class TextEditor {  
    private SpellChecker spellChecker;  
  
    public TextEditor(SpellChecker spellChecker) {  
        this.spellChecker = spellChecker;  
    }  
}
```

Here, the TextEditor should not worry about SpellChecker implementation. The SpellChecker will be implemented independently and will be provided to the TextEditor at the time of TextEditor instantiation. This entire procedure is controlled by the Spring Framework.

Here, we have removed total control from the TextEditor and kept it somewhere else (i.e. XML configuration file) and the dependency (i.e. class SpellChecker) is being injected into the class TextEditor through a Class Constructor. Thus the flow



of control has been "inverted" by Dependency Injection (DI) because you have effectively delegated dependences to some external system.

The second method of injecting dependency is through Setter Methods of the `TextEditor` class where we will create a `SpellChecker` instance. This instance will be used to call setter methods to initialize `TextEditor`'s properties.

Thus, DI exists in two major variants and the following two sub-chapters will cover both of them with examples –

| Sr.No. | Dependency Injection Type & Description |
|--------|---|
| 1 | Constructor-based dependency injection Constructor-based DI is accomplished when the container invokes a class constructor with a number of arguments, each representing a dependency on the other class. |
| 2 | Setter-based dependency injection Setter-based DI is accomplished by the container calling setter methods on your beans after invoking a no-argument constructor or no-argument static factory method to instantiate your bean. |

Setter Injection

Setter-based DI is accomplished by the container calling setter methods on your beans after invoking a no-argument constructor or no-argument static factory method to instantiate your bean.

Example

The following example shows a class `TextEditor` that can only be dependency-injected using pure setter-based injection.

Let us have a working Eclipse IDE in place and take the following steps to create a Spring application –

| Steps | Description |
|-------|---|
| 1 | Create a project with a name <code>SpringExample</code> and create a package <code>com.edubridge</code> under the <code>src</code> folder in the created project. |
| 2 | Add required Spring libraries using Add External JARs option as explained in the Spring Hello World Example chapter. |
| 3 | Create Java classes <code>TextEditor</code> , <code>SpellChecker</code> and <code>MainApp</code> under the <code>com.edubridge</code> package. |
| 4 | Create Beans configuration file <code>Beans.xml</code> under the <code>src</code> folder. |



- | | |
|---|---|
| 5 | The final step is to create the content of all the Java files and Bean Configuration file and run the application as explained below. |
|---|---|

Here is the content of `TextEditor.java` file –

```
package com.edubridge;

public class TextEditor {
    private SpellChecker spellChecker;

    // a setter method to inject the dependency.
    public void setSpellChecker(SpellChecker spellChecker) {
        System.out.println("Inside setSpellChecker." );
        this.spellChecker = spellChecker;
    }
    // a getter method to return spellChecker
    public SpellChecker getSpellChecker() {
        return spellChecker;
    }
    public void spellCheck() {
        spellChecker.checkSpelling();
    }
}
```

Here you need to check the naming convention of the setter methods. To set a variable `spellChecker` we are using `setSpellChecker()` method which is very similar to Java POJO classes. Let us create the content of another dependent class file `SpellChecker.java` –

```
package com.edubridge;

public class SpellChecker {
    public SpellChecker(){
        System.out.println("Inside SpellChecker constructor." );
    }
    public void checkSpelling() {
        System.out.println("Inside checkSpelling." );
    }
}
```

Following is the content of the `MainApp.java` file –



```
package com.edubridge;

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

public class MainApp {
    public static void main(String[] args) {
        ApplicationContext context = new ClassPathXmlApplicationContext("Beans.xml");

        TextEditor te = (TextEditor) context.getBean("textEditor");
        te.spellCheck();
    }
}
```

Following is the configuration file Beans.xml which has configuration for the setter-based injection –

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

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

    <!-- Definition for textEditor bean -->
    <bean id = "textEditor" class = "com.edubridge.TextEditor">
        <property name = "spellChecker" ref = "spellChecker"/>
    </bean>

    <!-- Definition for spellChecker bean -->
    <bean id = "spellChecker" class = "com.edubridge.SpellChecker"></bean>

</beans>
```

You should note the difference in Beans.xml file defined in the constructor-based injection and the setter-based injection. The only difference is inside the <bean> element where we have used <constructor-arg> tags for constructor-based injection and <property> tags for setter-based injection.

The second important point to note is that in case you are passing a reference to an object, you need to use ref attribute of <property> tag and if you are passing a value directly then you should use value attribute.

Once you are done creating the source and bean configuration files, let us run the application. If everything is fine with your application, this will print the following message –

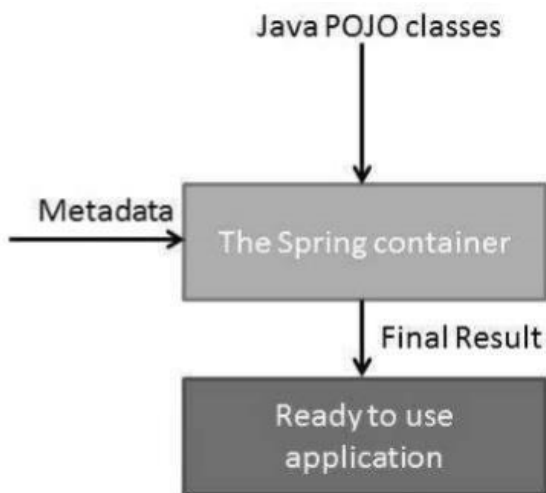


Inside SpellChecker constructor.
Inside setSpellChecker.
Inside checkSpelling.

Spring Container

The Spring container is at the core of the Spring Framework. The container will create the objects, wire them together, configure them, and manage their complete life cycle from creation till destruction. The Spring container uses DI to manage the components that make up an application. These objects are called Spring Beans, which we will discuss in the next chapter.

The container gets its instructions on what objects to instantiate, configure, and assemble by reading the configuration metadata provided. The configuration metadata can be represented either by XML, Java annotations, or Java code. The following diagram represents a high-level view of how Spring works. The Spring IoC container makes use of Java POJO classes and configuration metadata to produce a fully configured and executable system or application.



Spring provides the following two distinct types of containers.

1. Spring BeanFactory Container

This is the simplest container providing the basic support for DI and defined by the `org.springframework.beans.factory.BeanFactory` interface. The BeanFactory and related interfaces, such as



BeanFactoryAware, InitializingBean, DisposableBean, are still present in Spring for the purpose of backward compatibility with a large number of third-party frameworks that integrate with Spring.

There are a number of implementations of the BeanFactory interface that are come straight out-of-the-box with Spring. The most commonly used BeanFactory implementation is the XmlBeanFactory class. This container reads the configuration metadata from an XML file and uses it to create a fully configured system or application.

The BeanFactory is usually preferred where the resources are limited like mobile devices or applet-based applications. Thus, use an ApplicationContext unless you have a good reason for not doing so.

Example

Let us take a look at a working Eclipse IDE in place and take the following steps to create a Spring application –

| Steps | Description |
|-------|---|
| 1 | Create a project with a name SpringExample and create a package com.edubridge under the src folder in the created project. |
| 2 | Add the required Spring libraries using Add External JARs option as explained in the Spring Hello World Example chapter. |
| 3 | Create Java classes HelloWorld and MainApp under the com.edubridge package. |
| 4 | Create Beans configuration file Beans.xml under the src folder. |
| 5 | The final step is to create the content of all the Java files and Bean Configuration file. Finally, run the application as explained below. |

Here is the content of HelloWorld.java file –



```
package com.edubridge;

public class HelloWorld {
    private String message;

    public void setMessage(String message){
        this.message = message;
    }
    public void getMessage(){
        System.out.println("Your Message : " + message);
    }
}
```

Following is the content of the second file MainApp.java

```
package com.edubridge;

import org.springframework.beans.factory.InitializingBean;
import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.core.io.ClassPathResource;

public class MainApp {
    public static void main(String[] args) {
        XmlBeanFactory factory = new XmlBeanFactory (new ClassPathResource("Beans.xml"));
        HelloWorld obj = (HelloWorld) factory.getBean("helloWorld");
        obj.getMessage();
    }
}
```

Following two important points should be noted about the main program –

- The first step is to create a factory object where we used the framework API `XmlBeanFactory()` to create the factory bean and `ClassPathResource()` API to load the bean configuration file available in CLASSPATH. The `XmlBeanFactory()` API takes care of creating and initializing all the objects, i.e. beans mentioned in the configuration file.
- The second step is used to get the required bean using `getBean()` method of the created bean factory object. This method uses bean ID to return a generic object, which finally can be casted to the actual object. Once you have the object, you can use this object to call any class method.



Following is the content of the bean configuration file Beans.xml

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

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

  <bean id = "helloWorld" class = "com.edubridge.HelloWorld">
    <property name = "message" value = "Hello World!"/>
  </bean>

</beans>
```

Once you are done with creating the source and the bean configuration files, let us run the application. If everything is fine with your application, it will print the following message –

Your Message: Hello World!

2. Spring ApplicationContext Container

The Application Context is Spring's advanced container. Similar to BeanFactory, it can load bean definitions, wire beans together, and dispense beans upon request. Additionally, it adds more enterprise-specific functionality such as the ability to resolve textual messages from a properties file and the ability to publish application events to interested event listeners. This container is defined by `org.springframework.context.ApplicationContext` interface.

The ApplicationContext includes all functionality of the BeanFactory, It is generally recommended over BeanFactory. BeanFactory can still be used for lightweight applications like mobile devices or applet-based applications.

The most commonly used ApplicationContext implementations are –

- **FileSystemXmlApplicationContext** – This container loads the definitions of the beans from an XML file. Here you need to provide the full path of the XML bean configuration file to the constructor.
- **ClassPathXmlApplicationContext** – This container loads the definitions of the beans from an XML file. Here you do not need to provide the full path of the XML file but you need to set CLASSPATH properly because this container will look like bean configuration XML file in CLASSPATH.
- **WebXmlApplicationContext** – This container loads the XML file with definitions of all beans from within a web



application.

We already have seen an example on `ClassPathXmlApplicationContext` container in Spring Hello World Example, and we will talk more about `XmlWebApplicationContext` in a separate chapter when we will discuss web-based Spring applications. So let us see one example on `FileSystemXmlApplicationContext`.

Example

Let us have a working Eclipse IDE in place and take the following steps to create a Spring application –

| Steps | Description |
|-------|---|
| 1 | Create a project with a name <code>SpringExample</code> and create a package <code>com.edubridge</code> under the <code>src</code> folder in the created project. |
| 2 | Add required Spring libraries using <code>Add External JARs</code> option as explained in the Spring Hello World Example chapter. |
| 3 | Create Java classes <code>HelloWorld</code> and <code>MainApp</code> under the <code>com.edubridge</code> package. |
| 4 | Create Beans configuration file <code>Beans.xml</code> under the <code>src</code> folder. |
| 5 | The final step is to create the content of all the Java files and Bean Configuration file and run the application as explained below. |

Here is the content of `HelloWorld.java` file –

```
package com.edubridge;

public class HelloWorld {
    private String message;

    public void setMessage(String message){
        this.message = message;
    }
    public void getMessage(){
        System.out.println("Your Message : " + message);
    }
}
```

Following is the content of the second file `MainApp.java` –



```
package com.edubridge;

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

public class MainApp {
    public static void main(String[] args) {
        ApplicationContext context = new FileSystemXmlApplicationContext
            ("C:/Users/ZARA/workspace/HelloSpring/src/Beans.xml");

        HelloWorld obj = (HelloWorld) context.getBean("helloWorld");
        obj.getMessage();
    }
}
```

Following two important points should be noted about the main program –

- The first step is to create factory object where we used framework `APIFileSystemXmlApplicationContext` to create the factory bean after loading the bean configuration file from the given path. The `FileSystemXmlApplicationContext()` API takes care of creating and initializing all the objects i.e. beans mentioned in the XML bean configuration file.
- The second step is used to get the required bean using `getBean()` method of the created context. This method uses bean ID to return a generic object, which finally can be casted to the actual object. Once you have an object, you can use this object to call any class method.

Following is the content of the bean configuration file `Beans.xml`

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

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

    <bean id = "helloWorld" class = "com.edubridge.HelloWorld">
        <property name = "message" value = "Hello World!"/>
    </bean>

</beans>
```



Once you are done with creating the source and bean configuration files, let us run the application. If everything is fine with your application, it will print the following message –

Your Message: Hello World!

Autowiring in Spring

Autowiring feature of spring framework enables you to inject the object dependency implicitly. It internally uses setter or constructor injection.

Autowiring can't be used to inject primitive and string values. It works with reference only.

Advantage of Autowiring

It requires the less code because we don't need to write the code to inject the dependency explicitly.

Disadvantage of Autowiring

No control of programmer.

It can't be used for primitive and string values.

Autowiring Modes

There are many autowiring modes:

| No. | Mode | Description |
|-----|-------------|---|
| 1) | no | It is the default autowiring mode. It means no autowiring by default. |
| 2) | byName | The byName mode injects the object dependency according to name of the bean. In such case, property name and bean name must be same. It internally calls setter method. |
| 3) | byType | The byType mode injects the object dependency according to type. So property name and bean name can be different. It internally calls setter method. |
| 4) | constructor | The constructor mode injects the dependency by calling the constructor of the class. It calls the constructor having large number of parameters. |
| 5) | autodetect | It is deprecated since Spring 3. |



Example of Autowiring

Let's see the simple code to use autowiring in spring. You need to use autowire attribute of bean element to apply the autowire modes.

```
<bean id="a" class="org.sssit.A" autowire="byName"></bean>
```

Let's see the full example of autowiring in spring. To create this example, we have created 4 files.

1. B.java

This class contains a constructor and method only.

```
package org.sssit;
public class B {
    B(){System.out.println("b is created");}
    void print(){System.out.println("hello b");}
}
```

2. A.java

This class contains reference of B class and constructor and method.

```
package org.sssit;
public class A {
    B b;
    A(){System.out.println("a is created");}
    public B getB() {
        return b;
    }
    public void setB(B b) {
        this.b = b;
    }
    void print(){System.out.println("hello a");}
    void display(){
        print();
        b.print();
    }
}
```

3. applicationContext.xml



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

  <bean id="b" class="org.sssit.B"></bean>
  <bean id="a" class="org.sssit.A" autowire="byName"></bean>

</beans>
```

4. Test.java

This class gets the bean from the applicationContext.xml file and calls the display method.

```
package org.sssit;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
public class Test {
  public static void main(String[] args) {
    ApplicationContext context=new ClassPathXmlApplicationContext("applicationContext.xml");
    A a=context.getBean("a",A.class);
    a.display();
  }
}
```

Output:

```
b is created
a is created
hello a
hello b
```

1) byName autowiring mode



In case of byName autowiring mode, bean id and reference name must be same.

It internally uses setter injection.

```
<bean id="b" class="org.sssit.B"></bean>
<bean id="a" class="org.sssit.A" autowire="byName"></bean>
```

But, if you change the name of bean, it will not inject the dependency.

Let's see the code where we are f5 changing the name of the bean from b to b1.

```
<bean id="b1" class="org.sssit.B"></bean>
<bean id="a" class="org.sssit.A" autowire="byName"></bean>
```

2) byType autowiring mode

In case of byType autowiring mode, bean id and reference name may be different. But there must be only one bean of a type.

```
<bean id="b1" class="org.sssit.B"></bean>
<bean id="a" class="org.sssit.A" autowire="byName"></bean>
```

In this case, it works fine because you have created an instance of B type. It doesn't matter that you have different bean name than reference name.

But, if you have multiple bean of one type, it will not work and throw exception.

Let's see the code where are many bean of type B.

```
<bean id="b1" class="org.sssit.B"></bean>
<bean id="b2" class="org.sssit.B"></bean>
<bean id="a" class="org.sssit.A" autowire="byName"></bean>
```

In such case, it will throw exception.

3) constructor autowiring mode

In case of constructor autowiring mode, spring container injects the dependency by highest parameterized constructor.

If you have 3 constructors in a class, zero-arg, one-arg and two-arg then injection will be performed by calling the two-arg constructor.

```
<bean id="b" class="org.sssit.B"></bean>
<bean id="a" class="org.sssit.A" autowire="constructor"></bean>
```

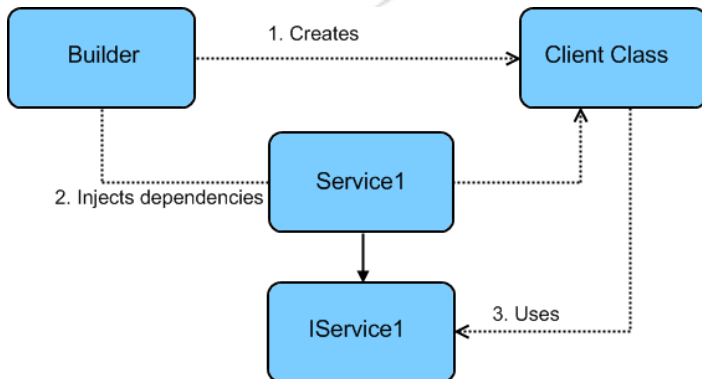
4) no autowiring mode

In case of no autowiring mode, spring container doesn't inject the dependency by auto wiring.

```
<bean id="b" class="org.sssit.B"></bean>
<bean id="a" class="org.sssit.A" autowire="no"></bean>
```

Dependency Injection

Every Java-based application has a few objects that work together to present what the end-user sees as a working application. When writing a complex Java application, application classes should be as independent as possible of other Java classes to increase the possibility to reuse these classes and to test them independently of other classes while unit testing. Dependency Injection (or sometime called wiring) helps in gluing these classes together and at the same time keeping them independent.



Dependency Injection by Constructor Example

We can inject the dependency by constructor. The `<constructor-arg>` sub element of `<bean>` is used for



constructor injection. Here we are going to inject.

- 1) primitive and String-based values
- 2) Dependent object (contained object)
- 3) Collection values etc.

Injecting primitive and string-based values

Let's see the simple example to inject primitive and string-based values. We have created three files here:

1) Employee.java

It is a simple class containing two fields id and name. There are four constructors and one method in this class.

```
package com.edubridge;

public class Employee {
    private int id;
    private String name;

    public Employee() {System.out.println("def cons");}

    public Employee(int id) {this.id = id;}

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

    public Employee(int id, String name) {
        this.id = id;
        this.name = name;
    }

    void show(){
        System.out.println(id+" "+name);
    } }
}
```

2) applicationContext.xml

We are providing the information into the bean by this file. The constructor-arg element invokes the constructor. In such case, parameterized constructor of int type will be invoked. The value attribute of constructor-arg element will assign the specified value. The type attribute specifies that int parameter constructor will be invoked.



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

  <bean id="e" class="com.edubridge.Employee">
    <constructor-arg value="10" type="int"></constructor-arg>
  </bean>
</beans>
```

3) Test.java

This class gets the bean from the applicationContext.xml file and calls the show method.

```
package com.edubridge;

import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.core.io.*;

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

    Resource r=new ClassPathResource("applicationContext.xml");
    BeanFactory factory=new XmlBeanFactory(r);

    Employee s=(Employee)factory.getBean("e");
    s.show();

  }
}
```

Output:10 null

Injecting string-based values

If you don't specify the type attribute in the constructor-arg element, by default string type constructor will be



invoked.

```
....
<bean id="e" class="com.edubridge.Employee">
<constructor-arg value="10"></constructor-arg>
</bean>
....
```

If you change the bean element as given above, string parameter constructor will be invoked and the output will be 010.

Output:0 10

You may also pass the string literal as following:

```
....
<bean id="e" class="com.edubridge.Employee">
<constructor-arg value="Sonoo"></constructor-arg>
</bean>
....
```

Output:0 Sonoo

Constructor Injection with Dependent Object

If there is HAS-A relationship between the classes, we create the instance of dependent object (contained object) first then pass it as an argument of the main class constructor. Here, our scenario is Employee HAS-A Address. The Address class object will be termed as the dependent object. Let's see the Address class first:

Address.java

This class contains three properties, one constructor and toString() method to return the values of these object.



```
package com.edubridge;

public class Address {
    private String city;
    private String state;
    private String country;

    public Address(String city, String state, String country) {
        super();
        this.city = city;
        this.state = state;
        this.country = country;
    }

    public String toString(){
        return city+" "+state+" "+country;
    }
}
```

Employee.java

It contains three properties id, name and address (dependent object), two constructors and show() method to show the records of the current object including the dependent object.



```
package com.edubridge;

public class Employee {
    private int id;
    private String name;
    private Address address;//Aggregation

    public Employee() {System.out.println("def cons");}

    public Employee(int id, String name, Address address) {
        super();
        this.id = id;
        this.name = name;
        this.address = address;
    }
    void show(){
        System.out.println(id+" "+name);
        System.out.println(address.toString());
    }
}
```

applicationContext.xml

The ref attribute is used to define the reference of another object, such way we are passing the dependent object as constructor argument.



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

  <bean id="a1" class="com.edubridge.Address">
    <constructor-arg value="ghaziabad"></constructor-arg>
    <constructor-arg value="UP"></constructor-arg>
    <constructor-arg value="India"></constructor-arg>
  </bean>

  <bean id="e" class="com.edubridge.Employee">
    <constructor-arg value="12" type="int"></constructor-arg>
    <constructor-arg value="Sonoo"></constructor-arg>
    <constructor-arg>
      <ref bean="a1"/>
    </constructor-arg>
  </bean>
</beans>
```

Test.java

This class gets the bean from the applicationContext.xml file and calls the show method.



```
package com.edubridge;

import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.core.io.*;

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

        Resource r=new ClassPathResource("applicationContext.xml");
        BeanFactory factory=new XmlBeanFactory(r);

        Employee s=(Employee)factory.getBean("e");
        s.show();

    }
}
```

Constructor Injection with Collection Example

We can inject collection values by constructor in spring framework. There can be used three elements inside the constructor-arg element.

It can be:

- list
- set
- map

Each collection can have string based and non-string based values.

In this example, we are taking the example of Forum where One question can have multiple answers. There are three pages:

1. Question.java
2. applicationContext.xml
3. Test.java

In this example, we are using list that can have duplicate elements, you may use set that have only unique elements. But, you need to change list to set in the applicationContext.xml file and List to Set in the Question.java file.



Question.java

This class contains three properties, two constructors and displayInfo() method that prints the information. Here, we are using List to contain the multiple answers.

```
package com.edubridge;

import java.util.Iterator;
import java.util.List;

public class Question {
    private int id;
    private String name;
    private List<String> answers;

    public Question() {}
    public Question(int id, String name, List<String> answers) {
        super();
        this.id = id;
        this.name = name;
        this.answers = answers;
    }

    public void displayInfo(){
        System.out.println(id+" "+name);
        System.out.println("answers are:");
        Iterator<String> itr=answers.iterator();
        while(itr.hasNext()){
            System.out.println(itr.next());
        }
    }
}
```

applicationContext.xml

The list element of constructor-arg is used here to define the list.



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

  <bean id="q" class="com.edubridge.Question">
    <constructor-arg value="111"></constructor-arg>
    <constructor-arg value="What is java?"></constructor-arg>
    <constructor-arg>
      <list>
        <value>Java is a programming language</value>
        <value>Java is a Platform</value>
        <value>Java is an Island of Indonasia</value>
      </list>
    </constructor-arg>
  </bean>

</beans>
```

Test.java

This class gets the bean from the applicationContext.xml file and calls the displayInfo method.



```
package com.edubridge;

import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.core.io.ClassPathResource;
import org.springframework.core.io.Resource;

public class Test {
    public static void main(String[] args) {
        Resource r=new ClassPathResource("applicationContext.xml");
        BeanFactory factory=new XmlBeanFactory(r);

        Question q=(Question)factory.getBean("q");
        q.displayInfo();
    }
}
```

Dependency Injection with Factory Method in Spring

Spring framework provides facility to inject bean using factory method. To do so, we can use two attributes of `<bean>` element.

1. **factory-method**: represents the factory method that will be invoked to inject the bean.
2. **factory-bean**: represents the reference of the bean by which factory method will be invoked. It is used if factory method is non-static.

A method that returns instance of a class is called **factory method**.

```
public class A {
    public static A getA(){//factory method
        return new A();
    }
}
```

Factory Method Types

There can be three types of factory method:

- 1) A static factory method that returns instance of its own class. It is used in singleton design pattern



```
<bean id="a" class="com.edubridge.A" factory-method="getA"></bean>
```

- 2) A static factory method that returns instance of another class. It is used instance is not known and decided at runtime.

```
<bean id="b" class="com.edubridge.A" factory-method="getB"></bean>
```

- 3) A non-static factory method that returns instance of another class. It is used instance is not known and decided at runtime.

```
<bean id="a" class="com.edubridge.A"></bean>  
<bean id="b" class="com.edubridge.A" factory-method="getB" factory-bean="a"></bean>
```

Type 1

Let's see the simple code to inject the dependency by static factory method.

```
<bean id="a" class="com.edubridge.A" factory-method="getA"></bean>
```

Let's see the full example to inject dependency using factory method in spring. To create this example, we have created 3 files.

1. A.java
2. applicationContext.xml
3. Test.java



A.java

This class is a singleton class.

```
package com.javatpoint;
public class A {
    private static final A obj=new A();
    private A(){System.out.println("private constructor");}
    public static A getA(){
        System.out.println("factory method ");
        return obj;
    }
    public void msg(){
        System.out.println("hello user");
    }
}
```

applicationContext.xml

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

    <bean id="a" class="com.javatpoint.A" factory-method="getA"></bean>

</beans>
```

Test.java

This class gets the bean from the applicationContext.xml file and calls the msg method.



```
package org.sssit;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
public class Test {
    public static void main(String[] args) {
        ApplicationContext context=new ClassPathXmlApplicationContext("applicationContext.xml");
        A a=(A)context.getBean("a");
        a.msg();
    }
}
```

Output:

```
private constructor
factory method
hello user
```

Type 2

Let's see the simple code to inject the dependency by static factory method that returns the instance of another class. To create this example, we have created 6 files.

1. Printable.java
2. A.java
3. B.java
4. PrintableFactory.java
5. applicationContext.xml
6. Test.java

Printable.java

```
package com.javatpoint;
public interface Printable {
    void print();
}
```



A.java

```
package com.javatpoint;
public class A implements Printable{
    @Override
    public void print() {
        System.out.println("hello a");
    }
}
```

B.java

```
package com.javatpoint;
public class B implements Printable{
    @Override
    public void print() {
        System.out.println("hello b");
    }
}
```

PrintableFactory.java

```
package com.javatpoint;
public class PrintableFactory {
    public static Printable getPrintable(){
        //return new B();
        return new A();//return any one instance, either A or B
    }
}
```

applicationContext.xml



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

  <bean id="p" class="com.javatpoint.PrintableFactory" factory-method="getPrintable"></bean>
</beans>
```

Test.java

This class gets the bean from the applicationContext.xml file and calls the print() method.

```
package org.sssit;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
public class Test {
  public static void main(String[] args) {
    ApplicationContext context=new ClassPathXmlApplicationContext("applicationContext.xml");
    Printable p=(Printable)context.getBean("p");
    p.print();
  }
}
```

Output:

```
hello a
```

Type 3

Let's see the example to inject the dependency by non-static factory method that returns the instance of another class.

To create this example, we have created 6 files.

1. **Printable.java**
2. **A.java**



3. B.java
4. PrintableFactory.java
5. applicationContext.xml
6. Test.java

All files are same as previous, you need to change only 2 files: PrintableFactory and applicationContext.xml.

PrintableFactory.java

```
package com.javatpoint;
public class PrintableFactory {
    //non-static factory method
    public Printable getPrintable(){
        return new A();//return any one instance, either A or B
    }
}
```

applicationContext.xml

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

    <bean id="pfactory" class="com.javatpoint.PrintableFactory"></bean>
    <bean id="p" class="com.javatpoint.PrintableFactory" factory-method="getPrintable"
    factory-bean="pfactory"></bean>

</beans>
```

Output:

```
hello a
```



Crucial Namespaces 'p' and 'c'

P-Namespace

In general, in the setter method dependency injection, to specify dependent values in the spring configuration file, we have to use `<property>` tags as per the number of properties in the bean class. In this context, to remove `<property>` tags and to provide dependent values as attributes in tag in the spring configuration file, we have to use "P- Namespace".

In simple words, while using Spring XML configuration for wiring beans, you would have used `<property>` element several times to provide property values and/or reference for beans. If you are looking for any shorter alternative to nested `<property>` element, you can use p-namespace in Spring.

To use p-namespace in the spring configuration file, we must define the "p" namespace in XSD like below.

```
xmlns:p="http://www.springframework.org/schema/p"
```

To provide value as attribute by using "p" namespace in `<bean>` tag we have to use the following syntax.

```
<bean id="--" class="--" p:prop_Name="value" p:prop_Name="value" .../>
```

If we want to specify object reference variable as dependent value then we have to use "-ref" along with property.

```
<bean id="--" class="--" p:prop_Name-ref="ref"/>
```

C-Namespace

In general, in constructor dependency injection, to specify dependent values in the spring configuration file, we have to use `<constructor-arg>` tags as per the no of parameters that we defined in the bean class constructor. In this context, to remove `<constructor-arg>` tags and to provide dependent values as attributes in tag in the spring configuration file, we have to use "C-Namespace".

The c-namespace in Spring enables you to use the bean element's attributes for configuring the constructor arguments rather than nested constructor-arg elements.

To use c-namespace in the spring configuration file, we must define the "c" namespace in XSD like below.

```
xmlns:c="http://www.springframework.org/schema/c"
```

We must use the following syntax to provide value as an attribute by using the "c" namespace in the `<bean>` tag.



```
<bean id="--" class="--" c:arg_Name="value" c:arg_Name="value" .../>
```

we want to specify object reference variable as dependent value then we have to use “-ref” along with argument_Name.

```
<bean id="--" class="--" c:arg_Name-ref="ref"/>
```

If we want to specify dependent values in the beans configuration file based on index values, we have to use XML code like below.

```
<bean id="--" class="--" c:_0="val1" c:_1="val2"...c:_4-ref="ref"/>
```

Example:

EduBridge





```
package com.tanaya.spring.core.bean.dependencyinjection.namespace.p.beans;

public class Employee {
    private String empName;
    private String empId;
    private String empAddress;
    private double salary;

    public String getEmpName() {
        return empName;
    }
    public void setEmpName(String empName) {
        this.empName = empName;
    }
    public String getEmpId() {
        return empId;
    }
    public void setEmpId(String empId) {
        this.empId = empId;
    }
    public String getEmpAddress() {
        return empAddress;
    }
    public void setEmpAddress(String empAddress) {
        this.empAddress = empAddress;
    }
    public double getSalary() {
        return salary;
    }
    public void setSalary(double salary) {
        this.salary = salary;
    }
    @Override
    public String toString() {
        return "Employee [empName=" + empName + ", empId=" + empId + ", empAddress=" +
            empAddress + ",salary=" + salary
            + "]\n";
    }
}
```



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

<bean id="emp"
class="com.tanaya.spring.core.bean.dependencyinjection.namespace.p.beans.Employee"
p:empName="Tanaya Deshpande " p:empId="Emp0092" p:empAddress="Mumbai"
p:salary="1,00, 000" />
</beans>
```

```
package com.tanaya.spring.core.bean.dependencyinjection.namespace.p.test;

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

import com.tanaya.spring.core.bean.dependencyinjection.namespace.p.beans.Employee;

public class TestSpringApplication {
    @SuppressWarnings("resource")
    public static void main(String[] args) {
        String configFile =
"/com/ashok/spring/core/bean/dependencyinjection/namespace/p/config/applicationContext.xml";
        ApplicationContext context = new ClassPathXmlApplicationContext(configFile);
        Employee emp = (Employee) context.getBean("emp");
        System.out.println(emp);
    }
}
```

Output:

```
Employee [empName=Tanaya Deshpande, empId=Emp0092, empAddress=Mumbai, salary=1,00,000]
```

Injecting Collection

You have seen how to configure primitive data type using value attribute and object references using ref attribute of the <property> tag in your Bean configuration file. Both the cases deal with passing singular value to a bean.



Now what if you want to pass plural values like Java Collection types such as List, Set, Map, and Properties. To handle the situation, Spring offers four types of collection configuration elements which are as follows –

| Sr.No | Element & Description |
|-------|--|
| 1 | <code><list></code> This helps in wiring ie injecting a list of values, allowing duplicates. |
| 2 | <code><set></code> This helps in wiring a set of values but without any duplicates. |
| 3 | <code><map></code> This can be used to inject a collection of name-value pairs where name and value can be of any type. |
| 4 | <code><props></code> This can be used to inject a collection of name-value pairs where the name and value are both Strings. |

You can use either `<list>` or `<set>` to wire any implementation of `java.util.Collection` or an array.

You will come across two situations (a) Passing direct values of the collection and (b) Passing a reference of a bean as one of the collection elements.

Example:

Let us have a working Eclipse IDE in place and take the following steps to create a Spring application –

| Steps | Description |
|-------|--|
| 1 | Create a project with a name SpringExample and create a package com.edubridge under the src folder in the created project. |
| 2 | Add required Spring libraries using Add External JARs option as explained in the Spring Hello World Example chapter. |
| 3 | Create Java classes JavaCollection, and MainApp under the com.edubridge package. |
| 4 | Create Beans configuration file Beans.xml under the src folder. |
| 5 | The final step is to create the content of all the Java files and Bean Configuration file and run the application as explained below |

Here is the content of JavaCollection.java file –



```
package com.edubridge;
import java.util.*;

public class JavaCollection {
    List addressList;
    Set addressSet;
    Map addressMap;
    Properties addressProp;
    // a setter method to set List
    public void setAddressList(List addressList) {
        this.addressList = addressList;
    }
    // prints and returns all the elements of the list.
    public List getAddressList() {
        System.out.println("List Elements :" + addressList);
        return addressList;
    }
    // a setter method to set Set
    public void setAddressSet(Set addressSet) {
        this.addressSet = addressSet;
    }
    // prints and returns all the elements of the Set.
    public Set getAddressSet() {
        System.out.println("Set Elements :" + addressSet);
        return addressSet;
    }
}
```




```
// a setter method to set Map
public void setAddressMap(Map addressMap) {
    this.addressMap = addressMap;
}
// prints and returns all the elements of the Map.
public Map getAddressMap() {
    System.out.println("Map Elements :" + addressMap);
    return addressMap;
}
// a setter method to set Property
public void setAddressProp(Properties addressProp) {
    this.addressProp = addressProp;
}
// prints and returns all the elements of the Property.
public Properties getAddressProp() {
    System.out.println("Property Elements :" + addressProp);
    return addressProp;
}
}
```

Following is the content of the MainApp.java file –

```
package com.edubridge;

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

public class MainApp {
    public static void main(String[] args) {
        ApplicationContext context = new ClassPathXmlApplicationContext("Beans.xml");
        JavaCollection jc=(JavaCollection)context.getBean("javaCollection");

        jc.getAddressList();
        jc.getAddressSet();
        jc.getAddressMap();
        jc.getAddressProp();
    }
}
```



Following is the configuration file **Beans.xml** which has configuration for all the type of collections –

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

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

  <!-- Definition for javaCollection -->
  <bean id = "javaCollection" class = "com.edubridge.JavaCollection">

    <!-- results in a setAddressList(java.util.List) call -->
    <property name = "addressList">
      <list>
        <value>INDIA</value>
        <value>Pakistan</value>
        <value>USA</value>
        <value>USA</value>
      </list>
    </property>

    <!-- results in a setAddressSet(java.util.Set) call -->
    <property name = "addressSet">
      <set>
        <value>INDIA</value>
        <value>Pakistan</value>
        <value>USA</value>
        <value>USA</value>
      </set>
    </property>
```



```
<!-- results in a setAddressSet(java.util.Set) call -->
<property name = "addressSet">
  <set>
    <value>INDIA</value>
    <value>Pakistan</value>
    <value>USA</value>
    <value>USA</value>
  </set>
</property>

<!-- results in a setAddressMap(java.util.Map) call -->
<property name = "addressMap">
  <map>
    <entry key = "1" value = "INDIA"/>
    <entry key = "2" value = "Pakistan"/>
    <entry key = "3" value = "USA"/>
    <entry key = "4" value = "USA"/>
  </map>
</property>

<!-- results in a setAddressProp(java.util.Properties) call -->
<property name = "addressProp">
  <props>
    <prop key = "one">INDIA</prop>
    <prop key = "one">INDIA</prop>
    <prop key = "two">Pakistan</prop>
    <prop key = "three">USA</prop>
    <prop key = "four">USA</prop>
  </props>
</property>
</bean>
</beans>
```

Once you are done creating the source and bean configuration files, let us run the application. If everything is fine with your application, it will print the following message –

```
List Elements :[INDIA, Pakistan, USA, USA]
Set Elements :[INDIA, Pakistan, USA]
ap Elements :{1 = INDIA, 2 = Pakistan, 3 = USA, 4 = USA}
Property Elements :{two = Pakistan, one = INDIA, three = USA, four = USA}
```



Metadata / Configuration

Annotation Based Configuration

Starting from Spring 2.5 it became possible to configure the dependency injection using annotations. So instead of using XML to describe a bean wiring, you can move the bean configuration into the component class itself by using annotations on the relevant class, method, or field declaration.

Annotation injection is performed before XML injection. Thus, the latter configuration will override the former for properties wired through both approaches.

Annotation wiring is not turned on in the Spring container by default. So, before we can use annotation-based wiring, we will need to enable it in our Spring configuration file. So, consider the following configuration file in case you want to use any annotation in your Spring application.

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

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

  <context:annotation-config/>
  <!-- bean definitions go here -->
</beans>
```

Once `<context:annotation-config/>` is configured, you can start annotating your code to indicate that Spring should automatically wire values into properties, methods, and constructors. Let us look at a few important annotations to understand how they work –

| Sr.No. | Annotation & Description |
|--------|---|
| 1 | @Required The @Required annotation applies to bean property setter methods. |
| 2 | @Autowired The @Autowired annotation can apply to bean property setter methods, non-setter methods, constructor and properties. |



| | |
|---|---|
| 3 | @Qualifier The @Qualifier annotation along with @Autowired can be used to remove the confusion by specifying which exact bean will be wired. |
| 4 | JSR-250 Annotations Spring supports JSR-250 based annotations which include @Resource, @PostConstruct and @PreDestroy annotations. |

- **Spring @Required Annotation**

The @Required annotation applies to bean property setter methods and it indicates that the affected bean property must be populated in XML configuration file at configuration time. Otherwise, the container throws a BeanInitializationException exception. Following is an example to show the use of @Required annotation.

Example

Let us have a working Eclipse IDE in place and take the following steps to create a Spring application –

| Steps | Description |
|-------|---|
| 1 | Create a project with a name SpringExample and create a package com.edubridge under the src folder in the created project. |
| 2 | Add required Spring libraries using Add External JARs option as explained in the Spring Hello World Example chapter. |
| 3 | Create Java classes Student and MainApp under the com.edubridge package. |
| 4 | Create Beans configuration file Beans.xml under the src folder. |
| 5 | The final step is to create the content of all the Java files and Bean Configuration file and run the application as explained below. |

Here is the content of **Student.java** file –



```
package com.edubridge;
import org.springframework.beans.factory.annotation.Required;
public class Student {
    private Integer age;
    private String name;

    @Required
    public void setAge(Integer age) {
        this.age = age;
    }
    public Integer getAge() {
        return age;
    }
    @Required
    public void setName(String name) {
        this.name = name;
    }
    public String getName() {
        return name;
    }
}
```

Following is the content of the MainApp.java file –

```
package com.edubridge;

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

public class MainApp {
    public static void main(String[] args) {
        ApplicationContext context = new ClassPathXmlApplicationContext("Beans.xml");

        Student student = (Student) context.getBean("student");
        System.out.println("Name : " + student.getName() );
        System.out.println("Age : " + student.getAge() );
    }
}
```



Following is the content of the configuration file Beans.xml –

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

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

  <context:annotation-config/>

  <!-- Definition for student bean -->
  <bean id = "student" class = "com.edubridge.Student">
    <property name = "name" value = "Zara" />

    <!-- try without passing age and check the result -->
    <!-- property name = "age" value = "11"-->
  </bean>
</beans>
```

Once you are done creating the source and bean configuration files, let us run the application. If everything is fine with your application, it will raise `BeanInitializationException` exception and print the following error along with other log messages –

```
Property 'age' is required for bean 'student'
```

Next, you can try the above example after removing the comment from 'age' property as follows –



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

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

  <context:annotation-config/>

  <!-- Definition for student bean -->
  <bean id = "student" class = "com.edubridge.Student">
    <property name = "name" value = "Zara" />
    <property name = "age" value = "11"/>
  </bean>
</beans>
```

The above example will produce the following result –

```
Name: Zara
Age: 11
```

- **Spring @Autowired Annotation**

The @Autowired annotation provides more fine-grained control over where and how autowiring should be accomplished. The @Autowired annotation can be used to autowire bean on the setter method just like @Required annotation, constructor, a property or methods with arbitrary names and/or multiple arguments.

@Autowired on Setter Methods

You can use @Autowired annotation on setter methods to get rid of the <property> element in XML configuration file. When Spring finds an @Autowired annotation used with setter methods, it tries to perform byType autowiring on the method.

Example

Let us have working Eclipse IDE in place and follow the following steps to create a Spring application –



| Step | Description |
|------|---|
| 1 | Create a project with a name SpringExample and create a package com.edubridge under the src folder in the created project. |
| 2 | Add required Spring libraries using Add External JARs option as explained in the Spring Hello World Example chapter. |
| 3 | Create Java classes TextEditor, SpellChecker and MainApp under the com.edubridge package. |
| 4 | Create Beans configuration file Beans.xml under the src folder. |
| 5 | The final step is to create the content of all the Java files and Bean Configuration file and run the application as explained below. |

Here is the content of **TextEditor.java** file –

```
package com.edubridge;

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

public class TextEditor {
    private SpellChecker spellChecker;

    @Autowired
    public void setSpellChecker( SpellChecker spellChecker ){
        this.spellChecker = spellChecker;
    }
    public SpellChecker getSpellChecker( ) {
        return spellChecker;
    }
    public void spellCheck() {
        spellChecker.checkSpelling();
    }
}
```

Following is the content of another dependent class file **SpellChecker.java**:



```
package com.edubridge;

public class SpellChecker {
    public SpellChecker(){
        System.out.println("Inside SpellChecker constructor." );
    }
    public void checkSpelling(){
        System.out.println("Inside checkSpelling." );
    }
}
```

Following is the content of the **MainApp.java** file –

```
package com.edubridge;

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

public class MainApp {
    public static void main(String[] args) {
        ApplicationContext context = new ClassPathXmlApplicationContext("Beans.xml");

        TextEditor te = (TextEditor) context.getBean("textEditor");

        te.spellCheck();
    }
}
```

Following is the configuration file **Beans.xml** –



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

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

  <context:annotation-config/>

  <!-- Definition for textEditor bean without constructor-arg -->
  <bean id = "textEditor" class = "com.edubridge.TextEditor">
  </bean>

  <!-- Definition for spellChecker bean -->
  <bean id = "spellChecker" class = "com.edubridge.SpellChecker">
  </bean>

</beans>
```

Once you are done with creating source and bean configuration files, let us run the application. If everything is finewith your application, this will print the following message –

```
Inside TextEditor constructor.
Inside SpellChecker constructor.
Inside checkSpelling.
```

@Autowired on Constructors

You can apply @Autowired to constructors as well. A constructor @Autowired annotation indicates that the constructor should be autowired when creating the bean, even if no <constructor-arg> elements are used whileconfiguring the bean in XML file. Let us check the following example.

Here is the content of **TextEditor.java** file –



```
package com.edubridge;

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

public class TextEditor {
    private SpellChecker spellChecker;

    @Autowired
    public TextEditor(SpellChecker spellChecker){
        System.out.println("Inside TextEditor constructor." );
        this.spellChecker = spellChecker;
    }
    public void spellCheck(){
        spellChecker.checkSpelling();
    }
}
```

Following is the configuration file **Beans.xml** –

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

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

    <context:annotation-config/>

    <!-- Definition for textEditor bean without constructor-arg -->
    <bean id = "textEditor" class = "com.edubridge.TextEditor">
    </bean>

    <!-- Definition for spellChecker bean -->
    <bean id = "spellChecker" class = "com.edubridge.SpellChecker">
    </bean>
</beans>
```



Once you are done with the above two changes in source and bean configuration files, let us run the application. If everything is fine with your application, this will print the following message:

```
Inside SpellChecker constructor.  
Inside TextEditor constructor.  
Inside checkSpelling.
```

- **@Resource Annotation**

You can use @Resource annotation on fields or setter methods and it works the same as in Java EE 5. The @Resource annotation takes a 'name' attribute which will be interpreted as the bean name to be injected. You can say, it follows by-name autowiring semantics as demonstrated in the following example –

```
package com.edubridge;  
import javax.annotation.Resource;  
  
public class TextEditor {  
    private SpellChecker spellChecker;  
  
    @Resource(name = "spellChecker")  
    public void setSpellChecker( SpellChecker spellChecker ){  
        this.spellChecker = spellChecker;  
    }  
    public SpellChecker getSpellChecker(){  
        return spellChecker;  
    }  
    public void spellCheck(){  
        spellChecker.checkSpelling();  
    }  
}
```

If no 'name' is specified explicitly, the default name is derived from the field name or setter method. In case of a field, it takes the field name; in case of a setter method, it takes the bean property name.



@Component, Component Scans, Component Filters

@Component

@Component is a class-level annotation. It is used to denote a class as a Component. We can use @Component across the application to mark the beans as Spring's managed components. A component is responsible for some operations. Spring framework provides three other specific annotations to be used when marking a class as a Component.

- @Service
- @Repository
- @Controller

1. **@Service:** We specify a class with @Service to indicate that they're holding the business logic. Besides being used in the service layer, there isn't any other special use for this annotation. The utility classes can be marked as Service classes.
2. **@Repository:** We specify a class with @Repository to indicate that they're dealing with CRUD operations, usually, it's used with DAO (Data Access Object) or Repository implementations that deal with database tables.
3. **@Controller:** We specify a class with @Controller to indicate that they're front controllers and responsible to handle user requests and return the appropriate response. It is mostly used with REST Web Services.

Example: Spring @Component

Let's create a very simple Spring boot application to showcase the use of Spring Component annotation and how Spring autodetects it with annotation-based configuration and classpath scanning.

Step 1: Create a Simple Spring Boot Project.

Step 2: Add the spring-context dependency in your pom.xml file. Go to the pom.xml file inside your project and add the following spring-context dependency.



```
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-context</artifactId>
  <version>5.3.13</version>
</dependency>
```

Step 3: Create a simple component class

Go to the src > main > java > your package name > right-click > New > Java Class and create your componentclass and mark it with @Component annotation.

Example

```
// Java Program to Illustrate Component class
package com.example.demo;

import org.springframework.stereotype.Component;

// Annotation
@Component

// Class
public class ComponentDemo {

    // Method
    public void demoFunction()
    {

        // Print statement when method is called
        System.out.println("Hello GeeksForGeeks");
    }
}
```

Step 4: Create an annotation-based spring context

Now go to your Application (@SpringBootApplication) file and here in this file create an annotation-based springcontext and get the ComponentDemo bean from it.



```
// Java Program to Illustrate Application class

// Importing package here
package com.example.demo;
// Importing required classes
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;

// Annotation
@SpringBootApplication

// Class
public class DemoApplication {

    // Main driver method
    public static void main(String[] args)
    {
        // Annotation based spring context
        AnnotationConfigApplicationContext context
            = new AnnotationConfigApplicationContext();
        context.scan("com.example.demo");
        context.refresh();

        // Getting the Bean from the component class
        ComponentDemo componentDemo
            = context.getBean(ComponentDemo.class);
        componentDemo.demoFunction();

        // Closing the context
        // using close() method
        context.close();
    }
}
```

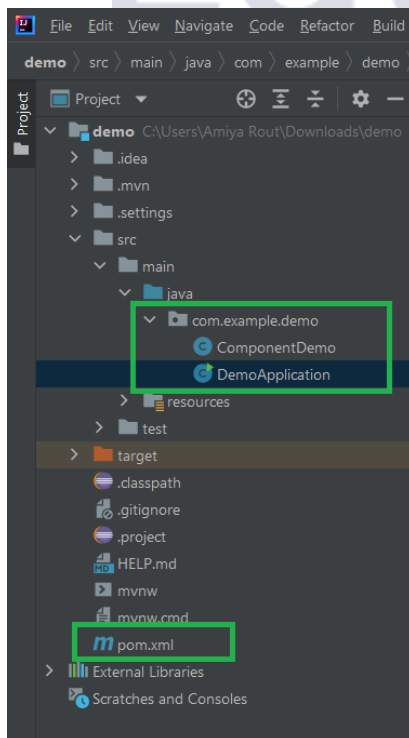
Output:



```

Run: DemoApplication
20:14:38.885 [main] DEBUG org.springframework.beans.factory.support.DefaultListableBeanFactory - Creating shared instance of singleton bean 'org.springframework.context.annotation.AnnotationConfigApplicationContext'
20:14:38.885 [main] DEBUG org.springframework.beans.factory.support.DefaultListableBeanFactory - Autowiring by type from bean name 'mappingJackson2HttpMessageConverter'
20:14:38.885 [main] DEBUG org.springframework.beans.factory.support.DefaultListableBeanFactory - Creating shared instance of singleton bean 'org.springframework.http.converter.json.MappingJackson2HttpMessageConverter'
20:14:38.885 [main] DEBUG org.springframework.beans.factory.support.DefaultListableBeanFactory - Creating shared instance of singleton bean 'org.springframework.context.annotation.AnnotationConfigApplicationContext'
20:14:38.887 [main] DEBUG org.springframework.beans.factory.support.DefaultListableBeanFactory - Creating shared instance of singleton bean 'messageConverters'
20:14:38.919 [main] DEBUG org.springframework.beans.factory.support.DefaultListableBeanFactory - Creating shared instance of singleton bean 'org.springframework.context.annotation.AnnotationConfigApplicationContext'
20:14:38.922 [main] DEBUG org.springframework.beans.factory.support.DefaultListableBeanFactory - Creating shared instance of singleton bean 'spring.info-org.springframework.boot.autoconfigure.SpringBootApplication'
20:14:38.923 [main] DEBUG org.springframework.beans.factory.support.DefaultListableBeanFactory - Autowiring by type from bean name 'org.springframework.boot.autoconfigure.SpringBootApplication'
20:14:38.926 [main] DEBUG org.springframework.beans.factory.support.DefaultListableBeanFactory - Creating shared instance of singleton bean 'org.springframework.context.annotation.AnnotationConfigApplicationContext'
20:14:38.926 [main] DEBUG org.springframework.beans.factory.support.DefaultListableBeanFactory - Creating shared instance of singleton bean 'spring.sql.init-org.springframework.boot.autoconfigure.SpringBootApplication'
20:14:38.928 [main] DEBUG org.springframework.beans.factory.support.DefaultListableBeanFactory - Creating shared instance of singleton bean 'org.springframework.context.annotation.AnnotationConfigApplicationContext'
20:14:38.930 [main] DEBUG org.springframework.beans.factory.support.DefaultListableBeanFactory - Creating shared instance of singleton bean 'scheduledBeanLazyInit'
20:14:38.931 [main] DEBUG org.springframework.beans.factory.support.DefaultListableBeanFactory - Creating shared instance of singleton bean 'taskSchedulerBuilder'
20:14:38.931 [main] DEBUG org.springframework.beans.factory.support.DefaultListableBeanFactory - Creating shared instance of singleton bean 'spring.task.scheduling'
20:14:38.933 [main] DEBUG org.springframework.beans.factory.support.DefaultListableBeanFactory - Autowiring by type from bean name 'taskSchedulerBuilder' via factory method
20:14:38.933 [main] DEBUG org.springframework.beans.factory.support.DefaultListableBeanFactory - Creating shared instance of singleton bean 'org.springframework.context.annotation.AnnotationConfigApplicationContext'
Hello GeeksForGeeks
20:14:38.938 [main] DEBUG org.springframework.context.annotation.AnnotationConfigApplicationContext - Closing org.springframework.context.annotation.AnnotationConfigApplicationContext
Process finished with exit code 0
    
```

So, you can see the power of `@Component` annotation, we didn't have to do anything to inject our component to spring context. The below image shows the directory structure of our Spring Component example project.



All rights reserved.

No part of this document may be reproduced in any material form (including printing and photocopying or storing it in any medium by electronic or other means or not transiently or incidentally to some other use of this document) without the prior written permission of EBSC Technologies Pvt. Ltd. Application for written permission to reproduce any part of this document should be addressed to the CEO of EBSC Technologies Pvt Ltd.



@ComponentScan

Before we rely completely on @Component, we must understand that it's only a plain annotation. The annotation serves the purpose of differentiating beans from other objects, such as domain objects.

However, Spring uses the @ComponentScan annotation to actually gather them all into its ApplicationContext.

If we're writing a Spring Boot application, it is helpful to know that @SpringBootApplication is a composed annotation that includes @ComponentScan. As long as our @SpringBootApplication class is at the root of our project, it will scan every @Component we define by default.

But in case our @SpringBootApplication class can't be at the root of our project or we want to scan outside sources, we can configure @ComponentScan explicitly to look in whatever package we specify, as long as it exists on the classpath.

Let's define an out-of-scope @Component bean:

```
package com.baeldung.component.scannedscope;
```

```
@Component  
public class ScannedScopeExample {  
}
```

Next, we can include it via explicit instructions to our @ComponentScan annotation:

```
package com.edubridge.component.inscope;
```

```
@SpringBootApplication  
@ComponentScan({"com.edubridge.component.inscope", "com.edubridge.component.scannedscope"})  
public class ComponentApplication {  
    //public static void main(String[] args) {...}  
}
```

Finally, we can test that it exists:

```
@Test
public void givenScannedScopeComponent_whenSearchingInApplicationContext_thenFindIt() {
    assertNotNull(applicationContext.getBean(ScannedScopeExample.class));
}
```

@ComponentScan Filter

By default, classes annotated with @Component, @Repository, @Service, @Controller are registered as Spring beans. The same goes for classes annotated with a custom annotation that is annotated with @Component. We can extend this behavior by using includeFilters and excludeFilters parameters of the @ComponentScan annotation.

There are five types of filters available for ComponentScan.Filter:

- ANNOTATION
- ASSIGNABLE_TYPE
- ASPECTJ
- REGEX
- CUSTOM

- **FilterType.ANNOTATION**

The ANNOTATION filter type filters candidates marked with a given annotation. You can either include or exclude components those are marked with specific annotation. Let's see an example, suppose that we have @Mammal annotation.

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
public @interface Mammal {
}
```

Following are the components those are annotation with @Mammal annotation.



```
@Mammal
public class Bear {
}
@Mammal
public class Tiger {
}
@Mammal
public class Whale {
}
```

Let's use the **FilterType.ANNOTATION** to tell Spring to scan for annotated classes with @Mammal annotation.

useDefaultFilters indicates whether automatic detection of classes annotated with @Component, @Repository, @Service, or @Controller should be enabled, by default is **true**. We are disabling it to find components specifically annotated with @Mammal.

```
@Configuration
@ComponentScan(basePackages = "com.javabydeveloper.spring.bean.animal",
    useDefaultFilters = false,
    includeFilters = @ComponentScan.Filter(type = FilterType.ANNOTATION, classes = Mammal.class))
public class ScanAnnotationFilterConfig {
}
```

Testing:

```
public class ScanAnnotationFilterConfigDemo {
    public static void main(String[] args) {
        ApplicationContext ctx = new AnnotationConfigApplicationContext(ScanAnnotationFilterConfig.class);

        String[] springComponents = ctx.getBeanDefinitionNames();

        for(String bean : springComponents)
            System.out.println(bean);
    }
}
```

You will see following spring-beans available in application context.



```
scanAnnotationFilterConfig
bear
tiger
whale
```

- **FilterType. ASSIGNABLE_TYPE**

The ASSIGNABLE_TYPE filter type filter candidates assignable to a given type. The component classes can be implementation classes if the given type is interface OR can be child classes if given types is a concrete class.

Implementing Amphibian interface.

```
public interface Amphibian {
}
public class Frog implements Amphibian {
}
public class Newt implements Amphibian {
}
public class Toad implements Amphibian {
}
```

Extending Crocodile class

```
public class Crocodile {
}
public class DwarfCrocodile extends Crocodile {
}
public class NileCrocodile extends Crocodile{
}
public class OrinocoCrocodile extends Crocodile{
}
public class SiameseCrocodile extends Crocodile{
}
```

Let's use the FilterType.ASSIGNABLE_TYPE to tell Spring to scan for the classes which extending Crocodile class or implementing Amphibian interface.



```
@Configuration
@ComponentScan(basePackages = "com.javabydeveloper.spring.bean.animal",
    useDefaultFilters = false,
    includeFilters = @ComponentScan.Filter(type = FilterType.ASSIGNABLE_TYPE,
        classes = {Amphibian.class, Crocodile.class}))
public class ScanAssignableTypeFilterConfig {
}
```

Testing:

```
public class ScanAssignableTypeFilterConfigDemo {
    public static void main(String[] args) {
        ApplicationContext ctx = new
        AnnotationConfigApplicationContext(ScanAssignableTypeFilterConfig.class);

        String[] springComponents = ctx.getBeanDefinitionNames();

        for(String bean : springComponents)
            System.out.println(bean);
    }
}
```

Results:

```
scanAssignableTypeFilterConfig
frog
newt
toad
crocodile
dwarfCrocodile
nileCrocodile
orinocoCocodile
siameseCrocodile
```

- **FilterType.ASPECTJ**

The ASPECTJ filter type, Filter candidates matching a given AspectJ type pattern expression. which is useful to filter complex matched patterns.



Let's say we need to scan classes only whose name ends with Cobra and does not contain Mono within reptiles package. Let's see an example, to filter components using ASPECTJ expression.]

```
@Configuration
@ComponentScan(basePackages = "com.javabydeveloper.spring.bean.animal.reptiles",
    useDefaultFilters = false,
    includeFilters = @ComponentScan.Filter(type = FilterType.ASPECTJ,
        pattern = "*..*Cobra && !(*..Mono*)"))
// pattern = "*..Black* || *..E*)"
public class ScanAspectJFilterConfig {
}
```

Testing:

```
public class ScanAspectJFilterConfigDemo {
    public static void main(String[] args) {
        ApplicationContext ctx = new AnnotationConfigApplicationContext(ScanAspectJFilterConfig.class);

        String[] springComponents = ctx.getBeanDefinitionNames();

        for(String bean : springComponents)
            System.out.println(bean);
    }
}
```

Results:

Refer the **snakes** package to understand results.

```
scanAspectJFilterConfig
capeCobra
egyptianCobra
equatorialCobra
```

- **FilterType.REGEX**

The REGEX filter type, filter candidates matching a given regular expression type pattern expression. which is useful to filter simple or fully-qualified class names.

Let's say we need to scan classes whose name ends with Mamba. Let's see an example, to filter components using



REGEX expression.

```
@Configuration
@ComponentScan(basePackages = "com.javabydeveloper.spring.bean.animal",
    useDefaultFilters = false,
    includeFilters = @ComponentScan.Filter(type = FilterType.REGEX,
        pattern = ".*(Mamba)"))
public class ScanRegexFilterConfig {
}
```

Testing:

```
public class ScanRegexFilterConfigDemo {
    public static void main(String[] args) {
        ApplicationContext ctx = new AnnotationConfigApplicationContext(ScanRegexFilterConfig.class);

        String[] springComponents = ctx.getBeanDefinitionNames();

        for(String bean : springComponents)
            System.out.println(bean);
    }
}
```

Results:

Refer the **snakes** package to understand results.

```
scanRegexFilterConfig
blackMamba
easternGreenMamba
```




- **FilterType.CUSTOM**

Sometimes the above filter types sufficient for our requirement to filter components, in such cases we can create custom filter and can apply that using CUSTOM filter type.

In order to create our custom filter, need to implements `org.springframework.core.type.filter.TypeFilter` interface and provide the implementation for match method.

Let's implement Custom Filter Type to filter classes those are implements Amphibian interface or class name ends with Crocodile.

```
public class CustomTypeFilter implements TypeFilter {

    @Override
    public boolean match(MetadataReader metadataReader, MetadataReaderFactory
        metadataReaderFactory)
        throws IOException {

        ClassMetadata classMetadata = metadataReader.getClassMetadata();
        String[] interfaces = classMetadata.getInterfaceNames();

        if (Arrays.stream(interfaces).anyMatch(Amphibian.class.getName()::equals))
            return true;

        String className = classMetadata.getClassName();

        if(className.endsWith("Crocodile"))
            return true;

        return false;
    }
}
```

Let's use the FilterType.CUSTOM to tell Spring to scan for the classes those names are ending with Crocodile or implementing Amphibian interface.



```
@Configuration
@ComponentScan(basePackages = "com.javabydeveloper.spring.bean.animal",
    useDefaultFilters = false,
    includeFilters = @ComponentScan.Filter(type = FilterType.CUSTOM, classes = CustomTypeFilter.class))
public class ScanCustomTypeFilterConfig {
}
```

Testing:

```
public class ScanCustomFilterTypeConfigDemo {
    public static void main(String[] args) {
        ApplicationContext ctx = new AnnotationConfigApplicationContext(ScanCustomTypeFilterConfig.class);
        String[] springComponents = ctx.getBeanDefinitionNames();

        for(String bean : springComponents)
            System.out.println(bean);
    }
}
```

Results:

Refer **amphibians** and **crocodiles** packages to understand results.

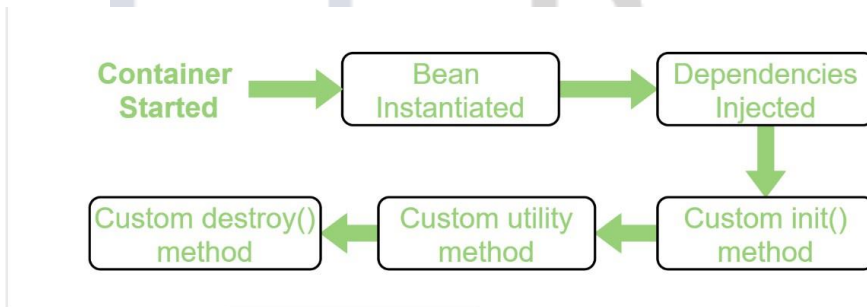
```
scanCustomTypeFilterConfig
frog
newt
toad
crocodile
dwarf crocodile
nile crocodile
orinoco crocodile
siamese crocodile
```

Bean life cycle

The lifecycle of any object means when & how it is born, how it behaves throughout its life, and when & how it dies. Similarly, the bean life cycle refers to when & how the bean is instantiated, what action it performs until it lives, and when & how it is destroyed.

Bean life cycle is managed by the spring container. When we run the program then, first of all, the spring container gets started. After that, the container creates the instance of a bean as per the request, and then dependencies are injected. And finally, the bean is destroyed when the spring container is closed. Therefore, if we want to execute some code on the bean instantiation and just after closing the spring container, then we can write that code inside the custom `init()` method and the `destroy()` method.

The following image shows the process flow of the bean life cycle.



Note: We can choose custom method name instead of `init()` and `destroy()`. Here, we will use `init()` method to execute all its code as the spring container starts up and the bean is instantiated, and `destroy()` method to execute all its code on closing the container.

Ways to implement the life cycle of a bean

Spring provides three ways to implement the life cycle of a bean. In order to understand these three ways, let's take an example. In this example, we will write and activate `init()` and `destroy()` method for our bean (`HelloWorld.java`) to print some message on start and close of Spring container. Therefore, the three ways to implement this are:

1. **By XML:** In this approach, in order to avail custom `init()` and `destroy()` method for a bean we have to register these two methods inside Spring XML configuration file while defining a bean.
2. **By Programmatic Approach:** To provide the facility to the created bean to invoke custom `init()` method on the startup of a spring container and to invoke the custom `destroy()` method on closing the container, we need to implement our bean with two interfaces namely `InitializingBean`, `DisposableBean` and will have to override `afterPropertiesSet()` and `destroy()` method. `afterPropertiesSet()` method is invoked as the container starts and the bean is instantiated whereas, the `destroy()` method is invoked just after the



container is closed.

Note: To invoke destroy method we have to call a close() method of ConfigurableApplicationContext.

3. **Using Annotation:** To provide the facility to the created bean to invoke custom init() method on the startup of aspring container and to invoke the custom destroy() method on closing the container, we need annotate init() method by @PostConstruct annotation and destroy() method by @PreDestroy annotation.

Note: To invoke the destroy() method we have to call the close() method of ConfigurableApplicationContext. Therefore, the following steps are followed:

- Firstly, we need to create a bean HelloWorld.java in this case and annotate the custom init() methodwith @PostConstruct and destroy() method with @PreDestroy.

Java

EduBridge



```
// Java program to create a bean
// in the spring framework
package beans;

import javax.annotation.PostConstruct;
import javax.annotation.PreDestroy;

// HelloWorld class
public class HelloWorld {

    // Annotate this method to execute it
    // automatically as the bean is
    // instantiated
    @PostConstruct
    public void init() throws Exception
    {
        System.out.println(
            "Bean HelloWorld has been "
            + "instantiated and I'm the "
            + "init() method");
    }

    // Annotate this method to execute it
    // when Spring container is closed
    @PreDestroy
    public void destroy() throws Exception
    {
        System.out.println(
            "Container has been closed "
            + "and I'm the destroy() method");
    }
}
```

Now, we need to configure the spring XML file spring.xml and define the bean.



HTML

```
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN"
    "http://www.springframework.org/dtd/spring-beans-2.0.dtd">

<beans>

    <!-- activate the @PostConstruct and
    @PreDestroy annotation -->

    <bean class="org.springframework
    .context.annotation
    .CommonAnnotationBeanPostProcessor"/>

    <!-- configure the bean -->
    <bean class="beans.HelloWorld"/>

</beans>
```

Finally, we need to create a driver class to run this bean.

Java



```
// Java program to call the
// bean initialized above

package test;

import org.springframework
    .context
    .ConfigurableApplicationContext;

import org.springframework
    .context.support
    .ClassPathXmlApplicationContext;

import beans.HelloWorld;

// Driver class
public class Client {

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

        // Loading the Spring XML configuration
        // file into Spring container and
        // it will create the instance of the
        // bean as it loads into container
        ConfigurableApplicationContext cap
            = new ClassPathXmlApplicationContext(
                "resources/spring.xml");

        // It will close the Spring container
        // and as a result invokes the
        // destroy() method
        cap.close();
    }
}
```

Output:

Bean HelloWorld has been instantiated and I'm the init() method
Container has been closed and I'm the destroy() method



Java Based Configuration

@Configuration & @Bean Annotations

Annotating a class with the @Configuration indicates that the class can be used by the Spring IoC container as a source of bean definitions. The @Bean annotation tells Spring that a method annotated with @Bean will return an object that should be registered as a bean in the Spring application context. The simplest possible @Configuration class would be as follows –

```
package com.edubridge;
import org.springframework.context.annotation.*;

@Configuration
public class HelloWorldConfig {
    @Bean
    public HelloWorld helloWorld(){
        return new HelloWorld();
    }
}
```

The above code will be equivalent to the following XML configuration –

```
<beans>
  <bean id = "helloWorld" class = "com.edubridge.HelloWorld" />
</beans>
```

Here, the method name is annotated with @Bean works as bean ID and it creates and returns the actual bean. Your configuration class can have a declaration for more than one @Bean. Once your configuration classes are defined, you can load and provide them to Spring container using AnnotationConfigApplicationContext as follows –

```
public static void main(String[] args) {
    ApplicationContext ctx = new AnnotationConfigApplicationContext(HelloWorldConfig.class);

    HelloWorld helloWorld = ctx.getBean(HelloWorld.class);
    helloWorld.setMessage("Hello World!");
    helloWorld.getMessage();
}
```

You can load various configuration classes as follows –



```
public static void main(String[] args) {
    AnnotationConfigApplicationContext ctx = new AnnotationConfigApplicationContext();

    ctx.register(AppConfig.class, OtherConfig.class);
    ctx.register(AdditionalConfig.class);
    ctx.refresh();

    MyService myService = ctx.getBean(MyService.class);
    myService.doStuff();
}
```

Example

Let us have a working Eclipse IDE in place and take the following steps to create a Spring application –

| Steps | Description |
|-------|---|
| 1 | Create a project with a name <i>SpringExample</i> and create a package <i>com.edubridge</i> under the src folder in the created project. |
| 2 | Add required Spring libraries using <i>Add External JARs</i> option as explained in the <i>Spring Hello World Example</i> chapter. |
| 3 | Because you are using Java-based annotations, so you also need to add <i>CGLIB.jar</i> from your Java installation directory and <i>ASM.jar</i> library which can be downloaded from asm.ow2.org . |
| 4 | Create Java classes <i>HelloWorldConfig</i> , <i>HelloWorld</i> and <i>MainApp</i> under the <i>com.edubridge</i> package. |
| 5 | The final step is to create the content of all the Java files and Bean Configuration file and run the application as explained below. |

Here is the content of HelloWorldConfig.java file



```
package com.edubridge;
import org.springframework.context.annotation.*;

@Configuration
public class HelloWorldConfig {
    @Bean
    public HelloWorld helloWorld(){
        return new HelloWorld();
    }
}
```

Here is the content of HelloWorld.java file

```
package com.edubridge;

public class HelloWorld {
    private String message;

    public void setMessage(String message){
        this.message = message;
    }
    public void getMessage(){
        System.out.println("Your Message : " + message);
    }
}
```

Following is the content of the MainApp.java file



```
package com.edubridge;

import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.*;

public class MainApp {
    public static void main(String[] args) {
        ApplicationContext ctx =
            new AnnotationConfigApplicationContext(HelloWorldConfig.class);

        HelloWorld helloWorld = ctx.getBean(HelloWorld.class);
        helloWorld.setMessage("Hello World!");
        helloWorld.getMessage();
    }
}
```

Once you are done creating all the source files and adding the required additional libraries, let us run the application. You should note that there is no configuration file required. If everything is fine with your application, it will print the following message –

Your Message: Hello World!

Exercise

Trainer will initiate a discussion of common questions on design pattern as given below:

1. Which of the below is not a valid classification of design pattern?

- | | |
|-------------------------|------------------------|
| a) Creational patterns | b) Structural patterns |
| c) Behavioural patterns | d) Java Patterns |

2. Which design pattern provides a single class that provides simplified methods required by the client and delegates call to those methods?

- a) Adapter pattern
- b) Builder pattern
- c) Facade pattern
- d) Prototype Pattern

3. Which design pattern suggests multiple classes through which a request is passed and multiple but only relevant classes carry out operations on the request?

- a) Singleton pattern
- b) Builder pattern
- c) Facade pattern
- d) Prototype Pattern

4. Attach additional responsibilities to an object dynamically. It provides a flexible alternative to subclassing for extending functionality.

- a) Chain of responsibility
- b) Adapter
- c) Decorator
- d) Prototype

5. Which of the following pattern refers to creating duplicate object while keeping performance in mind?

- a) Builder pattern
- b) Bridge pattern
- c) Prototype pattern
- d) Filter pattern



Introduction to Spring Boot

What is Micro Service

Micro Service is an architecture that allows the developers to develop and deploy services independently. Each service running has its own process and this achieves the lightweight model to support business applications.

Advantages

Micro services offer the following advantages to its developers –

- Easy deployment
- Simple scalability
- Compatible with Containers
- Minimum configuration
- Lesser production time

What is Spring Boot?

- Spring Boot is an open source Java-based framework used to create a micro Service. It is developed by PivotalTeam and is used to build stand-alone and production ready spring applications.
- Spring Boot provides a good platform for Java developers to develop a stand-alone and production-grade spring application that you can just run. You can get started with minimum configurations without the need for an entire Spring configuration setup.

Advantages

Spring Boot offers the following advantages to its developers –

- Easy to understand and develop spring applications
- Increases productivity
- Reduces the development time

Goals

Spring Boot is designed with the following goals –

- To avoid complex XML configuration in Spring
- To develop a production ready Spring applications in an easier way
- To reduce the development time and run the application independently
- Offer an easier way of getting started with the application



Why Spring Boot?

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

- It provides a flexible way to configure Java Beans, XML configurations, and Database Transactions.
- It provides a powerful batch processing and manages REST endpoints.
- In Spring Boot, everything is auto configured; no manual configurations are needed.
- It offers annotation-based spring application
- Eases dependency management
- It includes Embedded Servlet Container

How does spring boot work?

Spring Boot automatically configures your application based on the dependencies you have added to the project by using **@EnableAutoConfiguration** annotation. For example, if MySQL database is on your classpath, but you have not configured any database connection, then Spring Boot auto-configures an in-memory database.

The entry point of the spring boot application is the class contains **@SpringBootApplication** annotation and the main method.

Spring Boot automatically scans all the components included in the project by using **@ComponentScan** annotation.

Spring Boot Starters

Handling dependency management is a difficult task for big projects. Spring Boot resolves this problem by providing a set of dependencies for developer's convenience.

For example, if you want to use Spring and JPA for database access, it is sufficient if you include **spring-boot-starter-data-jpa** dependency in your project.

Note that all Spring Boot starters follow the same naming pattern **spring-boot-starter-***, where * indicates that it is a type of the application.

Examples

Look at the following Spring Boot starters explained below for a better understanding –

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



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

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

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

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

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

Spring Boot Starter Thyme Leaf dependency is used to create a web application. Its code is shown below –

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

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

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



Gradle Plugin

The Spring Boot Gradle plugin helps us manage Spring Boot dependencies, as well as package and run our application when using Gradle as a build tool.

Build File Configuration

First, we need to add the Spring Boot plugin to our build.gradle file by including it in our plugins section:

```
plugins {  
    id "org.springframework.boot" version "2.0.1.RELEASE"  
}
```

If we're using a Gradle version earlier than 2.1 or we need dynamic configuration, we can add it like this instead:

```
buildscript {  
    ext {  
        springBootVersion = '2.0.1.RELEASE'  
    }  
    repositories {  
        mavenCentral()  
    }  
    dependencies {  
        classpath(  
            "org.springframework.boot:spring-boot-gradle-plugin:${springBootVersion}")  
    }  
}  
  
apply plugin: 'org.springframework.boot'
```

Spring Boot CLI

The Spring Boot CLI is a command line tool that you can use if you want to quickly develop a Spring application. It lets you run Groovy scripts, which means that you have a familiar Java-like syntax without so much boilerplate code. You can also bootstrap a new project or write your own command for it.

The Spring Boot CLI (Command-Line Interface) can be installed manually by using SDKMAN! (the SDK Manager) or by using Homebrew or MacPorts if you are an OSX user.



Manual Installation

You can download the Spring CLI distribution from the Spring software repository:

- [spring-boot-cli-2.7.3-bin.zip](#)
- [spring-boot-cli-2.7.3-bin.tar.gz](#)

Once downloaded, follow the INSTALL.txt instructions from the unpacked archive. In summary, there is a spring script(spring.bat for Windows) in a bin/ directory in the .zip file. Alternatively, you can use java -jar with the .jar file (the script helps you to be sure that the classpath is set correctly).

After downloading, extract the zip file. It contains a bin folder, in which spring setup is stored. We can use it to execute Spring Boot application.

CLI executes groovy files. So, first, we need to create a groovy file for Spring Boot application.

Running Applications with the CLI

You can compile and run Groovy source code by using the run command. The Spring Boot CLI is completely self-contained, so you do not need any external Groovy installation.

The following example shows a “hello world” web application written in Groovy:

```
@RestController
class WebApplication {

    @RequestMapping("/")
    String home() {
        "Hello World!"
    }
}
```

To compile and run the application, type the following command:

```
$ spring run hello.groovy
```

To pass command-line arguments to the application, use -- to separate the commands from the “spring” command arguments, as shown in the following example:

```
$ spring run hello.groovy-----server.port=9000
```



To set JVM command line arguments, you can use the JAVA_OPTS environment variable, as shown in the following example:

```
$ JAVA_OPTS=-Xmx1024m spring run hello.groovy
```

Spring Boot Application

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

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

Observe the following code for a better understanding –

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

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

Build Systems

In Spring Boot, choosing a build system is an important task. We recommend Maven or Gradle as they provide a good support for dependency management. Spring does not support well other build systems.

Dependency Management

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



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

Maven Dependency

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

```
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>1.5.8.RELEASE</version>
</parent>
```

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

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

Gradle Dependency

We can import the Spring Boot Starters dependencies directly into build.gradle file. We do not need Spring Boot startParent dependency like Maven for Gradle. Observe the code given below:

```
buildscript {
  ext {
    springBootVersion = '1.5.8.RELEASE'
  }
  repositories {
    mavenCentral()
  }
  dependencies {
    classpath("org.springframework.boot:spring-boot-gradle-plugin:${springBootVersion}")
  }
}
```



Similarly, in Gradle, we need not specify the Spring Boot version number for dependencies. Spring Boot automatically configures the dependency based on the version.

```
dependencies {
    compile('org.springframework.boot:spring-boot-starter-web')
}
```

Code Structure

Spring Boot does not have any code layout to work with. However, there are some best practices that will help us. This chapter talks about them in detail.

Default package

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

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

Typical Layout

The typical layout of Spring Boot application is shown in the image given below:

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

The Application.java file should declare the main method along with @SpringBootApplication. Observe the code given below for a better understanding:



```
package com.tutorialspoint.myproject;

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

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

Beans and Dependency Injection

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

If you followed the Spring Boot typical layout, no need to specify any arguments for `@ComponentScan` annotation. All component class files are automatically registered with Spring Beans.

The following example provides an idea about Auto wiring the Rest Template object and creating a Bean for the same:

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

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



```
package com.edubridge.demo;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;
import org.springframework.web.client.RestTemplate;

@SpringBootApplication
public class DemoApplication {
    @Autowired
    RestTemplate restTemplate;

    public static void main(String[] args) {
        SpringApplication.run(DemoApplication.class, args);
    }
    @Bean
    public RestTemplate getRestTemplate() {
        return new RestTemplate();
    }
}
```

Spring Boot - Application Properties

Application Properties support us to work in different environments.

Command Line Properties

Spring Boot application converts the command line properties into Spring Boot Environment properties. Command line properties take precedence over the other property sources. By default, Spring Boot uses the 8080 port number to start the Tomcat. Let us learn how to change the port number by using command line properties.

Step 1: After creating an executable JAR file, run it by using the command `java -jar <JARFILE>`.

Step 2: Use the command given in the screenshot given below to change the port number for Spring Boot application by using command line properties.



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

Note: You can provide more than one application properties by using the delimiter –.

Properties File

Properties files are used to keep ‘N’ number of properties in a single file to run the application in a different environment. In Spring Boot, properties are kept in the application.properties file under the classpath.

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

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

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

YAML File

Spring Boot supports YAML based properties configurations to run the application. Instead of application.properties, we can use application.yml file. This YAML file also should be kept inside the classpath. The sample application.yml file is given below –

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

Externalized Properties

Spring Boot allows you to externalize your configuration so you can work with the same application code in different environments. You can use properties files, YAML files, environment variables and command-line arguments to externalize configuration. Property values can be injected directly into your beans using the @Value annotation, accessed via Spring’s Environment abstraction or bound to structured objects.



Instead of keeping the properties file under classpath, we can keep the properties in different location or path. While running the JAR file, we can specify the properties file path. You can use the following command to specify the location of properties file while running the JAR:

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



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

Use of @Value Annotation

The @Value annotation is used to read the environment or application property value in Java code. The syntax to read the property value is shown below:

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

Look at the following example that shows the syntax to read the spring.application.name property value in Java variable by using @Value annotation.

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

Observe the code given below for a better understanding:



```
import org.springframework.beans.factory.annotation.Value;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;
```

```
@SpringBootApplication
@RestController
public class DemoApplication {
    @Value("${spring.application.name}")
    private String name;
    public static void main(String[] args) {
        SpringApplication.run(DemoApplication.class, args);
    }
    @RequestMapping(value = "/")
    public String name() {
        return name;
    }
}
```

Note: If the property is not found while running the application, Spring Boot throws the Illegal Argument exception as Could not resolve placeholder 'spring.application.name' in value "\${spring.application.name}".

To resolve the placeholder issue, we can set the default value for the property using the syntax given below –

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

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

Logging

Spring Boot uses Apache Commons logging for all internal logging. Spring Boot's default configurations provide support for the use of Java Util Logging, Log4j2, and Logback. Using these, we can configure the console logging as well as file logging.

If you are using Spring Boot Starters, Logback will provide a good support for logging. Besides, Logback also provides a use of good support for Common Logging, Util Logging, Log4J, and SLF4J.



Log Format

The default Spring Boot Log format is shown in the screenshot given below.

```
2017-11-26 09:30:27.873 INFO 5040 --- [main] s.b.c.e.t.TomcatEmbeddedServletContainer : Tomcat initialized with port(s): 8080 (http)
2017-11-26 09:30:27.895 INFO 5040 --- [main] o.apache.catalina.core.StandardService : Starting service [Tomcat]
2017-11-26 09:30:27.898 INFO 5040 --- [main] org.apache.catalina.core.StandardEngine : Starting Servlet Engine: Apache Tomcat/8.5.23
2017-11-26 09:30:28.040 INFO 5040 --- [ost-startStop-1] o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring embedded WebApplicationContext
2017-11-26 09:30:28.040 INFO 5040 --- [ost-startStop-1] o.s.web.context.ContextLoader : Root WebApplicationContext: initialization completed in 2759 ms
```

which gives you the following information –

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

Console Log Output

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

If you have to enable the debug level log, add the debug flag on starting your application using the command shown below –

```
java -jar demo.jar --debug
```

You can also add the debug mode to your application.properties file as shown here –

```
debug = true
```

File Log Output

By default, all logs will print on the console window and not in the files. If you want to print the logs in a file, you need to set the property logging.file or logging.path in the application.properties file.

You can specify the log file path using the property shown below. Note that the log file name is spring.log.



```
logging.path = /var/tmp/
```

You can specify the own log file name using the property shown below –

```
logging.path = /var/tmp/
```

Log Levels

Spring Boot supports all logger levels such as “TRACE”, “DEBUG”, “INFO”, “WARN”, “ERROR”, “FATAL”, “OFF”. You can define Root logger in the application.properties file as shown below:

```
logging.level.root = WARN
```

Note: Logback does not support “FATAL” level log. It is mapped to the “ERROR” level log.

Configure Logback

Logback supports XML based configuration to handle Spring Boot Log configurations. Logging configuration details are configured in logback.xml file. The logback.xml file should be placed under the classpath.

You can configure the ROOT level log in Logback.xml file using the code given below:

```
<?xml version = "1.0" encoding = "UTF-8"?>
<configuration>
  <root level = "INFO">
    </root>
  </configuration>
```

You can configure the console appender in Logback.xml file given below.



```
<?xml version = "1.0" encoding = "UTF-8"?>
<configuration>
  <appender name = "STDOUT" class = "ch.qos.logback.core.ConsoleAppender"></appender>
  <root level = "INFO">
    <appender-ref ref = "STDOUT"/>
  </root>
</configuration>
```

You can configure the file appender in Logback.xml file using the code given below. Note that you need to specify theLog file path insider the file appender.

```
<?xml version = "1.0" encoding = "UTF-8"?>
<configuration>
  <appender name = "FILE" class = "ch.qos.logback.core.FileAppender">
    <File>/var/tmp/mylog.log</File>
  </appender>
  <root level = "INFO">
    <appender-ref ref = "FILE"/>
  </root>
</configuration>
```

You can define the Log pattern in logback.xml file using the code given below. You can also define the set of supportedlog patterns inside the console or file log appender using the code given below:

```
<pattern>[%d{yyyy-MM-dd'T'HH:mm:ss.sss'Z'}] [%C] [%t] [%L] [%-5p] %m%n</pattern>
```

The code for complete logback.xml file is given below. You have to place this in the class path.



```
<?xml version = "1.0" encoding = "UTF-8"?>
<configuration>
  <appender name = "STDOUT" class = "ch.qos.logback.core.ConsoleAppender">
    <encoder>
      <pattern>[%d{yyyy-MM-dd'T'HH:mm:ss'Z'}} [%C] [%t] [%L] [%-5p] %m%n</pattern>
    </encoder>
  </appender>
  <appender name = "FILE" class = "ch.qos.logback.core.FileAppender">
    <File>/var/tmp/mylog.log</File>
    <encoder>
      <pattern>[%d{yyyy-MM-dd'T'HH:mm:ss'Z'}} [%C] [%t] [%L] [%-5p] %m%n</pattern>
    </encoder>
  </appender>
  <root level = "INFO">
    <appender-ref ref = "FILE"/>
    <appender-ref ref = "STDOUT"/>
  </root>
</configuration>
```

The code given below shows how to add the slf4j logger in Spring Boot main class file.

```
package com.edubridge.demo;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
@SpringBootApplication
public class DemoApplication {
  private static final Logger logger = LoggerFactory.getLogger(DemoApplication.class);
  public static void main(String[] args) {
    logger.info("this is a info message");
    logger.warn("this is a warn message");
    logger.error("this is a error message");
    SpringApplication.run(DemoApplication.class, args);
  }
}
```

The output that you can see in the console window is shown here:



```
[main] [14] [INFO ] this is a info message  
[main] [15] [WARN ] this is a warn message  
[main] [16] [ERROR] this is a error message
```

The output that you can see in the log file is shown here:

```
[main] [14] [INFO ] this is a info message  
[main] [15] [WARN ] this is a warn message  
[main] [16] [ERROR] this is a error message
```

Spring Boot CRUD Operations

What is the CRUD operation?

The CRUD stands for Create, Read/Retrieve, Update, and Delete. These are the four basic functions of the persistence storage.

The CRUD operation can be defined as user interface conventions that allow view, search, and modify information through computer-based forms and reports. CRUD is data-oriented and the standardized use of HTTP action verbs. HTTP has a few important verbs.

- POST: Creates a new resource
- GET: Reads a resource
- PUT: Updates an existing resource
- DELETE: Deletes a resource

Within a database, each of these operations maps directly to a series of commands. However, their relationship with a RESTful API is slightly more complex.

Standard CRUD Operation

- CREATE Operation: It performs the INSERT statement to create a new record.
- READ Operation: It reads table records based on the input parameter.
- UPDATE Operation: It executes an update statement on the table. It is based on the input parameter.
- DELETE Operation: It deletes a specified row in the table. It is also based on the input parameter.

How CRUD Operations Works

CRUD operations are at the foundation of the most dynamic websites. Therefore, we should differentiate CRUD from the HTTP action verbs.



Suppose, if we want to create a new record, we should use HTTP action verb POST. To update a record, we should use the PUT verb. Similarly, if we want to delete a record, we should use the DELETE verb. Through CRUD operations, users and administrators have the right to retrieve, create, edit, and delete records online.

We have many options for executing CRUD operations. One of the most efficient choices is to create a set of stored procedures in SQL to execute operations.

The CRUD operations refer to all major functions that are implemented in relational database applications. Each letter of the CRUD can map to a SQL statement and HTTP methods.

| Operation | SQL | HTTP verbs | RESTful Web Service |
|-----------|--------|----------------|---------------------|
| Create | INSERT | PUT/POST | POST |
| Read | SELECT | GET | GET |
| Update | UPDATE | PUT/POST/PATCH | PUT |
| Delete | DELETE | DELETE | DELETE |

Spring Boot CrudRepository

Spring Boot provides an interface called CrudRepository that contains methods for CRUD operations. It is defined in the package org.springframework.data.repository. It extends the Spring Data Repository interface. It provides generic Crud operation on a repository. If we want to use CrudRepository in an application, we have to create an interface and extend the CrudRepository.

Syntax:

```
public interface CrudRepository<T,ID> extends Repository<T,ID>
```

where,

- T is the domain type that repository manages.
- ID is the type of the id of the entity that repository manages.

For example:

```
public interface StudentRepository extends CrudRepository<Student, Integer>
{
}
```



In the above example, we have created an interface named `StudentRepository` that extends `CrudRepository`. Where `Student` is the repository to manage, and `Integer` is the type of `Id` that is defined in the `Student` repository.

Spring Boot JpaRepository

`JpaRepository` provides JPA related methods such as flushing, persistence context, and deletes a record in a batch. It is defined in the package `org.springframework.data.jpa.repository`. `JpaRepository` extends both `CrudRepository` and `PagingAndSortingRepository`.

For example:

```
public interface BookDAO extends JpaRepository
{
}
```

PagingAndSortingRepository

This repository interface, which extends `CrudRepository`.

```
public interface PagingAndSortingRepository<T, ID extends Serializable>
extends CrudRepository<T, ID> {

    Iterable<T> findAll(Sort sort);

    Page<T> findAll(Pageable pageable);
}
```

This interface provides a method `findAll(Pageable pageable)`, which is the key to implementing Pagination.

When using `Pageable`, we create a `Pageable` object with certain properties and we've to specify at least:

1. Page size
2. Current page number
3. Sorting

So, let's assume that we want to show the first page of a result set sorted by `lastName`, ascending, having no



more than five records each. This is how we can achieve this using a PageRequest and a Sort definition:

```
Sort sort = new Sort(new Sort.Order(Direction.ASC, "lastName"));
Pageable pageable = new PageRequest(0, 5, sort);
```

Passing the pageable object to the Spring data query will return the results in question (the first parameter of PageRequest is zero-based).

Why should we use these interfaces?

The interfaces allow Spring to find the repository interface and create proxy objects for that. It provides methods that allow us to perform some common operations. We can also define custom methods as well.



CrudRepository vs. JpaRepository

| CrudRepository | JpaRepository |
|--|--|
| CrudRepository does not provide any method for pagination and sorting. | JpaRepository extends PagingAndSortingRepository. It provides all the methods for implementing the pagination. |
| It works as a marker interface. | JpaRepository extends both CrudRepository and PagingAndSortingRepository . |



| | |
|--|---|
| It provides CRUD function only. For example findById() , findAll() , etc. | It provides some extra methods along with the method of PagingAndSortingRepository and CrudRepository. For example, flush() , deleteInBatch() . |
| It is used when we do not need the functions provided by JpaRepository and PagingAndSortingRepository. | It is used when we want to implement pagination and sorting functionality in an application. |

Spring Boot CRUD Operation Example

Let's set up a Spring Boot application and perform CRUD operation.

Step 1: Open Spring Initializr <http://start.spring.io>.

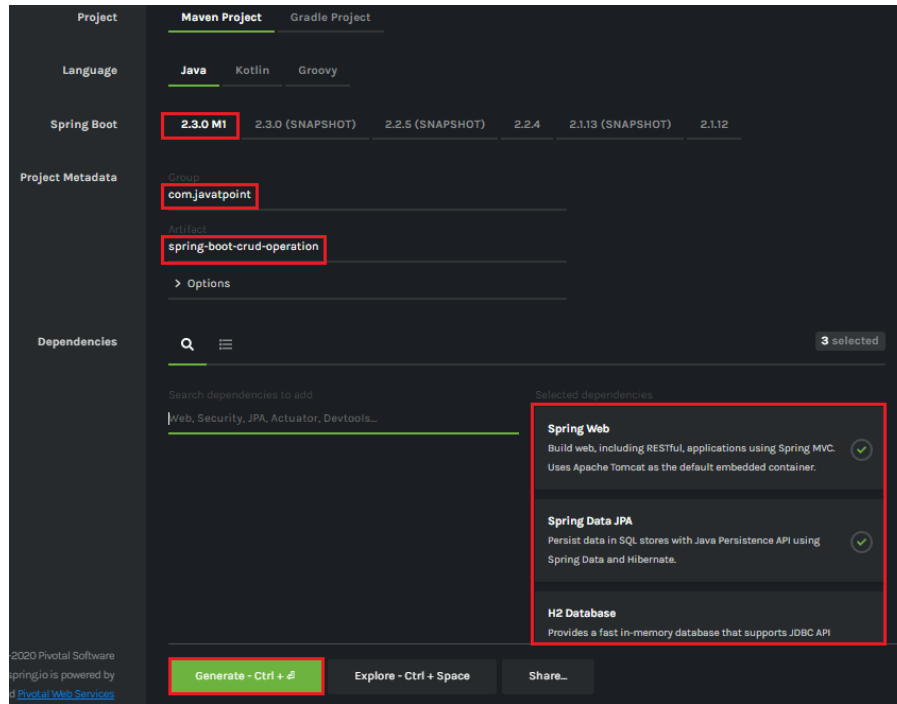
Step 2: Select the Spring Boot version 2.3.0.M1.

Step 2: Provide the Group name. We have provided com.edubridge.

Step 3: Provide the Artifact Id. We have provided spring-boot-crud-operation.

Step 5: Add the dependencies Spring Web, Spring Data JPA, and H2 Database.

Step 6: Click on the Generate button. When we click on the Generate button, it wraps the specifications in a Jar file and downloads it to the local system.



Step 7: Extract the Jar file and paste it into the STS workspace.

Step 8: Import the project folder into STS.

File -> Import -> Existing Maven Projects -> Browse -> Select the folder spring-boot-crud-operation ->

FinishIt takes some time to import.

Step 9: Create a package with the name com.edubridge.model in the folder src/main/java.

Step 10: Create a model class in the package com.edubridge.model. We have created a model class with the nameBooks. In the Books class, we have done the following:

- Define four variable bookid, bookname, author, and
- Generate Getters and Setters.
- Right-click on the file -> Source -> Generate Getters and Setters.
- Mark the class as an Entity by using the annotation @Entity.
- Mark the class as Table name by using the annotation @Table.
- Define each variable as Column by using the annotation @Column.

- **Books.java**

```
package com.javatpoint.model;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.Table;
//mark class as an Entity
@Entity
//defining class name as Table name
@Table
public class Books
{
//Defining book id as primary key
@Id
@Column
private int bookid;
@Column
private String bookname;
@Column
private String author;
@Column
private int price;
public int getBookid()
{
return bookid;
}
public void setBookid(int bookid)
{
this.bookid = bookid;
}
public String getBookname()
{
return bookname;
}
public void setBookname(String bookname)
{
this.bookname = bookname;
}
```



```
public String getAuthor()
{
    return author;
}
public void setAuthor(String author)
{
    this.author = author;
}
public int getPrice()
{
    return price;
}
public void setPrice(int price)
{
    this.price = price;
}
}
```

Step 11: Create a package with the name `com.edubridge.controller` in the folder `src/main/java`.

Step 12: Create a Controller class in the package `com.javatpoint.controller`. We have created a controller class with the name `BooksController`. In the `BooksController` class, we have done the following:

- Mark the class as `RestController` by using the annotation `@RestController`.
- Autowire the `BooksService` class by using the annotation `@Autowired`.
- Define the following methods:
 - `getAllBooks()`: It returns a List of all Books.
 - `getBooks()`: It returns a book detail that we have specified in the path variable. We have passed `bookid` as an argument by using the annotation `@PathVariable`. The annotation indicates that a method parameter should be bound to a URI template variable.
 - `deleteBook()`: It deletes a specific book that we have specified in the path variable.
 - `saveBook()`: It saves the book detail. The annotation `@RequestBody` indicates that a method parameter should be bound to the body of the web request.
 - `update()`: The method updates a record. We must specify the record in the body, which we want to update. To achieve the same, we have used the annotation `@RequestBody`.



BooksController.java

```
package com.javatpoint.controller;
import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.DeleteMapping;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.PutMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RestController;
import com.javatpoint.model.Books;
import com.javatpoint.service.BooksService;
//mark class as Controller
@RestController
public class BooksController
{
    //autowire the BooksService class
    @Autowired
    BooksService booksService;
    //creating a get mapping that retrieves all the books detail from the database
    @GetMapping("/book")
    private List<Books> getAllBooks()
    {
        return booksService.getAllBooks();
    }
    //creating a get mapping that retrieves the detail of a specific book
    @GetMapping("/book/{bookid}")
    private Books getBooks(@PathVariable("bookid") int bookid)
    {
        return booksService.getBooksById(bookid);
    }
    //creating a delete mapping that deletes a specified book
    @DeleteMapping("/book/{bookid}")
    private void deleteBook(@PathVariable("bookid") int bookid)
    {
        booksService.delete(bookid);
    }
}
```



```
//creating post mapping that post the book detail in the database
@PostMapping("/books")
private int saveBook(@RequestBody Books books)
{
    booksService.saveOrUpdate(books);
    return books.getBookid();
}

//creating put mapping that updates the book detail
@PutMapping("/books")
private Books update(@RequestBody Books books)
{
    booksService.saveOrUpdate(books);
    return books;
}
}
```

Step 13: Create a package with the name com.javatpoint.service in the folder src/main/java.

Step 14: Create a Service class. We have created a service class with the name BooksService in the package com.edubridge.service.

BooksService.java

```
package com.javatpoint.service;
import java.util.ArrayList;
import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import com.javatpoint.model.Books;
import com.javatpoint.repository.BooksRepository;
//defining the business logic
@Service
public class BooksService
{
    @Autowired
    BooksRepository booksRepository;
    //getting all books record by using the method findaAll() of CrudRepository
```



```
public List<Books> getAllBooks()
{
    List<Books> books = new ArrayList<Books>();
    booksRepository.findAll().forEach(books1 -> books.add(books1));
    return books;
}
//getting a specific record by using the method findById() of CrudRepository
public Books getBooksById(int id)
{
    return booksRepository.findById(id).get();
}
//saving a specific record by using the method save() of CrudRepository
public void saveOrUpdate(Books books)
{
    booksRepository.save(books);
}
//deleting a specific record by using the method deleteById() of CrudRepository
public void delete(int id)
{
    booksRepository.deleteById(id);
}
//updating a record
public void update(Books books, int bookid)
{
    booksRepository.save(books);
}
}
```

Step 15: Create a package with the name `com.javatpoint.repository` in the folder `src/main/java`.

Step 16: Create a Repository interface. We have created a repository interface with the name `BooksRepository` in the package `com.javatpoint.repository`. It extends the `Crud Repository` interface.



BooksRepository.java

```
package com.javatpoint.repository;
import org.springframework.data.repository.CrudRepository;
import com.javatpoint.model.Books;
//repository that extends CrudRepository
public interface BooksRepository extends CrudRepository<Books, Integer>
{
}
```

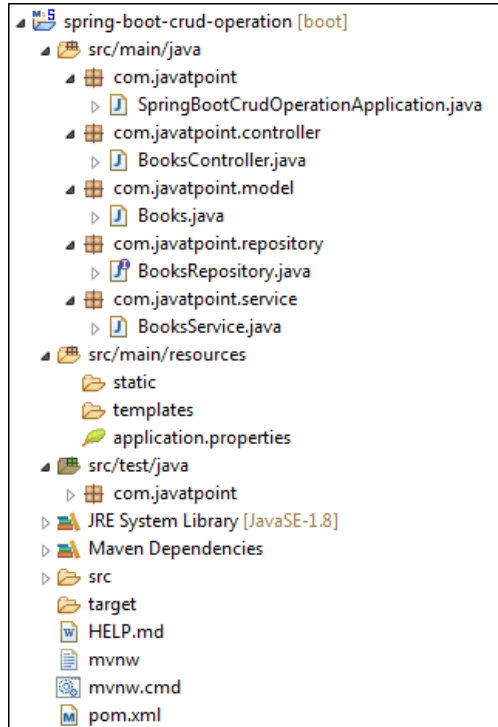
Now we will configure the datasource URL, driver class name, username, and password, in the application.propertiesfile.

Step 17: Open the application.properties file and configure the following

properties.**application.properties**

```
spring.datasource.url=jdbc:h2:mem:books_data
spring.datasource.driverClassName=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=
spring.jpa.database-platform=org.hibernate.dialect.H2Dialect
#enabling the H2 console
spring.h2.console.enabled=true
```

After creating all the classes and packages, the project directory looks like the following.



Now we will run the application.

Step 18: Open SpringBootCrudOperationApplication.java file and run it as Java Application.

SpringBootCrudOperationApplication.java

```
package com.javatpoint;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
@SpringBootApplication
public class SpringBootCrudOperationApplication
{
    public static void main(String[] args)
    {
        SpringApplication.run(SpringBootCrudOperationApplication.class, args);
    }
}
```



Step 19: Open the Postman and do the following:

- Select the POST
- Invoke the URL `http://localhost:8080/books`.
- Select the Body
- Select the Content-Type JSON (`application/json`).
- Insert the data. We have inserted the following data in the Body:

```
{
  "bookid": "5433",
  "bookname": "Core and Advance Java",
  "author": "R. Nageswara Rao",
  "price": "800"
}
```

- Click on the Send

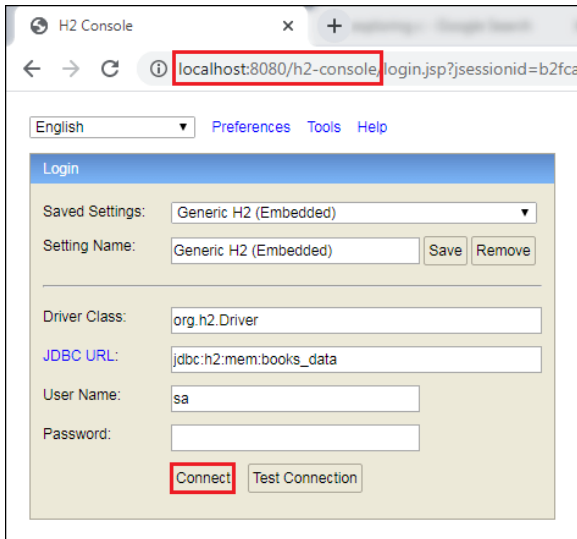
When the request is successfully executed, it shows the Status:200 OK. It means the record has been successfully inserted in the database.

Similarly, we have inserted the following data.

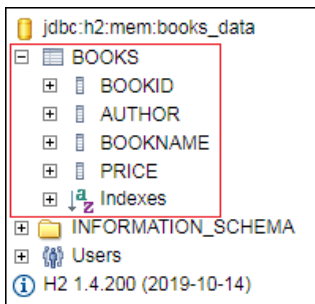
```
{
  "bookid": "0982",
  "bookname": "Programming with Java",
  "author": "E. Balagurusamy",
  "price": "350"
}
{
  "bookid": "6321",
  "bookname": "Data Structures and Algorithms in Java",
  "author": "Robert Lafore",
  "price": "590"
}
{
  "bookid": "5433",
  "bookname": "Effective Java",
  "author": "Joshua Bloch",
  "price": "670"
}
```

Let's access the H2 console to see the data.

Step 20: Open the browser and invoke the URL <http://localhost:8080/h2-console>. Click on the Connect button, as shown below.



After clicking on the Connect button, we see the Books table in the database, as shown below.



Step 21: Click on the Books table and then click on the Run button. The table shows the data that we have inserted in the body.

SQL statement:

SELECT * FROM BOOKS

SELECT * FROM BOOKS BOOKS;

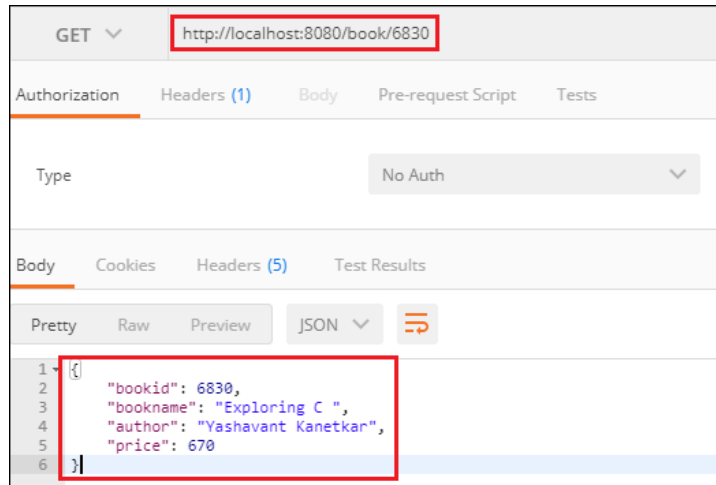
| BOOKID | AUTHOR | BOOKNAME | PRICE |
|--------|--------------------|--|-------|
| 982 | E. Balagurusamy | Programming with Java | 350 |
| 5433 | R. Nageswara Rao | Core and Advance Java | 800 |
| 6321 | Robert Lafore | Data Structures and Algorithms in Java | 590 |
| 6830 | Yashavant Kanetkar | Exploring C | 670 |

 (4 rows, 23 ms)

Step 22: Open the Postman and send a GET request with the URL <http://localhost:8080/books>. It returns the data that we have inserted in the database.

```
[
  {
    "bookid": 982,
    "bookname": "Programming with Java",
    "author": "E. Balagurusamy",
    "price": 350
  },
  {
    "bookid": 5433,
    "bookname": "Core and Advance Java",
    "author": "R. Nageswara Rao",
    "price": 800
  },
  {
    "bookid": 6321,
    "bookname": "Data Structures and Algorithms in Java",
    "author": "Robert Lafore",
    "price": 590
  },
  {
    "bookid": 6830,
    "bookname": "Exploring C ",
    "author": "Yashavant Kanetkar",
    "price": 670
  }
]
```

Let's send a GET request with the URL <http://localhost:8080/book/{bookid}>. We have specified the bookid 6830. It returns the detail of the book whose id is 6830.



Similarly, we can also send a DELETE request to delete a record. Suppose we want to delete a book record whose id is 5433.

Select the DELETE method and invoke the URL `http://localhost:8080/book/5433`. Again, execute the Select query in the H2 console. We see that the book whose id is 5433 has been deleted from the database.

SELECT * FROM BOOKS;

| BOOKID | AUTHOR | BOOKNAME | PRICE |
|--------|--------------------|--|-------|
| 982 | E. Balagurusamy | Programming with Java | 350 |
| 6321 | Robert Lafore | Data Structures and Algorithms in Java | 590 |
| 6830 | Yashavant Kanetkar | Exploring C | 670 |

(3 rows, 23 ms)

Similarly, we can also update a record by sending a PUT request. Let's update the price of the book whose id is 6321.

- Select the PUT
- In the request body, paste the record which you want to update and make the changes. In our case, we want to update the record of the book whose id is 6321. In the following record, we have changed the price of the book.



```
{
  "bookid": "6321",
  "bookname": "Data Structures and Algorithms in Java",
  "author": "Robert Lafore",
  "price": "500"
}
```

- Click on the Send

Now, move to the H2 console and see the changes have reflected or not. We see that the price of the book has been changed, as shown below.

SELECT * FROM BOOKS;

| BOOKID | AUTHOR | BOOKNAME | PRICE |
|--------|--------------------|--|-------|
| 982 | E. Balagurusamy | Programming with Java | 350 |
| 6321 | Robert Lafore | Data Structures and Algorithms in Java | 500 |
| 6830 | Yashavant Kanetkar | Exploring C | 390 |

(3 rows, 20 ms)

Activity One

Create an application using CRUD operations. First, create a table as shown below in MySQL database. It has 4 fields: id, name, salary, and designation. Here, id is auto incremented which is generated by the sequence.

| Column Name | Data Type | Nullable | Default | Primary Key |
|-------------|----------------|----------|---------|-------------|
| ID | NUMBER | No | - | 1 |
| NAME | VARCHAR2(4000) | Yes | - | - |
| SALARY | NUMBER | Yes | - | - |
| DESIGNATION | VARCHAR2(4000) | Yes | - | - |
| 1 - 4 | | | | |

Follow the instruction to get the application as shown

1. Add dependencies to pom.xml file.

Pom.xml



```
<!-- https://mvnrepository.com/artifact/org.springframework/spring-webmvc -->
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-webmvc</artifactId>
  <version>5.1.1.RELEASE</version>
</dependency>

<!-- https://mvnrepository.com/artifact/org.apache.tomcat/tomcat-jasper -->
<dependency>
  <groupId>org.apache.tomcat</groupId>
  <artifactId>tomcat-jasper</artifactId>
  <version>9.0.12</version>
</dependency>

<!-- https://mvnrepository.com/artifact/javax.servlet/javax.servlet-api -->
<dependency>
  <groupId>javax.servlet</groupId>
  <artifactId>servlet-api</artifactId>
  <version>3.0-alpha-1</version>
</dependency>

<!-- https://mvnrepository.com/artifact/javax.servlet/jstl -->
<dependency>
  <groupId>javax.servlet</groupId>
  <artifactId>jstl</artifactId>
  <version>1.2</version>
</dependency>

<!-- https://mvnrepository.com/artifact/mysql/mysql-connector-java -->
<dependency>
  <groupId>mysql</groupId>
  <artifactId>mysql-connector-java</artifactId>
  <version>8.0.11</version>
</dependency>

<!-- https://mvnrepository.com/artifact/org.springframework/spring-jdbc -->
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-jdbc</artifactId>
  <version>5.1.1.RELEASE</version>
</dependency>
```




2. Create the bean class

Here, the bean class contains the variables (along setter and getter methods) corresponding to the fields exist in the database.

Emp.java

```
package com.javatpoint.beans;

public class Emp {
    private int id;
    private String name;
    private float salary;
    private String designation;
    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 float getSalary() {
        return salary;
    }
    public void setSalary(float salary) {
        this.salary = salary;
    }
    public String getDesignation() {
        return designation;
    }
    public void setDesignation(String designation) {
        this.designation = designation;
    }
}
```

3. Create the controller class

EmpController.java

```
package com.javatpoint.controllers;
import java.util.List;
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 com.javatpoint.beans.Emp;
import com.javatpoint.dao.EmpDao;
@Controller
public class EmpController {
    @Autowired
    EmpDao dao;//will inject dao from XML file

    /*It displays a form to input data, here "command" is a reserved request attribute
    *which is used to display object data into form
    */
    @RequestMapping("/empform")
    public String showform(Model m){
        m.addAttribute("command", new Emp());
        return "empform";
    }
    /*It saves object into database. The @ModelAttribute puts request data
    * into model object. You need to mention RequestMethod.POST method
    * because default request is GET*/
    @RequestMapping(value="/save",method = RequestMethod.POST)
    public String save(@ModelAttribute("emp") Emp emp){
        dao.save(emp);
        return "redirect:/viewemp";//will redirect to viewemp request mapping
    }
}
```



```

/* It provides list of employees in model object */
@RequestMapping("/viewemp")
public String viewemp(Model m){
    List<Emp> list=dao.getEmployees();
    m.addAttribute("list",list);
    return "viewemp";
}
/* It displays object data into form for the given id.
 * The @PathVariable puts URL data into variable.*/
@RequestMapping(value="/editemp/{id}")
public String edit(@PathVariable int id, Model m){
    Emp emp=dao.getEmpById(id);
    m.addAttribute("command",emp);
    return "empeditform";
}
/* It updates model object. */
@RequestMapping(value="/editsave",method = RequestMethod.POST)
public String editsave(@ModelAttribute("emp") Emp emp){
    dao.update(emp);
    return "redirect:/viewemp";
}
/* It deletes record for the given id in URL and redirects to /viewemp */
@RequestMapping(value="/deleteemp/{id}",method = RequestMethod.GET)
public String delete(@PathVariable int id){
    dao.delete(id);
    return "redirect:/viewemp";
}
}

```

4. Create the DAO class

Let's create a DAO class to access the required data from the database.

EmpDao.java



```

package com.javatpoint.dao;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.List;
import org.springframework.jdbc.core.BeanPropertyRowMapper;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.RowMapper;import
com.javatpoint.beans.Emp;
public class EmpDao {
    JdbcTemplate template;

    public void setTemplate(JdbcTemplate template) {
        this.template = template;
    }
    public int save(Emp p){
        String sql="insert into Emp99(name,salary,designation)
values('"+p.getName()+"','"+p.getSalary()+"','"+p.getDesignation()+"'";
        return template.update(sql);
    }
    public int update(Emp p){
        String sql="update Emp99 set name='"+p.getName()+"',
salary='"+p.getSalary()+"',designation='"+p.getDesignation()+"' where id='"+p.getId()+"'";
        Return template.update(sql);
    }
    public int delete(int id){
        String sql="delete from Emp99 where id="+id+"";
        return template.update(sql); }
    public Emp getEmpById(int id){
        String sql="select * from Emp99 where id=?";
        return template.queryForObject(sql, new Object[]{id},new BeanPropertyRowMapper<Emp>(Emp.class)); }
    public List<Emp> getEmployees(){
return template.query("select * from Emp99",new RowMapper<Emp>(){
    public Emp mapRow(ResultSet rs, int row) throws SQLException {
        Emp e=new Emp();
        e.setId(rs.getInt(1));
        e.setName(rs.getString(2));
        e.setSalary(rs.getFloat(3));
        e.setDesignation(rs.getString(4));
        return e;
    } }); } }
    
```

5. Provide the entry of controller in the web.xml file

web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://java.sun.com/xml/ns/javaee" xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd" id="WebApp_ID" version="3.0">
  <display-name>SpringMVC</display-name>
  <servlet>
    <servlet-name>spring</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
  </servlet>
  <servlet-mapping>
    <servlet-name>spring</servlet-name>
    <url-pattern>/</url-pattern>
  </servlet-mapping>
</web-app>
```

6. Define the bean in the xml file

Spring-servlet.xml

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



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

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

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

<bean id="dao" class="com.javatpoint.dao.EmpDao">
<property name="template" ref="jt"></property>
</bean>
</beans>
```

7. Create the requested page

Index.jsp

```
<a href="empform">Add Employee</a>
<a href="viewemp">View Employees</a>
```

8. Create the other view components

Empform.jsp



```
<%@ taglib uri="http://www.springframework.org/tags/form" prefix="form"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
```

```
<h1>Add New Employee</h1>
<form:form method="post" action="save">
<table >
<tr>
<td>Name : </td>
<td><form:input path="name" /></td>
</tr>
<tr>
<td>Salary :</td>
<td><form:input path="salary" /></td>
</tr>
<tr>
<td>Designation :</td>
<td><form:input path="designation" /></td>
</tr>
<tr>
<td></td>
<td><input type="submit" value="Save" /></td>
</tr>
</table>
</form:form>
```

empeditform.jsp

Here "/SpringMVCCRUDSimple" is the project name, change this if you have different project name. For live application, you can provide full URL.



```
<%@ taglib uri="http://www.springframework.org/tags/form" prefix="form"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
```

```
<h1>Edit Employee</h1>
<form:form method="POST" action="/SpringMVCCRUDSimple/editsave">
<table >
<tr>
<td></td>
<td><form:hidden path="id" /></td>
</tr>
<tr>
<td>Name : </td>
<td><form:input path="name" /></td>
</tr>
<tr>
<td>Salary :</td>
<td><form:input path="salary" /></td>
</tr>
<tr>
<td>Designation :</td>
<td><form:input path="designation" /></td>
</tr>
<tr>
<td> </td>
<td><input type="submit" value="Edit Save" /></td>
</tr>
</table>
</form:form>
```

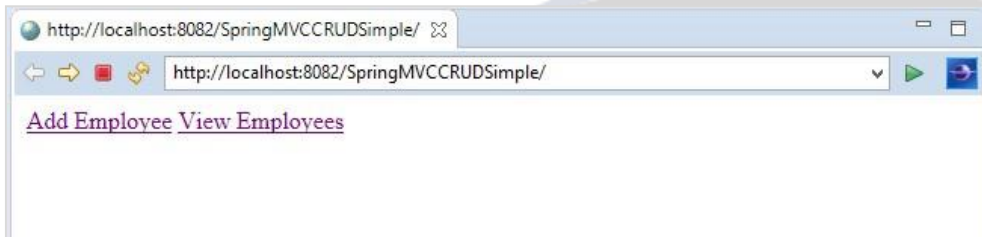
viewemp.jsp



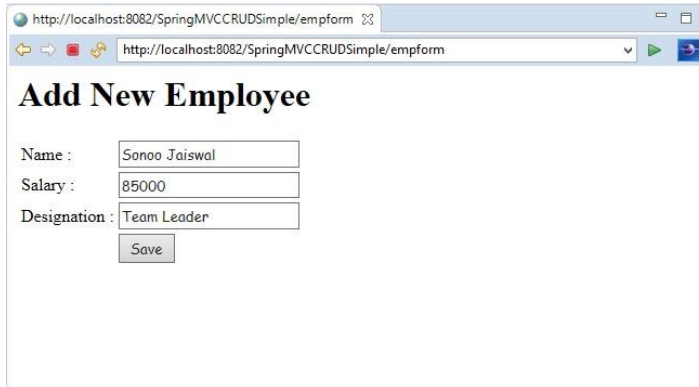
```
<%@ taglib uri="http://www.springframework.org/tags/form" prefix="form"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>

<h1>Employees List</h1>
<table border="2" width="70%" cellpadding="2">
<tr><th>Id</th><th>Name</th><th>Salary</th><th>Designation</th><th>Edit</th><th>Delete</th></tr>
<c:forEach var="emp" items="${list}">
<tr>
<td>${emp.id}</td>
<td>${emp.name}</td>
<td>${emp.salary}</td>
<td>${emp.designation}</td>
<td><a href="editemp/${emp.id}">Edit</a></td>
<td><a href="deleteemp/${emp.id}">Delete</a></td>
</tr>
</c:forEach>
</table>
<br/>
<a href="empform">Add New Employee</a>
```

Output:



On clicking Add Employee, you will see the following form.



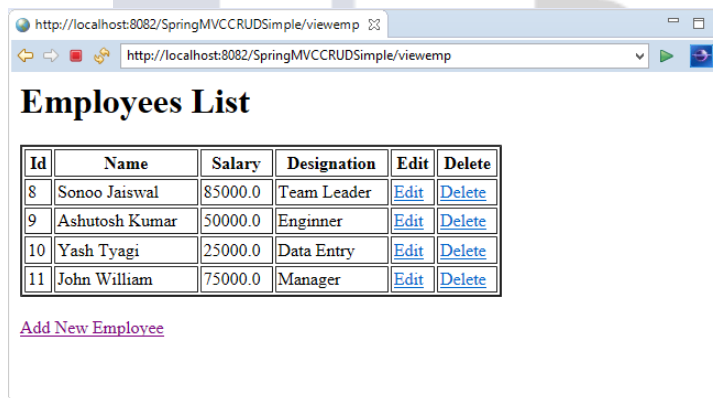
Add New Employee

Name :

Salary :

Designation :

Fill the form and click Save to add the entry into the database.



Employees List

| Id | Name | Salary | Designation | Edit | Delete |
|----|----------------|---------|-------------|----------------------|------------------------|
| 8 | Sonoo Jaiswal | 85000.0 | Team Leader | Edit | Delete |
| 9 | Ashutosh Kumar | 50000.0 | Enginner | Edit | Delete |
| 10 | Yash Tyagi | 25000.0 | Data Entry | Edit | Delete |
| 11 | John William | 75000.0 | Manager | Edit | Delete |

[Add New Employee](#)

Now, click Edit to make some changes in the provided data.



Edit Employee

Name :

Salary :

Designation :

Now, click Edit Save to add the entry with changes into the database.



http://localhost:8082/SpringMVCCRUDSimple/viewemp

Employees List

| Id | Name | Salary | Designation | Edit | Delete |
|----|----------------|---------|---------------------|----------------------|------------------------|
| 8 | Sonoo Jaiswal | 85000.0 | Team Leader | Edit | Delete |
| 9 | Ashutosh Kumar | 50000.0 | Enginner | Edit | Delete |
| 10 | Yash Tyagi | 35000.0 | Data Entry Operator | Edit | Delete |
| 11 | John William | 75000.0 | Manager | Edit | Delete |

[Add New Employee](#)

Now, click Delete to delete the entry from the database.

http://localhost:8082/SpringMVCCRUDSimple/viewemp

Employees List

| Id | Name | Salary | Designation | Edit | Delete |
|----|----------------|---------|-------------|----------------------|------------------------|
| 8 | Sonoo Jaiswal | 85000.0 | Team Leader | Edit | Delete |
| 9 | Ashutosh Kumar | 50000.0 | Enginner | Edit | Delete |
| 11 | John William | 75000.0 | Manager | Edit | Delete |

[Add New Employee](#)

Spring Boot Essentials

What Is an Embedded Server?

- An embedded server is embedded as part of the deployable application.
- If we talk about Java applications, that would be a JAR.
- The advantage with this is you don't need the server pre-installed in the deployment environment.
- With SpringBoot, the default embedded server is Tomcat. Other options available are Jetty and Undertow.
- A lot of developers used to working with WAR and EAR files tend to assume that using an embedded server in a JAR is not stable.

All rights reserved.

- Embedded servers are quite scalable, and can host applications that support millions of users. These are no less scalable than the conventional fat servers.

Think about what you would need to be able to deploy your application (typically) on a virtual machine.

Step 1: Install Java

Step 2: Install the Web/Application Server (Tomcat/Websphere/Weblogic etc)

Step 3: Deploy the application war

What if we want to simplify this?

How about making the server a part of the application?

You would just need a virtual machine with Java installed and you would be able to directly deploy the application on the virtual machine.

This idea is the genesis for Embedded Servers. When we create an application deployable, we would embed the server (for example, tomcat) inside the deployable.

For example, for a Spring Boot Application, you can generate an application jar which contains Embedded Tomcat. You can run a web application as a normal Java application!

Embedded server implies that our deployable unit contains the binaries for the server (example, tomcat.jar).

Default Embedded Server with Spring Boot - Tomcat

We have included Spring Boot Starter Web in our dependencies when creating the spring boot project.

Let's take a quick look at the dependencies for spring-boot-starter-web

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-tomcat</artifactId>
  <version>2.3.1.RELEASE</version>
  <scope>compile</scope>
</dependency>
```

You can see that by default Starter Web includes a dependency on starter tomcat.

Starter Tomcat has the following dependencies.



```
<dependency>
  <groupId>org.apache.tomcat.embed</groupId>
  <artifactId>tomcat-embed-core</artifactId>
  <scope>compile</scope>
</dependency>
<dependency>
  <groupId>org.apache.tomcat.embed</groupId>
  <artifactId>tomcat-embed-el</artifactId>
  <scope>compile</scope>
</dependency>
<dependency>
  <groupId>org.apache.tomcat.embed</groupId>
  <artifactId>tomcat-embed-websocket</artifactId>
  <scope>compile</scope>
</dependency>
```

Starter Tomcat brings in all the dependencies need to run Tomcat as an embedded server.

Run the web application using an Embedded Server

When you run `SpringBootTutorialBasicsApplication.java` as a Java Application, you would see that the server would start up and start serving requests.

An extract from the log

```
o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat initialized with port(s): 8080 (http)
```

You can see that tomcat has started by default on port 8080.

You can customize the port in `application.properties`

```
server.port=9080
```

When you do a 'mvn clean install' on the project, a jar artifact named `spring-boot-tutorial-basics-0.0.1-SNAPSHOT.jar` is generated.

Embedded Server Spring Boot Configuration

Spring Boot provides a number of options to configure the embedded server through `application.properties`



```
# EMBEDDED SERVER CONFIGURATION (ServerProperties)
server.compression.enabled=false # If response compression is enabled.
server.context-path= # Context path of the application.
server.display-name=application # Display name of the application.
server.error.include-stacktrace=never # When to include a "stacktrace" attribute.
server.error.path=/error # Path of the error controller.
server.error.whitelabel.enabled=true # Enable the default error page displayed in browsers in case of a
server error.
server.port=8080 # Server HTTP port.
server.server-header= # Value to use for the Server response header (no header is sent if empty)
server.servlet-path=/ # Path of the main dispatcher servlet.
```

Transaction Management

A database transaction is a sequence of actions that are treated as a single unit of work. These actions should either complete entirely or take no effect at all. Transaction management is an important part of RDBMS-oriented enterprise application to ensure data integrity and consistency. The concept of transactions can be described with the following four key properties described as ACID –

- **Atomicity** – A transaction should be treated as a single unit of operation, which means either the entire sequence of operations is successful or unsuccessful.
- **Consistency** – This represents the consistency of the referential integrity of the database, unique primary keys in tables, etc.
- **Isolation** – There may be many transactions processing with the same data set at the same time. Each transaction should be isolated from others to prevent data corruption.
- **Durability** – Once a transaction has completed, the results of this transaction have to be made permanent and cannot be erased from the database due to system failure.

A real RDBMS database system will guarantee all four properties for each transaction. The simplistic view of a transaction issued to the database using SQL is as follows –

- Begin the transaction using begin transaction command.
- Perform various deleted, update or insert operations using SQL queries.
- If all the operation are successful then perform commit otherwise rollback all the operations.

Spring framework provides an abstract layer on top of different underlying transaction management APIs.

Spring's transaction support aims to provide an alternative to EJB transactions by adding transaction capabilities



to POJOs. Spring supports both programmatic and declarative transaction management. EJBs require an application server, but Spring transaction management can be implemented without the need of an application server.

Local vs. Global Transactions

- Local transactions are specific to a single transactional resource like a JDBC connection, whereas global transactions can span multiple transactional resources like transaction in a distributed system.
- Local transaction management can be useful in a centralized computing environment where application components and resources are located at a single site, and transaction management only involves a local data manager running on a single machine. Local transactions are easier to be implemented.
- Global transaction management is required in a distributed computing environment where all the resources are distributed across multiple systems. In such a case, transaction management needs to be done both at local and global levels. A distributed or a global transaction is executed across multiple systems, and its execution requires coordination between the global transaction management system and all the local data managers of all the involved systems.

Programmatic vs. Declarative

Spring supports two types of transaction management –

- Programmatic transaction management – This means that you have to manage the transaction with the help of programming. That gives you extreme flexibility, but it is difficult to maintain.
- Declarative transaction management – This means you separate transaction management from the business code. You only use annotations or XML-based configuration to manage the transactions.

Declarative transaction management is preferable over programmatic transaction management though it is less flexible than programmatic transaction management, which allows you to control transactions through your code. But as a kind of crosscutting concern, declarative transaction management can be modularized with the AOP approach. Spring supports declarative transaction management through the Spring AOP framework.

Spring Transaction Abstractions

The key to the Spring transaction abstraction is defined by the `org.springframework.transaction.PlatformTransactionManager` interface, which is as follows:



```
public interface PlatformTransactionManager {
    TransactionStatus getTransaction(TransactionDefinition definition);
    throws TransactionException;

    void commit(TransactionStatus status) throws TransactionException;
    void rollback(TransactionStatus status) throws TransactionException;
}
```

| Sr No | Method & Description |
|-------|--|
| 1 | TransactionStatus getTransaction(TransactionDefinition definition) This method returns a currently active transaction or creates a new one, according to the specified propagation behavior. |
| 2 | void commit(TransactionStatus status) This method commits the given transaction, with regard to its status. |
| 3 | void rollback(TransactionStatus status) This method performs a rollback of the given transaction. |

The TransactionDefinition is the core interface of the transaction support in Spring and it is defined as follows –

```
public interface TransactionDefinition {
    int getPropagationBehavior();
    int getIsolationLevel();
    String getName();
    int getTimeout();
    boolean isReadOnly();
}
```




| Sr.No | Method & Description |
|-------|--|
| 1 | int getPropagationBehavior() This method returns the propagation behavior. Spring offers all of the transaction propagation options familiar from EJB CMT. |
| 2 | int getIsolationLevel() This method returns the degree to which this transaction is isolated from the work of other transactions. |
| 3 | String getName() This method returns the name of this transaction. |
| 4 | int getTimeout() This method returns the time in seconds in which the transaction must complete. |
| 5 | boolean isReadOnly() This method returns whether the transaction is read-only. |

The TransactionStatus interface provides a simple way for transactional code to control transaction execution and query transaction status.

```
public interface TransactionStatus extends SavepointManager {
    boolean isNewTransaction();
    boolean hasSavepoint();
    void setRollbackOnly();
    boolean isRollbackOnly();
    boolean isCompleted();
}
```

| Sr.No. | Method & Description |
|--------|--|
| 1 | boolean hasSavepoint() This method returns whether this transaction internally carries a savepoint, i.e., has been created as nested transaction based on a savepoint. |
| 2 | boolean isCompleted() This method returns whether this transaction is completed, i.e., whether it has already been committed or rolled back. |



| | |
|---|--|
| 3 | boolean isNewTransaction() This method returns true in case the present transaction is new. |
| 4 | boolean isRollbackOnly() This method returns whether the transaction has been marked as rollback-only. |
| 5 | void setRollbackOnly() This method sets the transaction as rollback-only. |

Spring Boot Property Categories

There are sixteen categories of Spring Boot Property are as follows:

1. Core Properties
2. Cache Properties
3. Mail Properties
4. JSON Properties
5. Data Properties
6. Transaction Properties
7. Data Migration Properties
8. Integration Properties
9. Web Properties
10. Templating Properties
11. Server Properties
12. Security Properties
13. RSocket Properties
14. Actuator Properties
15. DevTools Properties
16. Testing Properties

Application Properties Table

The following tables provide a list of common Spring Boot properties:



| Property | Default Values | Description |
|----------------------------------|----------------------|--|
| Debug | false | It enables debug logs. |
| spring.application.name | | It is used to set the application name. |
| spring.application.admin.enabled | false | It is used to enable admin features of the application. |
| spring.config.name | application | It is used to set config file name. |
| spring.config.location | | It is used to config the file name. |
| server.port | 8080 | Configures the HTTP server port |
| server.servlet.context-path | | It configures the context path of the application. |
| logging.file.path | | It configures the location of the log file. |
| spring.banner.charset | UTF-8 | Banner file encoding. |
| spring.banner.location | classpath:banner.txt | It is used to set banner file location. |
| logging.file | | It is used to set log file name. For example, data.log. |
| spring.application.index | | It is used to set application index. |
| spring.application.name | | It is used to set the application name. |
| spring.application.admin.enabled | false | It is used to enable admin features for the application. |
| spring.config.location | | It is used to config the file locations. |
| spring.config.name | application | It is used to set config the file name. |
| spring.mail.default-encoding | UTF-8 | It is used to set default MimeMessage encoding. |



| | | |
|--|-------|--|
| spring.mail.host | | It is used to set SMTP server host. For example, smtp.example.com. |
| spring.mail.password | | It is used to set login password of the SMTP server. |
| spring.mail.port | | It is used to set SMTP server port. |
| spring.mail.test-connection | false | It is used to test that the mail server is available on startup. |
| spring.mail.username | | It is used to set login user of the SMTP server. |
| spring.main.sources | | It is used to set sources for the application. |
| server.address | | It is used to set network address to which the server should bind to. |
| server.connection-timeout | | It is used to set time in milliseconds that connectors will wait for another HTTP request before closing the connection. |
| server.context-path | | It is used to set context path of the application. |
| server.port | 8080 | It is used to set HTTP port. |
| server.server-header | | It is used for the Server response header (no header is sent if empty) |
| server.servlet-path | / | It is used to set path of the main dispatcher servlet |
| server.ssl.enabled | | It is used to enable SSL support. |
| spring.http.multipart.enabled | True | It is used to enable support of multi-part uploads. |
| spring.servlet.multipart.max-file-size | 1MB | It is used to set max file size. |



| | | |
|--|-----------------------|---|
| spring.mvc.async.request-timeout | | It is used to set time in milliseconds. |
| spring.mvc.date-format | | It is used to set date format. For example, dd/MM/yyyy. |
| spring.mvc.locale | | It is used to set locale for the application. |
| spring.social.facebook.app-id | | It is used to set application's Facebook App ID. |
| spring.social.linkedin.app-id | | It is used to set application's LinkedIn App ID. |
| spring.social.twitter.app-id | | It is used to set application's Twitter App ID. |
| security.basic.authorize-mode | role | It is used to set security authorize mode to apply. |
| security.basic.enabled | true | It is used to enable basic authentication. |
| Spring.test.database.replace | any | Type of existing DataSource to replace. |
| Spring.test.mockmvc.print | default | MVC Print option |
| spring.freemarker.content-type | text/html | Content Type value |
| server.server-header | | Value to use for the server response header. |
| spring.security.filter.dispatcher-type | async, error, request | Security filter chain dispatcher types. |
| spring.security.filter.order | -100 | Security filter chain order. |
| spring.security.oauth2.client.registration.* | | OAuth client registrations. |
| spring.security.oauth2.client.provider.* | | OAuth provider details. |

Auto-configuration

Spring Boot auto-configuration attempts to automatically configure your Spring application based on the jardependencies that you have added. For example, if HSQLDB is on your classpath, and you have not



manually configured any database connection beans, then Spring Boot auto-configures an in-memory database.

You need to opt-in to auto-configuration by adding the `@EnableAutoConfiguration` or `@SpringBootApplication` annotations to one of your `@Configuration` classes.

Gradually Replacing Auto-configuration

Auto-configuration is non-invasive. At any point, you can start to define your own configuration to replace specific parts of the auto-configuration. For example, if you add your own `DataSource` bean, the default embedded database support backs away.

If you need to find out what auto-configuration is currently being applied, and why, start your application with the `--debug` switch. Doing so enables debug logs for a selection of core loggers and logs conditions report to the console.

Disabling Specific Auto-configuration Classes

If you find that specific auto-configuration classes that you do not want are being applied, you can use the `exclude` attribute of `@SpringBootApplication` to disable them, as shown in the following example:

```
@SpringBootApplication(exclude = { DataSourceAutoConfiguration.class })
public class MyApplication {

}
```

If the class is not on the classpath, you can use the `excludeName` attribute of the annotation and specify the fully qualified name instead. If you prefer to use `@EnableAutoConfiguration` rather than `@SpringBootApplication`, `exclude` and `excludeName` are also available. Finally, you can also control the list of auto-configuration classes to exclude by using the `spring.autoconfigure.exclude` property.

Note:

Even though auto-configuration classes are public, the only aspect of the class that is considered public API is the name of the class which can be used for disabling the auto-configuration. The actual contents of those classes, such as nested configuration classes or bean methods are for internal use only and we do not recommend using those directly.



Creating a Spring Boot Project

Following are the steps to create a simple Spring Boot Project.

Step 1: Open the Spring initializr <https://start.spring.io>

Step 2: Provide the Group and Artifact name. We have provided Group name com.javatpoint and Artifact spring-boot-example.

Step 3: Now click on the Generate button.

The screenshot shows the Spring Initializr web form with the following configuration:

- Project:** Maven Project (selected)
- Language:** Java (selected)
- Spring Boot:** 2.2.2 (SNAPSHOT) (selected)
- Project Metadata:**
 - Group:** com.javatpoint
 - Artifact:** spring-boot-example
 - Options:** (expanded)
- Dependencies:**
 - Search dependencies to add:** Web, Security, JPA, Actuator, Devtools...
 - Selected dependencies:** No dependency selected

At the bottom, there are three buttons: **Generate - Ctrl + G** (highlighted with a red box), **Explore - Ctrl + Space**, and **Share...**.

When we click on the Generate button, it starts packing the project in a .rar file and downloads the project.

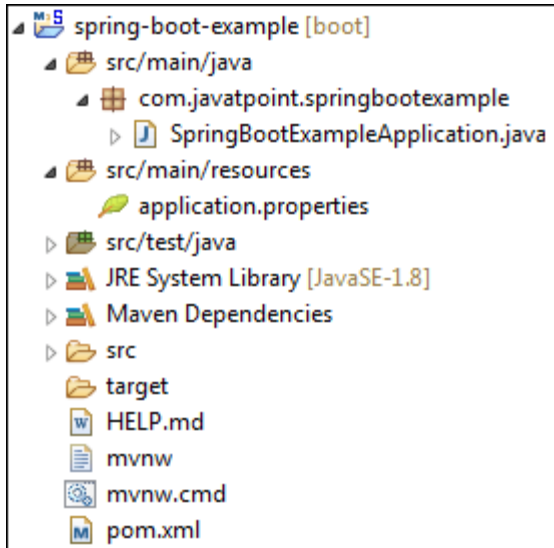
Step 4: Extract the RAR file.



Step 5: Import the folder.

File -> Import -> Existing Maven Project -> Next -> Browse -> Select the project -> Finish

It takes some time to import the project. When the project imports successfully, we can see the project directory in the Package Explorer. The following image shows the project directory:



SpringBootExampleApplication.java

```
package com.javatpoint.springbootexample;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
@SpringBootApplication
public class SpringBootExampleApplication
{
    public static void main(String[] args)
    {
        SpringApplication.run(SpringBootExampleApplication.class, args);
    }
}
```

Pom.xml



```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance" xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-
4.0.0.xsd">
<modelVersion>4.0.0</modelVersion>
<parent>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-parent</artifactId>
<version>2.2.2.BUILD-SNAPSHOT</version>
<relativePath/> <!-- lookup parent from repository -->
</parent>
<groupId>com.javatpoint</groupId>
<artifactId>spring-boot-example</artifactId>
<version>0.0.1-SNAPSHOT</version>
<name>spring-boot-example</name>
<description>Demo project for Spring Boot</description>
<properties>
<java.version>1.8</java.version>
</properties>
<dependencies>
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter</artifactId>
</dependency>
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-test</artifactId>
<scope>test</scope>
<exclusions>
<exclusion>
<groupId>org.junit.vintage</groupId>
<artifactId>junit-vintage-engine</artifactId>
</exclusion>
</exclusions>
</dependency>
</dependencies>
```

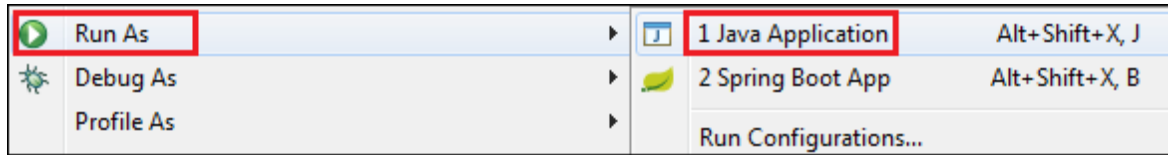


```
<build>
<plugins>
<plugin>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-maven-plugin</artifactId>
</plugin>
</plugins>
</build>
<repositories>
<repository>
<id>spring-milestones</id>
<name>Spring Milestones</name>
<url>https://repo.spring.io/milestone</url>
</repository>
<repository>
<id>spring-snapshots</id>
<name>Spring Snapshots</name>
<url>https://repo.spring.io/snapshot</url>
<snapshots>
<enabled>true</enabled>
</snapshots>
</repository>
</repositories>
<pluginRepositories>
<pluginRepository>
<id>spring-milestones</id>
<name>Spring Milestones</name>
<url>https://repo.spring.io/milestone</url>
</pluginRepository>
<pluginRepository>
<id>spring-snapshots</id>
<name>Spring Snapshots</name>
<url>https://repo.spring.io/snapshot</url>
<snapshots>
<enabled>true</enabled>
</snapshots>
</pluginRepository>
</pluginRepositories>
</project>
```



Step 6: Run the SpringBootExampleApplication.java file.

Right-click on the file -> Run As -> Java Applications



The following image shows the application runs successfully.

```
: Starting SpringBootExampleApplication on Anubhav-PC with PID 5096 (C:\Users\Anubhav
: No active profile set, falling back to default profiles: default
: Started SpringBootExampleApplication in 41.147 seconds (JVM running for 1856.017)
```

EduBridge





Spring Data JPA

Spring Data JPA

- Spring Data JPA API provides JpaTemplate class to integrate spring application with JPA.
- Spring Data JPA, part of the larger Spring Data family, makes it easy to easily implement JPA based repositories. It makes it easier to build Spring-powered applications that use data access technologies.
- Implementing a data access layer of an application has been cumbersome for quite a while. Too much boilerplate code has to be written to execute simple queries as well as perform pagination, and auditing. Spring Data JPA aims to significantly improve the implementation of data access layers by reducing the effort to the amount that's actually needed. As a developer you write your repository interfaces, including custom finder methods, and Spring will provide the implementation automatically.
- JPA (Java Persistent API) is the sun specification for persisting objects in the enterprise application. It is currently used as the replacement for complex entity beans.

The implementation of JPA specification is provided by many vendors such as:

- Hibernate
- Toplink
- iBatis
- OpenJPA etc.

Advantages

- You don't need to write the before and after code for persisting, updating, deleting or searching object such as creating Persistence instance, creating EntityManagerFactory instance, creating EntityTransaction instance, creating EntityManager instance, committing EntityTransaction instance and closing EntityManager.
- Create and support repositories created with Spring and JPA
- Support QueryDSL and JPA queries
- Audit of domain classes
- Support for batch loading, sorting, dynamical queries
- Supports XML mapping for entities

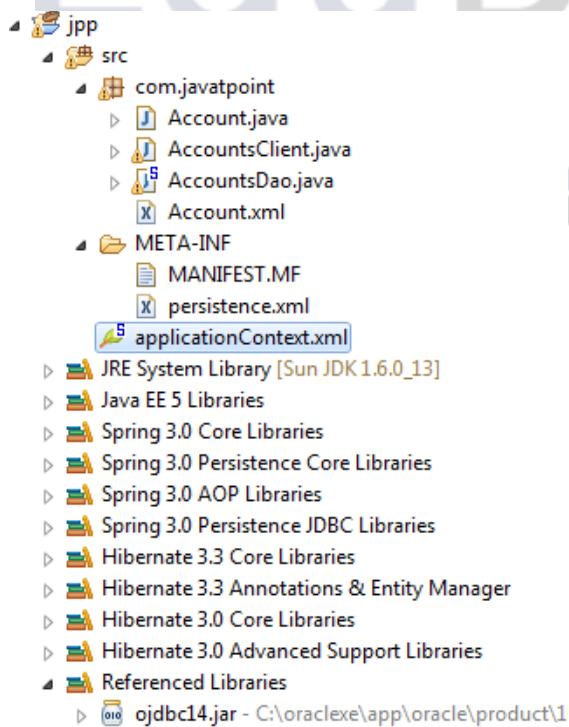
- Reduce code size for generic CRUD operations by using CrudRepository

Example of Spring and JPA Integration

Let's see the simple steps to integration spring application with JPA:

- create Account.java file
- create Account.xml file
- create AccountDao.java file
- create persistence.xml file
- create applicationContext.xml file
- create AccountsClient.java file

In this example, we are going to integrate the hibernate application with spring. Let's see the **directory structure** of jpa example with spring.



1) Account.java

It is a simple POJO class.

```
package com.javatpoint;

public class Account {
    private int accountNumber;
    private String owner;
    private double balance;
    //no-arg and parameterized constructor
    //getters and setters
}
```

2) Account.xml

This mapping file contains all the information of the persistent class.

```
<entity-mappings version="1.0"
    xmlns="http://java.sun.com/xml/ns/persistence/orm"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/persistence/orm
    http://java.sun.com/xml/ns/persistence/orm_1_0.xsd">

    <entity class="com.javatpoint.Account">
        <table name="account100"></table>
        <attributes>
            <id name="accountNumber">
                <column name="accountnumber"/>
            </id>
            <basic name="owner">
                <column name="owner"/>
            </basic>
            <basic name="balance">
                <column name="balance"/>
            </basic>
        </attributes>
    </entity>
</entity-mappings>
```



3) AccountDao.java

```
package com.javatpoint;
import java.util.List;
import org.springframework.orm.jpa.JpaTemplate;
import org.springframework.transaction.annotation.Transactional;
@Transactional
public class AccountsDao{
    JpaTemplate template;

    public void setTemplate(JpaTemplate template) {
        this.template = template;
    }
    public void createAccount(int accountNumber,String owner,double balance){
        Account account = new Account(accountNumber,owner,balance);
        template.persist(account);
    }
    public void updateBalance(int accountNumber,double newBalance){
        Account account = template.find(Account.class, accountNumber);
        if(account != null){
            account.setBalance(newBalance);
        }
        template.merge(account);
    }
    public void deleteAccount(int accountNumber){
        Account account = template.find(Account.class, accountNumber);
        if(account != null)
            template.remove(account);
    }
    public List<Account> getAllAccounts(){
        List<Account> accounts =template.find("select acc from Account acc");
        return accounts;
    }
}
```

4) Persistence.xml



```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
    http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd" version="1.0">

  <persistence-unit name="ForAccountsDB">
    <mapping-file>com/javatpoint/Account.xml</mapping-file>
    <class>com.javatpoint.Account</class>
  </persistence-unit>
</persistence>
```

5) applicationContext.xml

EduBridge





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

  <tx:annotation-driven transaction-manager="jpaTxnManagerBean" proxy-target-class="true"/>

  <bean id="dataSourceBean" class="org.springframework.jdbc.datasource.DriverManagerDataSource">
    <property name="driverClassName" value="oracle.jdbc.driver.OracleDriver"></property>
    <property name="url" value="jdbc:oracle:thin:@localhost:1521:xe"></property>
    <property name="username" value="system"></property>
    <property name="password" value="oracle"></property>
  </bean>

  <bean id="hbAdapterBean" class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter">
    <property name="showSql" value="true"></property>
    <property name="generateDdl" value="true"></property>
    <property name="databasePlatform" value="org.hibernate.dialect.OracleDialect"></property>
  </bean>

  <bean id="emfBean" class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
    <property name="dataSource" ref="dataSourceBean"></property>
    <property name="jpaVendorAdapter" ref="hbAdapterBean"></property>
  </bean>

  <bean id="jpaTemplateBean" class="org.springframework.orm.jpa.JpaTemplate">
    <property name="entityManagerFactory" ref="emfBean"></property>
  </bean>

  <bean id="accountsDaoBean" class="com.javatpoint.AccountsDao">
    <property name="template" ref="jpaTemplateBean"></property>
  </bean>

  <bean id="jpaTxnManagerBean" class="org.springframework.orm.jpa.JpaTransactionManager">
    <property name="entityManagerFactory" ref="emfBean"></property>
  </bean>
</beans>
```



The generateDdl property will create the table automatically.

The showSql property will show the sql query on console.

6) Accountscient.java

```
package com.javatpoint;

import java.util.List;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import org.springframework.context.support.FileSystemXmlApplicationContext;

public class AccountsClient{
    public static void main(String[] args){
        ApplicationContext context = new ClassPathXmlApplicationContext("applicationContext.xml");
        AccountsDao accountsDao = context.getBean("accountsDaoBean",AccountsDao.class);

        accountsDao.createAccount(15, "Jai Kumar", 41000);
        accountsDao.createAccount(20, "Rishi ", 35000);
        System.out.println("Accounts created");

        //accountsDao.updateBalance(20, 50000);
        //System.out.println("Account balance updated");

        /*List<Account> accounts = accountsDao.getAllAccounts();
        for (int i = 0; i < accounts.size(); i++) {
            Account acc = accounts.get(i);
            System.out.println(acc.getAccountNumber() + " : " + acc.getOwner() + " (" + acc.getBalance() + ")");
        }*/

        //accountsDao.deleteAccount(111);
        //System.out.println("Account deleted");

    }
}
```

Output:



```
Hibernate: insert into account100 (balance, owner, accountnumber) values (?, ?, ?)
Hibernate: insert into account100 (balance, owner, accountnumber) values (?, ?, ?)
Accounts created
```

Query Methods

Query methods are methods that find information from the database and are declared on the repository interface. For example, if we want to create a database query that finds the Todo object that has a specific id, we can create the query method by adding the `findById()` method to the `TodoRepository` interface. After we have done this, our repository interface looks as follows:

```
import org.springframework.data.repository.Repository;
interface TodoRepository extends Repository<Todo, Long> {

    //This is a query method.
    Todo findById(Long id);
}
```

Returning Values from Query Methods

A query method can return only one result or more than one result. Also, we can create a query method that is invoked asynchronously. This section addresses each of these situations and describes what kind of return values we can use in each situation.

First, if we are writing a query that should return only one result, we can return the following types:

- Basic type. Our query method will return the found basic type or null.
- Entity. Our query method will return an entity object or null.
- Guava / Java 8 `Optional<T>`. Our query method will return an `Optional` that contains the found object or an empty `Optional`.

Here are some examples of query methods that return only one result:



```
import java.util.Optional;
import org.springframework.data.jpa.repository.Query;
import org.springframework.data.repository.Repository;
import org.springframework.data.repository.query.Param;

interface TodoRepository extends Repository<Todo, Long> {

    @Query("SELECT t.title FROM Todo t where t.id = :id")
    String findTitleById(@Param("id") Long id);

    @Query("SELECT t.title FROM Todo t where t.id = :id")
    Optional<String> findTitleById(@Param("id") Long id);

    Todo findById(Long id);

    Optional<Todo> findById(Long id);
}
```

Second, if we are writing a query method that should return more than one result, we can return the following types:

- `List<T>`. Our query method will return a list that contains the query results or an empty list.
- `Stream<T>`. Our query method will return a Stream that can be used to access the query results or an emptyStream.

Here are some examples of query methods that return more than one result:

```
import java.util.stream.Stream;
import org.springframework.data.repository.Repository;

interface TodoRepository extends Repository<Todo, Long> {

    List<Todo> findByTitle(String title);

    Stream<Todo> findByTitle(String title);
}
```

Third, if we want that our query method is executed asynchronously, we have to annotate it with the `@Async` annotation and return a `Future<T>` object. Here are some examples of query methods that are executed asynchronously:



```
import java.util.concurrent.Future;
import java.util.stream.Stream;
import org.springframework.data.jpa.repository.Query;
import org.springframework.data.repository.Repository;
import org.springframework.data.repository.query.Param;
import org.springframework.scheduling.annotation.Async;

interface TodoRepository extends Repository<Todo, Long> {

    @Async
    @Query("SELECT t.title FROM Todo t where t.id = :id")
    Future<String> findTitleById(@Param("id") Long id);

    @Async
    @Query("SELECT t.title FROM Todo t where t.id = :id")
    Future<Optional<String>> findTitleById(@Param("id") Long id);

    @Async
    Future<Todo> findById(Long id);

    @Async
    Future<Optional<Todo>> findById(Long id);

    @Async
    Future<List<Todo>> findByTitle(String title);

    @Async
    Future<Stream<Todo>> findByTitle(String title);
}
```

Passing Method Parameters to Query Methods

We can pass parameters to our database queries by passing method parameters to our query methods. Spring DataJPA supports both position-based parameter binding and named parameters.

The position-based parameter binding means that the order of our method parameters decides which placeholders are replaced with them. In other words, the first placeholder is replaced with the first method parameter, the second placeholder is replaced with the second method parameter, and so on.



Here are some query methods that use the position based parameter binding:

```
import java.util.Optional
import org.springframework.data.jpa.repository.Query;
import org.springframework.data.repository.Repository;

interface TodoRepository extends Repository<Todo, Long> {

    public Optional<Todo> findByTitleAndDescription(String title, String description);

    @Query("SELECT t FROM Todo t where t.title = ?1 AND t.description = ?2")
    public Optional<Todo> findByTitleAndDescription(String title, String description);

    @Query(value = "SELECT * FROM todos t where t.title = ?0 AND t.description = ?1",
        nativeQuery=true
    )
    public Optional<Todo> findByTitleAndDescription(String title, String description);
}
```

Using position-based parameter binding is a bit error prone because we cannot change the order of the method parameters or the order of the placeholders without breaking our database query. We can solve this problem by using named parameters.

We can use named parameters by replacing the numeric placeholders found from our database queries with concrete parameter names, and annotating our method parameters with the `@Param` annotation.

Here are some query methods that use named parameters:



```
import java.util.Optional
import org.springframework.data.jpa.repository.Query;
import org.springframework.data.repository.Repository;
import org.springframework.data.repository.query.Param;

interface TodoRepository extends Repository<Todo, Long> {

    @Query("SELECT t FROM Todo t where t.title = :title AND t.description = :description")
    public Optional<Todo> findByTitleAndDescription(@Param("title") String title,
                                                    @Param("description") String description);

    @Query(
        value = "SELECT * FROM todos t where t.title = :title AND t.description = :description",
        nativeQuery=true
    )
    public Optional<Todo> findByTitleAndDescription(@Param("title") String title,
                                                    @Param("description") String description);
}
```

- Query methods are methods that find information from the database and are declared on the repository interface.
- Spring Data has pretty versatile support for different return values that we can leverage when we are adding query methods to our Spring Data JPA repositories.
- We can pass parameters to our database queries by using either position-based parameter binding or named parameters.

Standard CRUD functionality repositories usually have queries on the underlying datastore. With Spring Data, declaring those queries becomes a four-step process:

1. Declare an interface extending Repository or one of its subinterfaces and type it to the domain class and ID type that it will handle.

```
public interface PersonRepository extends Repository<User, Long> { ... }
```

2. Declare query methods on the interface.



```
List<Person> findByLastname(String lastname);
```

3. Set up Spring to create proxy instances for those interfaces. Either via JavaConfig:

```
import org.springframework.data.jpa.repository.config.EnableJpaRepositories;

@EnableJpaRepositories
class Config {}
```

or via XML configuration:

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

  <jpa:repositories base-package="com.acme.repositories"/>

</beans>
```

The JPA namespace is used in this example. If you are using the repository abstraction for any other store, you need to change this to the appropriate namespace declaration of your store module which should be exchanging jpa in favor of, for example, mongodb. Also, note that the JavaConfig variant doesn't configure a package explicitly as the package of the annotated class is used by default. To customize the package to scan

1. Get the repository instance injected and use it.



```
public class SomeClient {  
  
    @Autowired  
    private PersonRepository repository;  
  
    public void doSomething() {  
        List<Person> persons = repository.findByLastname("Matthews");  
    }  
}
```

Named Query with JPA

Named queries are one of the core concepts in JPA. They enable you to declare a query in your persistence layer and reference it in your business code. That makes it easy to reuse an existing query. It also enables you to separate the definition of your query from your business code.

You can define a named query using a `@NamedQuery` annotation on an entity class or using a `<named-query />` element in your XML mapping.

When you define a named query, you can provide a JPQL query or a native SQL query in very similar ways. Let's take a look at both options.

Entity - Entity.java

Following is the default code of Employee. It represents a Employee table with id, name, age and email columns



```
package com.tutorialspoint.entity;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.NamedQuery;
import javax.persistence.Table;

@Entity
@Table
@NamedQuery(name = "Employee.findByEmail",
query = "select e from Employee e where e.email = ?1")
public class Employee {
    @Id
    @Column
    private int id;
    @Column
    private String name;
    @Column
    private int age;
    @Column
    private String email;

    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 int getAge() {
        return age; }
    public void setAge(int age) {
        this.age = age; }
    public String getEmail() {
        return email; }
    public void setEmail(String email) {
        this.email = email; }
}
```



Repository - EmployeeRepository.java

Add a method to find an employee by its name and age.

```
package com.tutorialspoint.repository;

import org.springframework.data.repository.CrudRepository;
import org.springframework.stereotype.Repository;
import com.tutorialspoint.entity.Employee;

@Repository
public interface EmployeeRepository extends CrudRepository<Employee, Integer> {
    public List<Employee> findByName(String name);
    public List<Employee> findByAge(int age);
    public Employee findByEmail(String email);
}
```

Now Spring JPA will create the implementation of above methods automatically using the query provided in namedquery. Let's test the methods added by adding their test cases in test file. Last two methods of below file tests the named query method added.

Following is the complete code of **EmployeeRepositoryTest**.



```
package com.tutorialspoint.repository;

import static org.junit.jupiter.api.Assertions.assertEquals;
import java.util.ArrayList;
import java.util.List;
import javax.transaction.Transactional;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.context.junit.jupiter.SpringExtension;
import com.tutorialspoint.entity.Employee;
import com.tutorialspoint.springboot2.SprintBootH2Application;

@ExtendWith(SpringExtension.class)
@Transactional
@SpringBootTest(classes = SprintBootH2Application.class)
public class EmployeeRepositoryTest {
    @Autowired
    private EmployeeRepository employeeRepository;

    @Test
    public void testFindById() {
        Employee employee = getEmployee();
        employeeRepository.save(employee);
        Employee result = employeeRepository.findById(employee.getId()).get();
        assertEquals(employee.getId(), result.getId());
    }
    @Test
    public void testFindAll() {
        Employee employee = getEmployee();
        employeeRepository.save(employee);
        List<Employee> result = new ArrayList<>();
        employeeRepository.findAll().forEach(e -> result.add(e));
        assertEquals(result.size(), 1);
    }
}
```



```
@Test
public void testSave() {
    Employee employee = getEmployee();
    employeeRepository.save(employee);
    Employee found = employeeRepository.findById(employee.getId()).get();
    assertEquals(employee.getId(), found.getId());
}

@Test
public void testDeleteById() {
    Employee employee = getEmployee();
    employeeRepository.save(employee);
    employeeRepository.deleteById(employee.getId());
    List<Employee> result = new ArrayList<>();
    employeeRepository.findAll().forEach(e -> result.add(e));
    assertEquals(result.size(), 0);
}

private Employee getEmployee() {
    Employee employee = new Employee();
    employee.setId(1);
    employee.setName("Mahesh");
    employee.setAge(30);
    employee.setEmail("mahesh@test.com");
    return employee;
}

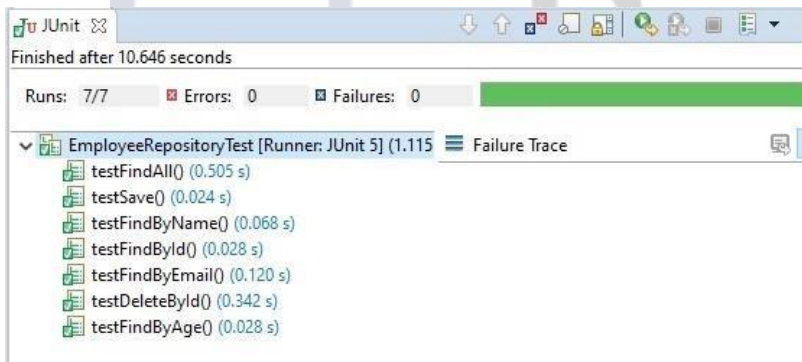
@Test
public void testFindByName() {
    Employee employee = getEmployee();
    employeeRepository.save(employee);
    List<Employee> result = new ArrayList<>();
    employeeRepository.findByName(employee.getName()).forEach(e -> result.add(e));
    assertEquals(result.size(), 1);
}

@Test
public void testFindByAge() {
    Employee employee = getEmployee();
    employeeRepository.save(employee);
    List<Employee> result = new ArrayList<>();
    employeeRepository.findByAge(employee.getAge()).forEach(e -> result.add(e));
    assertEquals(result.size(), 1);
}
```

```
@Test
public void testFindByEmail() {
    Employee employee = getEmployee();
    employeeRepository.save(employee);
    Employee result = employeeRepository.findByEmail(employee.getEmail());
    assertNotNull(result);
}
}
```

Run the test cases

Right Click on the file in eclipse and select Run a JUnit Test and verify the result.



Defining repository interfaces

The repository proxy has two ways to derive a store-specific query from the method name. It can derive the query from the method name directly, or by using an manually defined query. Available options depend on the actual store. However, there's got to be an strategy that decides what actual query is created. Let's have a look at the available options.

Query lookup strategies

The following strategies are available for the repository infrastructure to resolve the query. You can configure the strategy at the namespace through the query-lookup-strategy attribute in case of XML configuration or via the queryLookupStrategy attribute of the Enable\${store}Repositories annotation in case of Java config. Some strategies may not be supported for particular datastores.



CREATE

CREATE attempts to construct a store-specific query from the query method name. The general approach is to remove a given set of well-known prefixes from the method name and parse the rest of the method. Read more about query construction in the section called “Query creation”.

USE_DECLARED_QUERY

USE_DECLARED_QUERY tries to find a declared query and will throw an exception in case it can't find one. The query can be defined by an annotation somewhere or declared by other means. Consult the documentation of the specific store to find available options for that store. If the repository infrastructure does not find a declared query for the method at bootstrap time, it fails.

CREATE_IF_NOT_FOUND (default)

CREATE_IF_NOT_FOUND combines CREATE and USE_DECLARED_QUERY. It looks up a declared query first, and if no declared query is found, it creates a custom method name-based query. This is the default lookup strategy and thus will be used if you do not configure anything explicitly. It allows quick query definition by method names but also custom-tuning of these queries by introducing declared queries as needed.

Query creation

The query builder mechanism built into Spring Data repository infrastructure is useful for building constraining queries over entities of the repository. The mechanism strips the prefixes find...By, read...By, query...By, count...By, and get...By from the method and starts parsing the rest of it. The introducing clause can contain further expressions such as a Distinct to set a distinct flag on the query to be created. However, the first By acts as delimiter to indicate the start of the actual criteria. At a very basic level you can define conditions on entity properties and concatenate them with And and Or.

Example: Query creation from method names



```
public interface PersonRepository extends Repository<User, Long> {

    List<Person> findByEmailAddressAndLastname(EmailAddress emailAddress, String lastname);

    // Enables the distinct flag for the query
    List<Person> findDistinctPeopleByLastnameOrFirstname(String lastname, String firstname);
    List<Person> findPeopleDistinctByLastnameOrFirstname(String lastname, String firstname);

    // Enabling ignoring case for an individual property
    List<Person> findByLastnameIgnoreCase(String lastname);
    // Enabling ignoring case for all suitable properties
    List<Person> findByLastnameAndFirstnameAllIgnoreCase(String lastname, String firstname);

    // Enabling static ORDER BY for a query
    List<Person> findByLastnameOrderByFirstnameAsc(String lastname);
    List<Person> findByLastnameOrderByFirstnameDesc(String lastname);
}
```

The actual result of parsing the method depends on the persistence store for which you create the query. However, there are some general things to notice.

- The expressions are usually property traversals combined with operators that can be concatenated. You can combine property expressions with AND and OR. You also get support for operators such as Between, LessThan, GreaterThan, Like for the property expressions. The supported operators can vary by datastore, so consult the appropriate part of your reference documentation.
- The method parser supports setting an IgnoreCase flag for individual properties, for example, `findByLastnameIgnoreCase(...)` or for all properties of a type that support ignoring case (usually Strings, for example, `findByLastnameAndFirstnameAllIgnoreCase(...)`). Whether ignoring cases is supported may vary by store, so consult the relevant sections in the reference documentation for the store-specific query method.
- You can apply static ordering by appending an OrderBy clause to the query method that references a property and by providing a sorting direction (Asc or Desc).

Creating repository instances

One way to create instances and bean definitions for the repository interfaces is using the Spring namespace that is shipped with each Spring Data module that supports the repository mechanism although we generally recommend to use the Java-Config style configuration.

XML configuration

Each Spring Data module includes a `repositories` element that allows you to simply define a base package that Spring scans for you.

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

  <repositories base-package="com.acme.repositories" />

</beans:beans>
```

In the preceding example, Spring is instructed to scan `com.acme.repositories` and all its subpackages for interfaces extending `Repository` or one of its subinterfaces. For each interface found, the infrastructure registers the persistence technology-specific `FactoryBean` to create the appropriate proxies that handle invocations of the query methods. Each bean is registered under a bean name that is derived from the interface name, so an interface of `UserRepository` would be registered under `userRepository`. The `base-package` attribute allows wildcards, so that you can define a pattern of scanned packages.

Using filters

By default, the infrastructure picks up every interface extending the persistence technology-specific `Repository` subinterface located under the configured base package and creates a bean instance for it. However, you might want more fine-grained control over which interfaces bean instances get created for. To do this you use `<include-filter />` and `<exclude-filter />` elements inside `<repositories />`. The semantics are exactly equivalent to the elements in Spring's context namespace. For details, see Spring reference documentation on these elements.

For example, to exclude certain interfaces from instantiation as repository, you could use the following configuration:

Example: single exclude-filter element



```
<repositories base-package="com.acme.repositories">
  <context:exclude-filter type="regex" expression=".*SomeRepository" />
</repositories>
```

This example excludes all interfaces ending in `SomeRepository` from being instantiated.

JavaConfig

The repository infrastructure can also be triggered using a store-specific `@Enable${store}Repositories` annotation on a `JavaConfig` class.

A sample configuration to enable Spring Data repositories looks something like this.

Example: Sample annotation-based repository configuration

```
@Configuration
@EnableJpaRepositories("com.acme.repositories")
class ApplicationConfiguration {

    @Bean
    public EntityManagerFactory entityManagerFactory() {
        // ...
    }
}
```

Standalone usage

You can also use the repository infrastructure outside of a Spring container, e.g. in CDI environments. You still need some Spring libraries in your classpath, but generally you can set up repositories programmatically as well. The SpringData modules that provide repository support ship a persistence technology-specific `RepositoryFactory` that you can use as follows.

Example: Standalone usage of repository factory

```
RepositoryFactorySupport factory = ... // Instantiate factory here
UserRepository repository = factory.getRepository(UserRepository.class);
```



Spring Data REST

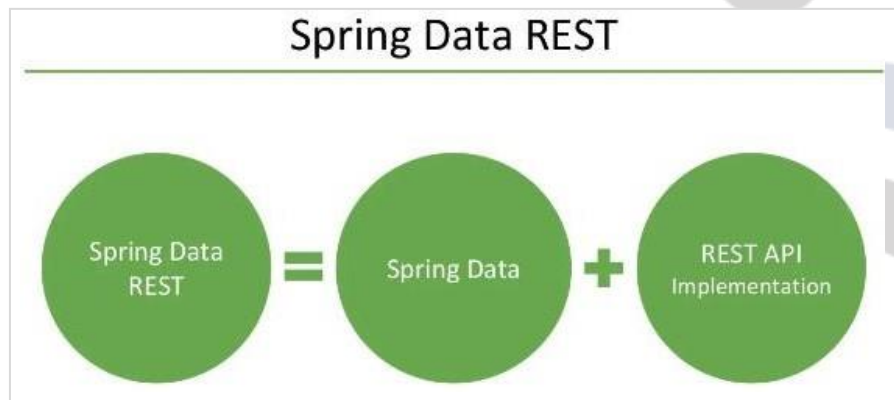
Introduction & Overview

REST web services have become the number one means for application integration on the web. In its core, REST defines that a system that consists of resources with which clients interact. These resources are implemented in a hypermedia-driven way.

Spring Data REST builds on top of the Spring Data repositories and automatically exports those as REST resources. It leverages hypermedia to let clients automatically find functionality exposed by the repositories and integrate these resources into related hypermedia-based functionality.

Spring Data REST is part of the umbrella Spring Data project and makes it easy to build hypermedia-driven REST webservices on top of Spring Data repositories.

Spring Data REST builds on top of Spring Data repositories, analyzes your application's domain model and exposes hypermedia-driven HTTP resources for aggregates contained in the model.



Why Spring Data REST?

RESTful Web Services are very famous for bringing simplicity in application integrations. REST defines few architectural constraints for a client and server interaction. Within the bounds of these constraints, an API exposes resources that a client can interact with.

The server often stores the resources in a persistence layer and exposes them by creating layered components. On the high level, there are two main layers that are present in every web service – data layer and web layer.



Out of which, the data layer is responsible for interacting with the persistence layer and transforming the resources to and from application's domain objects. The web layer, on the other hand, exposes the RESTful api and exchanges the domain objects with the data layer.

While adhering to the REST constraints a web layer is mainly responsible for

- Provide HTTP driven interaction points that clients can access. For example, GET, POST, PUT, OPTIONS, etc. endpoints.
- Serialize and deserialize the JSON payloads into the domain objects.
- Exchange the domain objects with the data persistence layer.
- Handle application-level exceptions and errors and issue appropriate HTTP Status Codes.

Upon paying a little attention to these tasks, we can figure out that most of the tasks remain the same across a different RESTful services. That means, although every RESTful service defines its own domain model, the web part follows the same templated flow.

This is why, Spring introduced Spring Data REST framework that help us avoid this repetitive boilerplate pattern in our web services. To do so, Spring Data REST detects the domain model and automatically exposes RESTful endpoints with a minimal configuration.

Spring Data REST Benefits

Spring Data REST is a framework that builds itself on top of the applications data repositories and expose those repositories in the form of REST endpoints. In order to make it easier for the clients to discover the HTTP access points exposed by the repositories, Spring Data REST uses hypermedia driven endpoints.

Spring Data REST is a web application that can be added using its dependency. Once added and configured (note: Spring Data REST in Spring Boot doesn't require any configurations) it detects any repositories having `@RestResource` or `@RepositoryRestResource` annotations. Based on the entity associated with the repository the Spring Data REST exposes the entity by providing single item resource and collection resource endpoints. Moreover, having the RESTful HAL APIs, clients can discover the available resources.

Let's list down some of the major advantages of using Spring Data REST.

- Helps reducing or nullifying boilerplate components and code blocks and speeds up the overall application development time.
- Works well within a Spring (non-boot) application with minimal configuration.

- Using it within a Spring Boot application requires zero configuration. That is because, Spring Boot auto-configuration takes care of all the necessary configurations.
- The hypermedia driven endpoints help clients to self discover the available resources as well as the resource profiles.
- Takes care about returning the standard HTTP status codes.
- Supports a variety of persistence providers through the respective Spring Data modules – Spring Data JPA, Spring Data MongoDB, Spring Data Neo4j, Spring Data Cassandra, and Spring Data GemFire.

How does it Works?

When we launch a Spring Data Repository application, it first detects all the repositories that have a `@RepositoryRestResource` annotation. For example, the next repository that serves the Student entity.

```
@RepositoryRestResource
public interface StudentRepository
    extends JpaRepository<Student, Long> {
}
```

For this repository the Spring Data REST will automatically expose next two endpoints.

- Single Item Resource: `/students/{id}`
- Collection Resource: `/students`

Note that, the resource name is same as the entity name in lower case and plural form. However, we can always customize the resource names by using the path attribute of the annotation.

Once the application is started, accessing the root URL for example `http://localhost:8080/` returns the below Response.



```
{
  "_links" : {
    "students" : {
      "href" : "http://localhost:8080/students{?page,size,sort}"
    },
    "profile" : {
      "href" : "http://localhost:8080/profile"
    }
  }
}
```

Spring Data REST produces the response in hal+json format. Adhering to the HAL standards, the response contains ‘_links’ to the available resources on the root. Also note that the students resource supports a standard pagination and sorting queries. This is because our repository (JpaRepository) supports pagination and sorting.

Adding Spring Data REST to a Spring Boot Project

Step 1: Add the Following Dependencies to the Project

```
// Dependency for Spring data jpa
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
// Dependency for Spring data rest
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-rest</artifactId>
</dependency>
```

Step 2: Creating a REST Repository

Next, you need to create one repository using any of the repository providers (JpaRepository, PagingAndSortingRepository, CrudRepository) by extending them.



```
/**
 *
 */
package com.trjoshi.dao.api;

import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.rest.core.annotation.RepositoryRestResource;

import com.trjoshi.bean.SpringDataRestConfig;

/**
 * @author tapasjoshi
 *
 */
@RepositoryRestResource(collectionResourceRel = "springDataRestTestConfigs", path =
"springDataRestTestConfigs")
public interface SpringDataRestTestConfigRestRepository extends JpaRepository<SpringDataRestConfig,
String> {

}
```

In the above code snippet, we have added the following annotation where we can provide the collectionResourceRel and the path to specify the REST service and key of the response.

```
@RepositoryRestResource(collectionResourceRel = "springDataRestTestConfigs", path = "springDataRestTestConfigs")
```

By adding this annotation to the repository, the Spring container will create all of the REST services and CRUD operations for this entity, which we can customize as per our needs.

Step 3: Enable Spring Data REST in Your Spring Boot Application

To enable Spring Data REST in your Spring Boot application, you need to add the following annotation to the mainSpring Boot class.

```
@Import(SpringDataRestConfiguration.class):
```



```
package com.trjoshi.springboot;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.boot.builder.SpringApplicationBuilder;
import org.springframework.boot.web.support.SpringBootServletInitializer;
import org.springframework.context.annotation.Import;

import springfox.documentation.spring.data.rest.configuration.SpringDataRestConfiguration;
import springfox.documentation.swagger2.annotations.EnableSwagger2;

@SpringBootApplication
@EnableSwagger2
@Import(SpringDataRestConfiguration.class)
public class SpringBootDemoApplication extends SpringBootServletInitializer{

    @Override
    protected SpringApplicationBuilder configure(SpringApplicationBuilder builder) {
        return builder.sources(SpringBootDemoApplication.class);
    }

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

Now, we are done with the minimum required steps for implementing Spring Data REST, and once we deploy the application, our REST services will be ready, as below.

```
GET : <Context Path>/springDataRestTestConfigs
POST : <Context Path>/springDataRestTestConfigs
DELETE : <Context Path>/springDataRestTestConfigs/{id}
GET : <Context Path>/springDataRestTestConfigs/{id}
PUT : <Context Path>/springDataRestTestConfigs/{id}
```




Configuring Spring Data REST

To install Spring Data REST alongside your existing Spring MVC application, you need to include the appropriate MVC configuration. Spring Data REST configuration is defined in a class called `RepositoryRestMvcConfiguration` and you can import that class into your application's configuration.

To customize the configuration, register a `RepositoryRestConfigurer` and implement or override the `configure...-methods` relevant to your use case.

Repository resources

Fundamentals

The core functionality of Spring Data REST is to export resources for Spring Data repositories. Thus, the core artifact to look at and potentially customize the way the exporting works is the repository interface. Consider the following repository interface:

```
public interface OrderRepository extends CrudRepository<Order, Long> { }
```

For this repository, Spring Data REST exposes a collection resource at `/orders`. The path is derived from the uncapitalized, pluralized, simple class name of the domain class being managed. It also exposes an item resource for each of the items managed by the repository under the URI template `/orders/{id}`.

By default, the HTTP methods to interact with these resources map to the according methods of `CrudRepository`. Read more on that in the sections on collection resources and item resources.

- **Repository methods exposure**

The HTTP resources exposed for a certain repository is mostly driven by the structure of the repository. In other words, the resource exposure will follow which methods you have exposed on the repository. If you extend `CrudRepository` you usually expose all methods required to expose all HTTP resources we can register by default. Each of the resources listed below will define which of the methods need to be present so that a particular HTTP method can be exposed for each of the resources. That means, that repositories that are not exposing those methods — either by not declaring them at all or explicitly using `@RestResource(exported = false)` — won't expose those HTTP methods on those resources.



For details on how to tweak the default method exposure or dedicated HTTP methods individually see Customizingsupported HTTP methods.

- **Default Status Codes**

For the resources exposed, we use a set of default status codes:

- 200 OK: For plain GET requests.
- 201 Created: For POST and PUT requests that create new resources.
- 204 No Content: For PUT, PATCH, and DELETE requests when the configuration is set to not return response bodies for resource updates (`RepositoryRestConfiguration.setResponseBodyOnUpdate(...)`). If the configuration value is set to include responses for PUT, 200 OK is returned for updates, and 201 Created is returned for resource created through PUT.

If the configuration values (`RepositoryRestConfiguration.returnBodyOnUpdate(...)` and `RepositoryRestConfiguration.returnBodyCreate(...)`) are explicitly set to null — which they are by default —, the presence of the HTTP Accept header is used to determine the response code. Read more on this in the detailed description of collection and item resources.

- **Resource Discoverability**

A core principle of HATEOAS is that resources should be discoverable through the publication of links that point to the available resources. There are a few competing de-facto standards of how to represent links in JSON. By default, Spring Data REST uses HAL to render responses. HAL defines the links to be contained in a property of the returned document.

Resource discovery starts at the top level of the application. By issuing a request to the root URL under which the Spring Data REST application is deployed, the client can extract, from the returned JSON object, a set of links that represent the next level of resources that are available to the client.

For example, to discover what resources are available at the root of the application, issue an HTTP GET to the root URL, as follows:

```
curl -v http://localhost:8080/
```

```
< HTTP/1.1 200 OK
< Content-Type: application/hal+json
```

```
{ "_links" : {
  "orders" : {
```



```
"href" : "http://localhost:8080/orders"
},
"profile" : {
  "href" : "http://localhost:8080/api/alps"
}
}
```

The property of the result document is an object that consists of keys representing the relation type, with nested linkobjects as specified in HAL.

The Collection Resource

Spring Data REST exposes a collection resource named after the uncapitalized, pluralized version of the domain class the exported repository is handling. Both the name of the resource and the path can be customized by using `@RepositoryRestResource` on the repository interface.

A collection resource is the one that returns list of all individual resource items. For example, the `/students` resource in the above example. The response of a collection resource for example (`http://localhost:8080/students`) looks like this.



```
{
  "_embedded" : {
    "students" : [ {
      "firstName" : "Jon",
      "lastName" : "Snow", "year" : 2024,
      "_links" : {
        "self" : {
          "href" : "http://localhost:8080/students/1"
        },
        "student" : {
          "href" : "http://localhost:8080/students/1"
        }
      }
    }, {
      "firstName" : "Alton",
      "lastName" : "Lannister",
      "year" : 2025,
      "_links" : {
        "self" : {
          "href" : "http://localhost:8080/students/2"
        },
        "student" : {
          "href" : "http://localhost:8080/students/2"
        }
      }
    }
  ],
  "_links" : {
    "self" : {
      "href" : "http://localhost:8080/students"
    },
    "profile" : {
      "href" : "http://localhost:8080/profile/students"
    }
  },
  "page" : {
    "size" : 20,
    "totalElements" : 2,
    "totalPages" : 1,
    "number" : 0
  }
}
```



As shown, the response contains a list of all available resources (students) in the form of individual URLs. Additionally, the response contains a link to the students profile and a block of page resource.

Supported HTTP Methods

The collection resource endpoint supports HTTP GET, POST and HEAD methods. Using any other HTTP method results in 405 – Method Not Allowed status.

- HTTP GET – The HTTP GET method on the collection resource endpoint makes use of `findAll(Pageable)`, `findAll(Sort)`, or `findAll()` methods of the repository. If the respective method is not exported then a default status code of 405 is returned.
- HTTP HEAD – HTTP Head is exactly similar to the GET, except that it doesn't return any data.
- HTTP POST – The HTTP POST on the collection resource makes use of `save(..)` method and it creates a new resource every time it is invoked.

The Item Resource

Spring Data REST exposes a resource for individual collection items as sub-resources of the collection resource. A single item resource locates an individual item by its primary key. For example, the `/students/{id}` endpoint. When we execute a GET `http://localhost:8080/students/1`, the student resource having `id = 1` is returned.

```
{
  "firstName": "Jon",
  "lastName": "Snow",
  "year": 2024,
  "_links": {
    "self": {
      "href": "http://localhost:8080/students/1"
    },
    "student": {
      "href": "http://localhost:8080/students/1"
    }
  }
}
```



Supported HTTP Methods

The single item resource supports HTTP GET, PUT, PATCH, DELETE and HEAD endpoints. These HTTP methods can return the status code 405 if the respective methods on the repository are not exported.

- HTTP GET – The HTTP GET on the endpoint uses `findById(Id)` method and returns 404 if the resource is not found.
- HTTP PUT and PATCH – Both HTTP PUT and PATCH methods make use of `save(..)` method on the repository. To know more about their differences read HTTP PUT vs HTTP PATCH methods.
- HTTP DELETE – The HTTP DELETE makes use of `delete(T)`, `delete(Id)`, or `delete(Iterable)` methods in the repository.
- HTTP HEAD – HTTP HEAD method is similar to HTTP GET that it makes use of `find(Id)` method. The only difference is that HEAD method does not return any content.

The Association Resource

Spring Data REST exposes sub-resources of every item resource for each of the associations the item resource has. The name and path of the resource defaults to the name of the association property and can be customized by using `@RestResource` on the association property.

Spring Data REST exports association resource if two entities have a relationship between them. To demonstrate this let's consider we have an Item entity that has a reference to Product and PurchaseOrder.



```
@Entity
@Data
public class Item {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    public long item_id;

    @ManyToOne
    @JoinColumn(name = "purchase_order_id")
    private PurchaseOrder purchaseOrder;

    @ManyToOne
    @JoinColumn(name = "product_id")
    private Product product;
}
```

Also, consider we have a dedicated repository for the Item.

```
@RepositoryRestResource
public interface ItemRepository
    extends CrudRepository<Item, Long> {
}
```

Now, let's access the Item resource that is exported by Spring Data REST.

```
http://localhost:8080/items
```

The response includes individual Item resources along with Association Resources.



```
{
  "_embedded": {
    "items": [ {
      "_links": {
        "self": {
          "href": "http://localhost:8080/items/1111"
        },
        "item": {
          "href": "http://localhost:8080/items/1111"
        },
        "purchaseOrder": {
          "href": "http://localhost:8080/items/1111/purchaseOrder"
        },
        "product": {
          "href": "http://localhost:8080/items/1111/product"
        }
      }
    }
  ],
  "_links": {
    "self": {
      "href": "http://localhost:8080/items"
    },
    "profile": {
      "href": "http://localhost:8080/profile/items"
    }
  }
}
```

As can be seen above, the response includes purchaseOrder and product resources.

```
http://localhost:8080/items/1111/purchaseOrder
http://localhost:8080/items/1111/product
```

Both of these nested resources are Association Resources because they are derived from the association between the entities.

Supported HTTP Methods

As the association resource represents an entity we can use HTTP GET, PUT, POST, and DELETE methods on them.

- HTTP GET – The GET Method returns 404 if the particular resource is not found or return 405 if respective query method is un-exported.
- HTTP POST – The HTTP POST works only when the association is of type collection and creates a new entity.
- HTTP PUT – The HTTP PUT method works on a single item association resource.
- HTTP DELETE – This methods deletes the association and return status code of 405 if that is not possible.

The Search Resource

The search resource returns links for all query methods exposed by a repository. The path and name of the query method resources can be modified using `@RestResource` on the method declaration.

Both of the collection and single item resource endpoints makes use of the default methods in repository. However, repositories can also have Spring Data – derived query methods. Spring DATA REST exposes these query methods through the Search Resources and related Query Method Resource (that we will see in the next section).

In order to enable a Search Resource, we will add a query method to our repository.

```
@RepositoryRestResource
public interface StudentRepository
    extends JpaRepository<Student, Long> {
    List<Student> findByFirstName(String firstName);
}
```

Doing this, Spring Data Repository exposes a new endpoint – `/students/search`. When we execute the Search resource `http://localhost:8080/students/search` we get the next output.



```
{
  "_links" : {
    "findByFirstName" : {
      "href" : "http://localhost:8080/students/search/findByFirstName{?firstName}",
      "templated" : true
    },
    "self" : {
      "href" : "http://localhost:8080/students/search"
    }
  }
}
```

As shown in the code block a Query Method Resource `findByFirstName` is now available.

Supported HTTP Methods

The Search Resource only supports HTTP GET and HTTP HEAD methods.

- HTTP GET – The HTTP GET method on the search resource returns a list of Query Method Resources each pointing to a query method in the repository.
- HTTP HEAD – The HTTP HEAD method doesn't return any data. However, if search resource is not available it returns status code of 404.

The Query Method Resource

The query method resource runs the exposed query through an individual query method on the repository interface. The Query Method Resource allows us to execute individual query methods. To do so, we need to use the query method as a resource and provide arguments in the form of query Strings.

In the previous section we have added `findByFirstName(firstName)` method to our repository. Thus, we can execute the query method endpoints like this.

```
http://localhost:8080/students/search/findByFirstName?firstName=Jon
```

And, obviously that returns list of resources matches by the given criteria.



```
{
  "_embedded": {
    "students": [ {
      "firstName": "Jon",
      "lastName": "Snow",
      "year": 2024,
      "_links": {
        "self": {
          "href": "http://localhost:8080/students/1"
        }
      },
      "student": {
        "href": "http://localhost:8080/students/1"
      }
    }
  ]
},
  "_links": {
    "self": {
      "href": "http://localhost:8080/students/search/findByFirstName?firstName=Jon"
    }
  }
}
```

Additionally, we can add Pageable to the query method argument.

```
List<Student> findByFirstName(String firstName, Pageable pageable);
```

That makes the pagination and sorting related arguments available to the Query Method resource like this

```
http://localhost:8080/students/search/findByFirstName{?firstName,page,size,sort}
```

Supported HTTP Methods

The Query Method Resource supports HTTP GET and HTTP HEAD methods.

- HTTP GET – Returns a list of resources that matches the query method arguments. If the



query methods supports pagination then we can use pagination and sorting related query methods.

- HTTP HEAD – Similar to the GET method, HTTP HEAD supports query method resources and their query parameters. However, it doesn't return any response. It will return status code of 404 if the query method resource is not found.

Spring Data REST Associations

Spring Data Rest allows to rapidly create a REST API to manipulate and query a database by exposing Spring Data repositories via its `@RepositoryRestResource` annotation.

One-to-One

Relationship

Data Model

Let's define two entity classes, Library and Address, having a one-to-one relationship by using the `@OneToOne` annotation. The association is owned by the Library end of the association:

```
@Entity
public class Library {

    @Id
    @GeneratedValue
    private long id;

    @Column
    private String name;

    @OneToOne
    @JoinColumn(name = "address_id")
    @RestResource(path = "libraryAddress", rel="address")
    private Address address;

    // standard constructor, getters, setters
}
```

```
@Entity
public class Address {

    @Id
    @GeneratedValue
    private long id;

    @Column(nullable = false)
    private String location;

    @OneToOne(mappedBy = "address")
    private Library library;

    // standard constructor, getters, setters
}
```

The `@RestResource` annotation is optional, and we can use it to customize the endpoint.

We must also be careful to have different names for each association resource. Otherwise, we'll encounter a `JsonMappingException` with the message "Detected multiple association links with same relation type! Disambiguate association."

The association name defaults to the property name, and we can customize it using the `rel` attribute of the `@RestResource` annotation:

If we were to add the `secondaryAddress` property above to the `Library` class, we'd have two resources named `address`, thus encountering a conflict.

We can resolve this by specifying a different value for the `rel` attribute, or by omitting the `RestResource` annotation so that the resource name defaults to `secondaryAddress`.

The Repositories

In order to expose these entities as resources, we'll create two repository interfaces for each of them by extending the `CrudRepository` interface:

```
public interface LibraryRepository extends CrudRepository<Library, Long> {}
```



```
public interface AddressRepository extends CrudRepository<Address, Long> {}
```

Creating the Resources

First, we'll add a Library instance to work with:

```
curl -i -X POST -H "Content-Type:application/json"
-d '{"name":"My Library"}' http://localhost:8080/libraries
```

Then the API returns the JSON object:

```
{
  "name" : "My Library",
  "_links" : {
    "self" : {
      "href" : "http://localhost:8080/libraries/1"
    },
    "library" : {
      "href" : "http://localhost:8080/libraries/1"
    },
    "address" : {
      "href" : "http://localhost:8080/libraries/1/libraryAddress"
    }
  }
}
```

Note that if we're using curl on Windows, we have to escape the double-quote character inside the String that represents the JSON body:

```
-d "{\"name\":\"My Library\"}"
```

We can see in the response body that an association resource has been exposed at the `libraries/{libraryId}/address` endpoint.

Before we create an association, sending a GET request to this endpoint will return an empty object.

However, if we want to add an association, we must first create an Address instance:



```
curl -i -X POST -H "Content-Type:application/json"
-d '{"location":"Main Street nr 5"}' http://localhost:8080/addresses
```

The result of the POST request is a JSON object containing the Address record:

```
{
  "location" : "Main Street nr 5",
  "_links" : {
    "self" : {
      "href" : "http://localhost:8080/addresses/1"
    },
    "address" : {
      "href" : "http://localhost:8080/addresses/1"
    },
    "library" : {
      "href" : "http://localhost:8080/addresses/1/library"
    }
  }
}
```

Creating the Associations

After persisting both instances, we can establish the relationship by using one of the association resources.

This is done using the HTTP method PUT, which supports a media type of text/uri-list, and a body containing the URI of the resource to bind to the association.

Since the Library entity is the owner of the association, we'll add an address to a library:

```
curl -i -X PUT -d "http://localhost:8080/addresses/1"
-H "Content-Type:text/uri-list" http://localhost:8080/libraries/1/libraryAddress
```

If successful, it'll return status 204. To verify this, we can check the library association resource of the address:



```
curl -i -X GET http://localhost:8080/addresses/1/library
```

It should return the Library JSON object with the name “My Library.”

To remove an association, we can call the endpoint with the DELETE method, making sure to use the associationresource of the owner of the relationship:

```
curl -i -X GET http://localhost:8080/addresses/1/library
```

It should return the Library JSON object with the name “My Library.”

To remove an association, we can call the endpoint with the DELETE method, making sure to use the associationresource of the owner of the relationship:

```
curl -i -X DELETE http://localhost:8080/libraries/1/libraryAddress
```

One-to-Many Relationship

We define a one-to-many relationship using the @OneToMany and @ManyToOne annotations. We can also add the optional @RestResource annotation to customize the association resource.

The Data Model

To exemplify a one-to-many relationship, we'll add a new Book entity, which represents the “many” end of a relationship with the Library entity:



```
@Entity
public class Book {

    @Id
    @GeneratedValue
    private long id;

    @Column(nullable=false)
    private String title;

    @ManyToOne
    @JoinColumn(name="library_id")
    private Library library;

    // standard constructor, getter, setter
}
```

Then we'll add the relationship to the Library class as well:

```
public class Library {

    //...

    @OneToMany(mappedBy = "library")
    private List<Book> books;

    //...

}
```

The Repository

We also need to create a BookRepository:

```
public interface BookRepository extends CrudRepository<Book, Long> { }
```



The Association Resources

In order to add a book to a library, we need to create a Book instance first by using the /books collection resource:

```
curl -i -X POST -d '{"title":"Book1"}'  
-H "Content-Type:application/json" http://localhost:8080/books
```

And here's the response from the POST request:

```
{  
  "title" : "Book1",  
  "_links" : {  
    "self" : {  
      "href" : "http://localhost:8080/books/1"  
    },  
    "book" : {  
      "href" : "http://localhost:8080/books/1"  
    },  
    "bookLibrary" : {  
      "href" : "http://localhost:8080/books/1/library"  
    }  
  }  
}
```

In the response body, we can see that the association endpoint, /books/{bookId}/library, has been created.

Now let's associate the book with the library we created in the previous section by sending a PUT request to the association resource that contains the URI of the library resource:

```
curl -i -X PUT http://localhost:8080/libraries/1/books
```

The returned JSON object will contain a books array:

```
{
  "_embedded": {
    "books": [ {
      "title": "Book1",
      "_links": {
        "self": {
          "href": "http://localhost:8080/books/1"
        },
        "book": {
          "href": "http://localhost:8080/books/1"
        },
        "bookLibrary": {
          "href": "http://localhost:8080/books/1/library"
        }
      }
    }
  ],
  "_links": {
    "self": {
      "href": "http://localhost:8080/libraries/1/books"
    }
  }
}
```

To remove an association, we can use the DELETE method on the association resource:

```
curl -i -X DELETE http://localhost:8080/books/1/library
```

Many-to-Many Relationship

We define a many-to-many relationship using the `@ManyToMany` annotation, to which we can also add `@RestResource`.

The Data Model

To create an example of a many-to-many relationship, we'll add a new model class, `Author`, which has a many-to-many relationship with the `Book` entity:



```
@Entity
public class Author {

    @Id
    @GeneratedValue
    private long id;

    @Column(nullable = false)
    private String name;

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

    //standard constructors, getters, setters
}
```

Then we'll add the association in the Book class as well:

```
public class Book {

    //...

    @ManyToMany(mappedBy = "books")
    private List<Author> authors;

    //...
}
```

The Repository

Next, we'll create a repository interface to manage the Author entity:

```
public interface AuthorRepository extends CrudRepository<Author, Long> { }
```



The Association Resources

As in the previous sections, we must first create the resources before we can establish the association. We'll create an Author instance by sending a POST request to the /authors collection resource:

```
curl -i -X POST -H "Content-Type:application/json"
-d '{"name":"author1"}' http://localhost:8080/authors
```

Next, we'll add a second Book record to our database:

```
curl -i -X POST -H "Content-Type:application/json"
-d '{"title":"Book 2"}' http://localhost:8080/books
```

Then we'll execute a GET request on our Author record to view the association URL:

```
{
  "name" : "author1",
  "_links" : {
    "self" : {
      "href" : "http://localhost:8080/authors/1"
    },
    "author" : {
      "href" : "http://localhost:8080/authors/1"
    },
    "books" : {
      "href" : "http://localhost:8080/authors/1/books"
    }
  }
}
```

Now we can create an association between the two Book records and the Author record using the endpoint authors/1/books with the PUT method, which supports a media type of text/uri-list and can receive more than one URI.

To send multiple URIs, we have to separate them by a line break:



```
curl -i -X PUT -H "Content-Type:text/uri-list"  
--data-binary @uris.txt http://localhost:8080/authors/1/books
```

The uris.txt file contains the URIs of the books, each on a separate line:

```
http://localhost:8080/books/1  
http://localhost:8080/books/2
```

To verify both books are associated with the author, we can send a GET request to the association endpoint:

```
curl -i -X GET http://localhost:8080/authors/1/books
```

And we'll receive this response:



```
{
  "_embedded": {
    "books": [ {
      "title": "Book 1",
      "_links": {
        "self": {
          "href": "http://localhost:8080/books/1"
        }
      }
    },
    {
      "title": "Book 2",
      "_links": {
        "self": {
          "href": "http://localhost:8080/books/2"
        }
      }
    }
  ],
  "_links": {
    "self": {
      "href": "http://localhost:8080/authors/1/books"
    }
  }
}
```

To remove an association, we can send a request with the DELETE method to the URL of the association resource followed by {bookId}:

```
curl -i -X DELETE http://localhost:8080/authors/1/books/1
```

Testing the Endpoints with TestRestTemplate

Let's create a test class that injects a TestRestTemplate instance, and defines the constants we'll use:



```
@RunWith(SpringRunner.class)
@SpringBootTest(classes = SpringDataRestApplication.class,
webEnvironment = WebEnvironment.DEFINED_PORT)
public class SpringDataRelationshipsTest {

    @Autowired
    private TestRestTemplate template;

    private static String BOOK_ENDPOINT = "http://localhost:8080/books/";
    private static String AUTHOR_ENDPOINT = "http://localhost:8080/authors/";
    private static String ADDRESS_ENDPOINT = "http://localhost:8080/addresses/";
    private static String LIBRARY_ENDPOINT = "http://localhost:8080/libraries/";

    private static String LIBRARY_NAME = "My Library";
    private static String AUTHOR_NAME = "George Orwell";
}
```

Testing the One-to-One Relationship

We'll create a @Test method that saves Library and Address objects by making POST requests to the collection resources.

Then it saves the relationship with a PUT request to the association resource, and verifies that it's been established with a GET request to the same resource:



```
@Test
public void whenSaveOneToOneRelationship_thenCorrect() {
    Library library = new Library(LIBRARY_NAME);
    template.postForEntity(LIBRARY_ENDPOINT, library, Library.class);

    Address address = new Address("Main street, nr 1");
    template.postForEntity(ADDRESS_ENDPOINT, address, Address.class);

    HttpHeaders requestHeaders = new HttpHeaders();
    requestHeaders.add("Content-type", "text/uri-list");
    HttpEntity<String> httpEntity
        = new HttpEntity<>(ADDRESS_ENDPOINT + "/1", requestHeaders);
    template.exchange(LIBRARY_ENDPOINT + "/1/libraryAddress",
        HttpMethod.PUT, httpEntity, String.class);

    ResponseEntity<Library> libraryGetResponse
        = template.getForEntity(ADDRESS_ENDPOINT + "/1/library", Library.class);
    assertEquals("library is incorrect",
        libraryGetResponse.getBody().getName(), LIBRARY_NAME);
}
```

Testing the One-to-Many Relationship

Now we'll create a `@Test` method that saves a `Library` instance and two `Book` instances, sends a `PUT` request to each `Book` object's `/library` association resource, and verifies that the relationship has been saved:



```
@Test
public void whenSaveOneToManyRelationship_thenCorrect() {
    Library library = new Library(LIBRARY_NAME);
    template.postForEntity(LIBRARY_ENDPOINT, library, Library.class);

    Book book1 = new Book("Dune");
    template.postForEntity(BOOK_ENDPOINT, book1, Book.class);

    Book book2 = new Book("1984");
    template.postForEntity(BOOK_ENDPOINT, book2, Book.class);

    HttpHeaders requestHeaders = new HttpHeaders();
    requestHeaders.add("Content-Type", "text/uri-list");
    HttpEntity<String> bookHttpEntity
        = new HttpEntity<>(LIBRARY_ENDPOINT + "/1", requestHeaders);
    template.exchange(BOOK_ENDPOINT + "/1/library",
        HttpMethod.PUT, bookHttpEntity, String.class);
    template.exchange(BOOK_ENDPOINT + "/2/library",
        HttpMethod.PUT, bookHttpEntity, String.class);

    ResponseEntity<Library> libraryGetResponse =
        template.getForEntity(BOOK_ENDPOINT + "/1/library", Library.class);
    assertEquals("library is incorrect",
        libraryGetResponse.getBody().getName(), LIBRARY_NAME);
}
```

Testing the Many-to-Many Relationship

For testing the many-to-many relationship between Book and Author entities, we'll create a test method that saves one Author record and two Book records.

Then it sends a PUT request to the /books association resource with the two Book's URIs, and verifies that the relationship has been established:



```
@Test
public void whenSaveManyToManyRelationship_thenCorrect() {
    Author author1 = new Author(AUTHOR_NAME);
    template.postForEntity(AUTHOR_ENDPOINT, author1, Author.class);

    Book book1 = new Book("Animal Farm");
    template.postForEntity(BOOK_ENDPOINT, book1, Book.class);

    Book book2 = new Book("1984");
    template.postForEntity(BOOK_ENDPOINT, book2, Book.class);

    HttpHeaders requestHeaders = new HttpHeaders();
    requestHeaders.add("Content-type", "text/uri-list");
    HttpEntity<String> httpEntity = new HttpEntity<>(
        BOOK_ENDPOINT + "/1\n" + BOOK_ENDPOINT + "/2", requestHeaders);
    template.exchange(AUTHOR_ENDPOINT + "/1/books",
        HttpMethod.PUT, httpEntity, String.class);

    String jsonResponse = template
        .getForObject(BOOK_ENDPOINT + "/1/authors", String.class);
    JSONObject jsonObj = new JSONObject(jsonResponse).getJSONObject("_embedded");
    JSONArray jsonArray = jsonObj.getJSONArray("authors");
    assertEquals("author is incorrect",
        jsonArray.getJSONObject(0).getString("name"), AUTHOR_NAME);
}
```

Building a Web Application with Spring Boot and

Angular The Maven Dependencies

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
  <groupId>com.h2database</groupId>
  <artifactId>h2</artifactId>
  <scope>runtime</scope>
</dependency>
```

Note that we included spring-boot-starter-web because we'll use it for creating the REST service, and spring-boot-starter-jpa for implementing the persistence layer.

The JPA Entity Class

To quickly prototype our application's domain layer, let's define a simple JPA entity class, which will be responsible for modelling users:

```
@Entity
public class User {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private long id;
    private final String name;
    private final String email;

    // standard constructors / setters / getters / toString
}
```

The UserRepository Interface

Since we'll need basic CRUD functionality on the User entities, we must also define a UserRepository interface:

```
@Repository
public interface UserRepository extends CrudRepository<User, Long>{}
```



The REST Controller

Now let's implement the REST API. In this case, it's just a simple REST controller:

```
@RestController
@CrossOrigin(origins = "http://localhost:4200")
public class UserController {

    // standard constructors

    private final UserRepository userRepository;

    @GetMapping("/users")
    public List<User> getUsers() {
        return (List<User>) userRepository.findAll();
    }

    @PostMapping("/users")
    void addUser(@RequestBody User user) {
        userRepository.save(user);
    }
}
```

There's nothing inherently complex in the definition of the UserController class.

Of course, the implementation detail worth noting here is the use of the @CrossOrigin annotation. As the name implies, the annotation enables Cross-Origin Resource Sharing (CORS) on the server.

This step isn't always necessary, but since we're deploying our Angular frontend to <http://localhost:4200>, and our Bootbackend to <http://localhost:8080>, the browser would otherwise deny requests from one to the other.

Regarding the controller methods, `getUser()` fetches all the User entities from the database. Similarly, the `addUser()` method persists a new entity in the database, which is passed in the request body.

To keep things simple, we deliberately left out the controller implementation triggering Spring Boot validation before persisting an entity. In production, however, we can't trust user input alone, so server-side validation should be a mandatory feature.

Bootstrapping the Spring Boot Application



Finally, let's create a standard Spring Boot bootstrapping class, and populate the database with a few User entities:

```
@SpringBootApplication
public class Application {

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

    @Bean
    CommandLineRunner init(UserRepository userRepository) {
        return args -> {
            Stream.of("John", "Julie", "Jennifer", "Helen", "Rachel").forEach(name -> {
                User user = new User(name, name.toLowerCase() + "@domain.com");
                userRepository.save(user);
            });
            userRepository.findAll().forEach(System.out::println);
        };
    }
}
```

Now let's run the application. As expected, we should see a list of User entities printed out to the console on startup:

```
User{id=1, name=John, email=john@domain.com}
User{id=2, name=Julie, email=julie@domain.com}
User{id=3, name=Jennifer, email=jennifer@domain.com}
User{id=4, name=Helen, email=helen@domain.com}
User{id=5, name=Rachel, email=rachel@domain.com}
```

The Angular Application

With our demo Spring Boot application up and running, we can now create a simple Angular application capable of consuming the REST controller API.



Angular CLI Installation

We'll use Angular CLI, a powerful command-line utility, to create our Angular application.

Angular CLI is an extremely valuable tool since it allows us to create an entire Angular project from scratch, generating components, services, classes, and interfaces with just a few commands.

Once we've installed npm (Node Package Manager), we'll open a command console and type the command:

```
npm install -g @angular/cli@1.7.4
```

That's it. The above command will install the latest version of Angular CLI.

Project Scaffolding With Angular CLI

We can generate our Angular application structure from the ground up, but honestly, this is an error-prone and time-consuming task that we should avoid in all cases.

Instead, we'll let Angular CLI do the hard work for us. So we can open a command console, then navigate to the folder where we want our application to be created, and type the command:

```
ng new angularclient
```

The new command will generate the entire application structure within the angularclient directory.

The Angular Application's Entry Point

If we look inside the angularclient folder, we'll see that Angular CLI has effectively created an entire project for us.

Angular's application files use TypeScript, a typed superset of JavaScript that compiles to plain JavaScript. However, the entry point of any Angular application is a plain old index.html file.

Let's edit this file:



```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Spring Boot - Angular Application</title>
  <base href="/">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="icon" type="image/x-icon" href="favicon.ico">
  <link rel="stylesheet"
    href="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0/css/bootstrap.min.css"
    integrity="sha384-Gn5384xqQ1aoWXA+058RXPxPg6fy4IWvTNh0E263XmFcJISAwIGgFAW/dAiS6JXm"
    crossorigin="anonymous">
</head>
<body>
  <app-root></app-root>
</body>
</html>
```

As we can see above, we included Bootstrap 4 so we can give our application UI components a more fancy look. Of course, it's possible to pick up another UI kit from the bunch available out there.

Please notice the custom `<app-root></app-root>` tags inside the `<body>` section. At first glance, they look rather weird, as `<app-root>` is not a standard HTML 5 element.

We'll keep them there, as `<app-root>` is the root selector that Angular uses for rendering the application's root component.

The app.component.ts Root Component

To better understand how Angular binds an HTML template to a component, let's go to the `src/app` directory and edit the `app.component.ts` TypeScript file, the root component:



```
import { Component } from '@angular/core';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {

  title: string;

  constructor() {
    this.title = 'Spring Boot - Angular Application';
  }
}
```

For obvious reasons, we won't dive deep into learning TypeScript. Even so, let's note that the file defines an AppComponent class, which declares a field title of type string (lower-cased). Definitely, it's typed JavaScript.

Additionally, the constructor initializes the field with a string value, which is pretty similar to what we do in Java.

The most relevant part is the @Component metadata marker or decorator, which defines three elements:

- selector – the HTML selector used to bind the component to the HTML template file
- templateUrl – the HTML template file associated with the component
- styleUrls – one or more CSS files associated with the component

As expected, we can use the app.component.html and app.component.css files to define the HTML template and the CSS styles of the root component.

Finally, the selector element binds the whole component to the <app-root> selector included in the index.html file.

The app.component.html File

Since the app.component.html file allows us to define the root component's HTML template, the AppComponent class, we'll use it for creating a basic navigation bar with two buttons.

If we click the first button, Angular will display a table containing the list of User entities stored in the database.



Similarly, if we click the second one, it will render an HTML form, which we can use for adding new entities to the database:

```
<div class="container">
  <div class="row">
    <div class="col-md-12">
      <div class="card bg-dark my-5">
        <div class="card-body">
          <h2 class="card-title text-center text-white py-3">{{ title }}</h2>
          <ul class="text-center list-inline py-3">
            <li class="list-inline-item">
              <a routerLink="/users" class="btn btn-info">List Users</a>
            </li>
            <li class="list-inline-item">
              <a routerLink="/adduser" class="btn btn-info">Add User</a>
            </li>
          </ul>
        </div>
      </div>
    </div>
  </div>
</div>
```

The bulk of the file is standard HTML, with a few caveats worth noting.

The first one is the `{{ title }}` expression. The double curly braces `{{ variable-name }}` is the placeholder that Angular uses for performing variable interpolation.

Let's keep in mind that the AppComponent class initialized the title field with the value Spring Boot – Angular Application. Thus, Angular will display the value of this field in the template. Likewise, changing the value in the constructor will be reflected in the template.

The second thing to note is the routerLink attribute.

Angular uses this attribute for routing requests through its routing module (more on this later). For now, it's sufficient to know that the module will dispatch a request to the `/users` path to a specific component and a request to `/adduser` to another component.



In each case, the HTML template associated with the matching component will be rendered within the `<router-outlet></router-outlet>` placeholder.

The User Class

Since our Angular application will fetch from and persist User entities in the database, let's implement a simple domain model with TypeScript.

Let's open a terminal console and create a model directory:

```
ng generate class user
```

Angular CLI will generate an empty User class, so let's populate it with a few fields:

```
export class User {  
  id: string;  
  name: string;  
  email: string;  
}
```

The UserService Service

With our client-side domain User class already set, we can now implement a service class that performs GET and POST requests to the `http://localhost:8080/users` endpoint.

This will allow us to encapsulate access to the REST controller in a single class, which we can reuse throughout the entire application.

Let's open a console terminal, then create a service directory, and within that directory, issue the following command:

```
ng generate service user-service
```

Now let's open the `user.service.ts` file that Angular CLI just created and refactor it:



```
import { Injectable } from '@angular/core';
import { HttpClient, HttpHeaders } from '@angular/common/http';
import { User } from '../model/user';
import { Observable } from 'rxjs/Observable';

@Injectable()
export class UserService {

    private usersUrl: string;

    constructor(private http: HttpClient) {
        this.usersUrl = 'http://localhost:8080/users';
    }

    public findAll(): Observable<User[]> {
        return this.http.get<User[]>(this.usersUrl);
    }

    public save(user: User) {
        return this.http.post<User>(this.usersUrl, user);
    }
}
```

We don't need a solid background on TypeScript to understand how the UserService class works. Simply put, it encapsulates within a reusable component all the functionality required to consume the REST controller API that we implemented before in Spring Boot.

The findAll() method performs a GET HTTP request to the http://localhost:8080/users endpoint via Angular's HttpClient. The method returns an Observable instance that holds an array of User objects.

Likewise, the save() method performs a POST HTTP request to the http://localhost:8080/users endpoint.

By specifying the type User in the HttpClient's request methods, we can consume back-end responses in an easier and more effective way.

Lastly, let's note the use of the @Injectable() metadata marker. This signals that the service should be created and injected via Angular's dependency injectors.



The UserListComponent Component

In this case, the UserService class is the thin middle-tier between the REST service and the application's presentation layer. Therefore, we need to define a component responsible for rendering the list of User entities persisted in the database.

Let's open a terminal console, then create a user-list directory, and generate a user list component:

```
ng generate component user-list
```

Angular CLI will generate an empty component class that implements the ngOnInit interface. The interface declares a hook ngOnInit() method, which Angular calls after it has finished instantiating the implementing class, and also after calling its constructor.

Let's refactor the class so that it can take a UserService instance in the constructor:

```
import { Component, OnInit } from '@angular/core';
import { User } from '../model/user';
import { UserService } from '../service/user.service';

@Component({
  selector: 'app-user-list',
  templateUrl: './user-list.component.html',
  styleUrls: ['./user-list.component.css']
})
export class UserListComponent implements OnInit {

  users: User[];

  constructor(private userService: UserService) {
  }

  ngOnInit() {
    this.userService.findAll().subscribe(data => {
      this.users = data;
    });
  }
}
```



The implementation of the `UserListComponent` class is pretty self-explanatory. It simply uses the `UserService`'s `findAll()` method to fetch all the entities persisted in the database and stores them in the `users` field.

In addition, we need to edit the component's HTML file, `user-list.component.html`, to create the table that displays the list of entities:

```
<div class="card my-5">
  <div class="card-body">
    <table class="table table-bordered table-striped">
      <thead class="thead-dark">
        <tr>
          <th scope="col">#</th>
          <th scope="col">Name</th>
          <th scope="col">Email</th>
        </tr>
      </thead>
      <tbody>
        <tr *ngFor="let user of users">
          <td>{{ user.id }}</td>
          <td>{{ user.name }}</td>
          <td><a href="mailto:{{ user.email }}">{{ user.email }}</a></td>
        </tr>
      </tbody>
    </table>
  </div>
</div>
```

We should note the use of the `*ngFor` directive. The directive is called a repeater, and we can use it for iterating over the contents of a variable and iteratively rendering HTML elements. In this case, we used it for dynamically rendering the table's rows.

In addition, we used variable interpolation for showing the id, name, and email of each user.

The UserFormComponent Component

Similarly, we need to create a component that allows us to persist a new `User` object in the database.

Let's create a `user-form` directory and type the following:



ng generate component user-form

Next let's open the user-form.component.ts file, and add to the UserFormComponent class a method for saving a User object:

```
import { Component } from '@angular/core';
import { ActivatedRoute, Router } from '@angular/router';
import { UserService } from '../service/user.service';
import { User } from '../model/user';

@Component({
  selector: 'app-user-form',
  templateUrl: './user-form.component.html',
  styleUrls: ['./user-form.component.css']
})
export class UserFormComponent {

  user: User;

  constructor(
    private route: ActivatedRoute,
    private router: Router,
    private userService: UserService) {
    this.user = new User();
  }

  onSubmit() {
    this.userService.save(this.user).subscribe(result => this.gotoUserList());
  }

  gotoUserList() {
    this.router.navigate(['/users']);
  }
}
```

In this case, UserFormComponent also takes a UserService instance in the constructor, which the onSubmit() method uses for saving the supplied User object.

Since we need to re-display the updated list of entities once we have persisted a new one, we call the



gotoUserList() method after the insertion, which redirects the user to the /users path.

In addition, we need to edit the user-form.component.html file, and create the HTML form for persisting a new user in the database:

```
<div class="card my-5">
  <div class="card-body">
    <form (ngSubmit)="onSubmit()" #userForm="ngForm">
      <div class="form-group">
        <label for="name">Name</label>
        <input type="text" [(ngModel)]="user.name"
          class="form-control"
          id="name"
          name="name"
          placeholder="Enter your name"
          required #name="ngModel">
      </div>
      <div [hidden]="!name.pristine" class="alert alert-danger">Name is required</div>
      <div class="form-group">
        <label for="email">Email</label>
        <input type="text" [(ngModel)]="user.email"
          class="form-control"
          id="email"
          name="email"
          placeholder="Enter your email address"
          required #email="ngModel">
      <div [hidden]="!email.pristine" class="alert alert-danger">Email is required</div>
      </div>
      <button type="submit" [disabled]="!userForm.form.valid"
        class="btn btn-info">Submit</button>
    </form>
  </div>
</div>
```

At a glance, the form looks pretty standard, but it encapsulates a lot of Angular's functionality behind the scenes.

Let's note the use of the ngSubmit directive, which calls the onSubmit() method when the form is submitted.

Next we defined the template variable #userForm, so Angular automatically adds an NgForm directive, which allows us to keep track of the form as a whole.



The NgForm directive holds the controls that we created for the form elements with an ngModel directive and a name attribute. It also monitors their properties, including their state.

The ngModel directive gives us two-way data binding functionality between the form controls and the client-side domain model, the User class.

This means that data entered in the form input fields will flow to the model, and the other way around. Changes in both elements will be reflected immediately via DOM manipulation.

Additionally, ngModel allows us to keep track of the state of each form control, and perform client-side validation by adding different CSS classes and DOM properties to each control.

In the above HTML file, we used the properties applied to the form controls only to display an alert box when the values in the form have been changed.

The app-routing.module.ts File

Although the components are functional in isolation, we still need to use a mechanism for calling them when the user clicks the buttons in the navigation bar.

This is where the RouterModule comes into play. Let's open the app-routing.module.ts file and configure the module, so it can dispatch requests to the matching components:

```
import { NgModule } from '@angular/core';
import { Routes, RouterModule } from '@angular/router';
import { UserListComponent } from './user-list/user-list.component';
import { UserFormComponent } from './user-form/user-form.component';

const routes: Routes = [
  { path: 'users', component: UserListComponent },
  { path: 'adduser', component: UserFormComponent }
];

@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule { }
```



As we can see above, the Routes array instructs the router which component to display when a user clicks a link or specifies a URL into the browser address bar.

A route is composed of two parts:

1. Path – a string that matches the URL in the browser address bar
2. Component – the component to create when the route is active (navigated)

If the user clicks the List Users button, which links to the /users path, or enters the URL in the browser address bar, the router will render the UserListComponent component's template file in the <router-outlet> placeholder.

Likewise, if they click the Add User button, it will render the UserFormComponent component.

The app.module.ts File

Next we need to edit the app.module.ts file, so Angular can import all the required modules, components, and services.

Additionally, we need to specify which provider we'll use for creating and injecting the UserService class. Otherwise, Angular won't be able to inject it into the component classes:



```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { AppRoutingModuleModule } from './app-routing.module';
import { FormsModule } from '@angular/forms';
import { HttpClientModule } from '@angular/common/http';
import { AppComponent } from './app.component';
import { UserListComponent } from './user-list/user-list.component';
import { UserFormComponent } from './user-form/user-form.component';
import { UserService } from './service/user.service';
```

```
@NgModule({
  declarations: [
    AppComponent,
    UserListComponent,
    UserFormComponent
  ],
  imports: [
    BrowserModule,
    AppRoutingModuleModule,
    HttpClientModule,
    FormsModule
  ],
  providers: [UserService],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Running the Application

Finally, we're ready to run our application.

To accomplish this, we'll first run the Spring Boot application, so the REST service is alive and listening for requests. Once the Spring Boot application has been started, we'll open a command console and type the following command:

```
ng serve --open
```



This will start Angular's live development server and also open the browser at <http://localhost:4200>.

We should see the navigation bar with the buttons for listing existing entities and for adding new ones. If we click the first button, we should see below the navigation bar a table with the list of entities persisted in the database:

Spring Boot - Angular Application

[List Users](#)
[Add User](#)

| # | Name | Email |
|---|----------|--|
| 1 | John | john@domain.com |
| 2 | Julie | julie@domain.com |
| 3 | Jennifer | jennifer@domain.com |
| 4 | Helen | helen@domain.com |
| 5 | Rachel | rachel@domain.com |

Similarly, clicking the second button will display the HTML form for persisting a new entity:



Spring Boot - Angular Application

[List Users](#)
[Add User](#)

Name

Name is required

Email

Email is required

Spring Security

Spring Security is a framework which provides various security features like: authentication, authorization to create secure Java Enterprise Applications.

It is a sub-project of Spring framework which was started in 2003 by Ben Alex. Later on, in 2004, It was released under the Apache License as Spring Security 2.0.0.

It overcomes all the problems that come during creating non spring security applications and manage new server environment for the application.

This framework targets two major areas of application are authentication and authorization.

Authentication is the process of knowing and identifying the user that wants to access.

Authorization is the process to allow authority to perform actions in the application. We can apply authorization to authorize web request, methods and access to individual domain.

Spring Security framework supports wide range of authentication models. These models either provided by



thirdparties or framework itself. Spring Security supports integration with all of these technologies.

- Digest authentication headers
- HTTP X.509 client certificate exchange
- LDAP (Lighweight Directory Access Protocol)
- Form-based authentication
- OpenID authentication
- Automatic remember-me authentication
- Kerberos
- JOSSO (Java Open Source HTTP BASIC authentication headers)
- HTTP Single Sign-On)
- AppFuse
- AndroMDA
- Mule ESB
- DWR(Direct Web Request)

The beauty of this framework is its flexible authentication nature to integrate with any software solution. Sometimes, developers want to integrate it with a legacy system that does not follow any security standard, there Spring Security works nicely.

Advantages

Spring Security has numerous advantages. Some of that are given below.

- Comprehensive support for authentication and authorization.
- Protection against common tasks
- Servlet API integration
- Integration with Spring MVC
- Portability
- CSRF protection
- Java Configuration support

Spring Security Features

- **LDAP (Lightweight Directory Access Protocol)**

It is an open application protocol for maintaining and accessing distributed directory information services over an Internet Protocol.



- **Single sign-on**

This feature allows a user to access multiple applications with the help of single account(user name and password).

- **JAAS (Java Authentication and Authorization Service) LoginModule**

It is a Pluggable Authentication Module implemented in Java. Spring Security supports it for its authentication process.

- **Basic Access Authentication**

Spring Security supports Basic Access Authentication that is used to provide user name and password while making request over the network.

- **Digest Access Authentication**

This feature allows us to make authentication process more secure than Basic Access Authentication. It asks the browser to confirm the identity of the user before sending sensitive data over the network.

- **Remember-me**

Spring Security supports this feature with the help of HTTP Cookies. It remembers the user and avoids login again from the same machine until the user logs out.

- **Web Form Authentication**

In this process, web forms collect and authenticate user credentials from the web browser. Spring Security supports it while we want to implement web form authentication.

- **Authorization**

Spring Security provides this feature to authorize the user before accessing resources. It allows developers to define access policies against the resources.

- **Software Localization**

This feature allows us to make application user interface in any language.

- **HTTP Authorization**

Spring provides this feature for HTTP authorization of web request URLs using Apache Ant paths or regular expressions.

Example: Login with Spring Security

- **Maven Dependencies**



When working with Spring Boot, the spring-boot-starter-security starter will automatically include all dependencies, such as spring-security-core, spring-security-web, and spring-security-config among others:

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-security</artifactId>
  <version>2.3.3.RELEASE</version>
</dependency>
```

- **Spring Security Java Configuration**

Let's start by creating a Spring Security configuration class that extends WebSecurityConfigurerAdapter. By adding @EnableWebSecurity, we get Spring Security and MVC integration support:

```
@Configuration
@EnableWebSecurity
public class SecSecurityConfig extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(final AuthenticationManagerBuilder auth) throws Exception {
        // authentication manager (see below)
    }

    @Override
    protected void configure(final HttpSecurity http) throws Exception {
        // http builder configurations for authorize requests and form login (see below)
    }
}
```

In this example, we used in-memory authentication and defined three users. Next, we'll go through the elements we used to create the form login configuration.

- **Authentication Manager**

The Authentication Provider is backed by a simple, in-memory implementation, InMemoryUserDetailsManager. This is useful for rapid prototyping when a full persistence mechanism is not yet necessary:



```
protected void configure(final AuthenticationManagerBuilder auth) throws Exception {
    auth.inMemoryAuthentication()
        .withUser("user1").password(passwordEncoder().encode("user1Pass")).roles("USER")
        .and()
        .withUser("user2").password(passwordEncoder().encode("user2Pass")).roles("USER")
        .and()
        .withUser("admin").password(passwordEncoder().encode("adminPass")).roles("ADMIN");
}
```

Here we'll configure three users with the username, password, and role hard-coded. Starting with Spring 5, we also have to define a password encoder. In our example, we'll use the BCryptPasswordEncoder:

```
@Bean
public PasswordEncoder passwordEncoder() {
    return new BCryptPasswordEncoder();
}
```

- **Configuration to Authorize Requests**

We'll start by doing the necessary configurations to Authorize Requests.

Here we're allowing anonymous access on /login so that users can authenticate. We'll restrict /admin to ADMIN roles and securing everything else:

```
@Override
protected void configure(final HttpSecurity http) throws Exception {
    http
        .csrf().disable()
        .authorizeRequests()
        .antMatchers("/admin/**").hasRole("ADMIN")
        .antMatchers("/anonymous*").anonymous()
        .antMatchers("/login*").permitAll()
        .anyRequest().authenticated()
        .and()
        // ...
}
```



Note that the order of the `antMatchers()` elements is significant; the more specific rules need to come first, followed by the more general ones.

- **Configuration for Form Login**

We will now extend the above configuration for form login and logout:

```
@Override
protected void configure(final HttpSecurity http) throws Exception {
    http
        // ...
        .and()
        .formLogin()
        .loginPage("/login.html")
        .loginProcessingUrl("/perform_login")
        .defaultSuccessUrl("/homepage.html", true)
        .failureUrl("/login.html?error=true")
        .failureHandler(authenticationFailureHandler())
        .and()
        .logout()
        .logoutUrl("/perform_logout")
        .deleteCookies("JSESSIONID")
        .logoutSuccessHandler(logoutSuccessHandler());
}
```

- `loginPage()` – the custom login page
- `loginProcessingUrl()` – the URL to submit the username and password to
- `defaultSuccessUrl()` – the landing page after a successful login
- `failureUrl()` – the landing page after an unsuccessful login
- `logoutUrl()` – the custom logout

- **Add Spring Security to the Web Application**

To use the above-defined Spring Security configuration, we need to attach it to the web application. We'll use the `WebApplicationInitializer`, so we don't need to provide any `web.xml`:



```
public class AppInitializer implements WebApplicationInitializer {

    @Override
    public void onStartup(ServletContext sc) {

        AnnotationConfigWebApplicationContext root = new AnnotationConfigWebApplicationContext();
        root.register(SecSecurityConfig.class);

        sc.addListener(new ContextLoaderListener(root));

        sc.addFilter("securityFilter", new DelegatingFilterProxy("springSecurityFilterChain"))
            .addMappingForUrlPatterns(null, false, "/*");
    }
}
```

Note that this initializer isn't necessary if we're using a Spring Boot application.

- **The Spring Security XML Configuration**

Let's also have a look at the corresponding XML configuration.

The overall project is using Java configuration, so we need to import the XML configuration file via a `Java@Configuration` class:

```
@Configuration
@ImportResource({ "classpath:webSecurityConfig.xml" })
public class SecSecurityConfig {
    public SecSecurityConfig() {
        super();
    }
}
```

And the Spring Security XML Configuration, `webSecurityConfig.xml`:



```
<http use-expressions="true">
  <intercept-url pattern="/login*" access="isAnonymous()" />
  <intercept-url pattern="/**" access="isAuthenticated()" />

  <form-login login-page="/login.html"
    default-target-url="/homepage.html"
    authentication-failure-url="/login.html?error=true" />
  <logout logout-success-url="/login.html" />
</http>

<authentication-manager>
  <authentication-provider>
    <user-service>
      <user name="user1" password="user1Pass" authorities="ROLE_USER" />
    </user-service>
    <password-encoder ref="encoder" />
  </authentication-provider>
</authentication-manager>

<beans:bean id="encoder"
  class="org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder">
</beans:bean>
```

- **The web.xml**

Before the introduction of Spring 4, we used to configure Spring Security in the web.xml; only an additional filter added to the standard Spring MVC web.xml:



```
<display-name>Spring Secured Application</display-name>

<!-- Spring MVC -->
<!-- ... -->

<!-- Spring Security -->
<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>
```

The filter – DelegatingFilterProxy – simply delegates to a Spring-managed bean – the FilterChainProxy – which itself is able to benefit from full Spring bean life-cycle management and such.

- **The Login Form**

The login form page is going to be registered with Spring MVC using the straightforward mechanism to map viewnames to URLs. Furthermore, there is no need for an explicit controller in between:

```
registry.addViewController("/login.html");
```

This, of course, corresponds to the login.jsp:



```
<html>
<head></head>
<body>
  <h1>Login</h1>
  <form name='f' action='login' method='POST'>
    <table>
      <tr>
        <td>User:</td>
        <td><input type='text' name='username' value=''></td>
      </tr>
      <tr>
        <td>Password:</td>
        <td><input type='password' name='password' /></td>
      </tr>
      <tr>
        <td><input name='submit' type='submit' value='submit' /></td>
      </tr>
    </table>
  </form>
</body>
</html>
```

The Spring Login form has the following relevant artifacts:

- login – the URL where the form is POSTed to trigger the authentication process
- username – the username
- password – the password
- **Further Configuring Spring Login**

We briefly discussed a few configurations of the login mechanism when we introduced the Spring Security Configuration above. Now let's go into some greater detail.

One reason to override most of the defaults in Spring Security is to hide that the application is secured with SpringSecurity. We also want to minimize the information a potential attacker knows about the application.

Fully configured, the login element looks like this:



```
@Override
protected void configure(HttpSecurity http) throws Exception {
    http.formLogin()
        .loginPage("/login.html")
        .loginProcessingUrl("/perform_login")
        .defaultSuccessUrl("/homepage.html",true)
        .failureUrl("/login.html?error=true")
}
```

Or the corresponding XML configuration:

```
<form-login
    login-page="/login.html"
    login-processing-url="/perform_login"
    default-target-url="/homepage.html"
    authentication-failure-url="/login.html?error=true"
    always-use-default-target="true"/>
```

- **The Login Page**

Next we'll configure a custom login page using the `loginPage()` method:

```
http.formLogin()
    .loginPage("/login.html")
```

Similarly, we can use the XML configuration:

```
login-page="/login.html"
```

If we don't specify this, Spring Security will generate a very basic Login Form at the `/login` URL.

- **The POST URL for Login**

The default URL where the Spring Login will POST to trigger the authentication process is `/login`, which used to be `/j_spring_security_check` before Spring Security 4.

We can use the `loginProcessingUrl` method to override this URL:



```
http.formLogin()  
.loginProcessingUrl("/perform_login")
```

We can also use the XML configuration:

```
login-processing-url="/perform_login"
```

By overriding this default URL, we're concealing that the application is actually secured with Spring Security. This information should not be available externally.

- **The Landing Page on Success**

After successfully logging in, we will be redirected to a page that by default is the root of the web application. We can override this via the `defaultSuccessUrl()` method:

```
http.formLogin()  
.defaultSuccessUrl("/homepage.html")
```

Or with XML configuration:

```
default-target-url="/homepage.html"
```

If the `always-use-default-target` attribute is set to `true`, then the user is always redirected to this page. If that attribute is set to `false`, then the user will be redirected to the previous page they wanted to visit before being prompted to authenticate.

- **The Landing Page on Failure**

Similar to the Login Page, the Login Failure Page is autogenerated by Spring Security at `/login?error` by default.

To override this, we can use the `failureUrl()` method:

```
http.formLogin()  
.failureUrl("/login.html?error=true")
```

Or with XML:



```
authentication-failure-url="/login.html?error=true"
```

When the project runs locally, the sample HTML can be accessed at:

```
http://localhost:8080/spring-security-mvc-login/login.html
```

Micro Services

Microservice Architecture is a Service Oriented Architecture. In the microservice architecture, there are a large number of microservices. By combining all the microservices, it constructs a big service. In the microservice architecture, all the services communicate with each other.

Definition: According to Sam Newman, "Microservices are the small services that work together."

According to James Lewis and Martin Fowler, "The microservice architectural style is an approach to develop a single application as a suite of small services. Each microservice runs its process and communicates with lightweight mechanisms. These services are built around business capabilities and independently developed by fully automated deployment machinery."

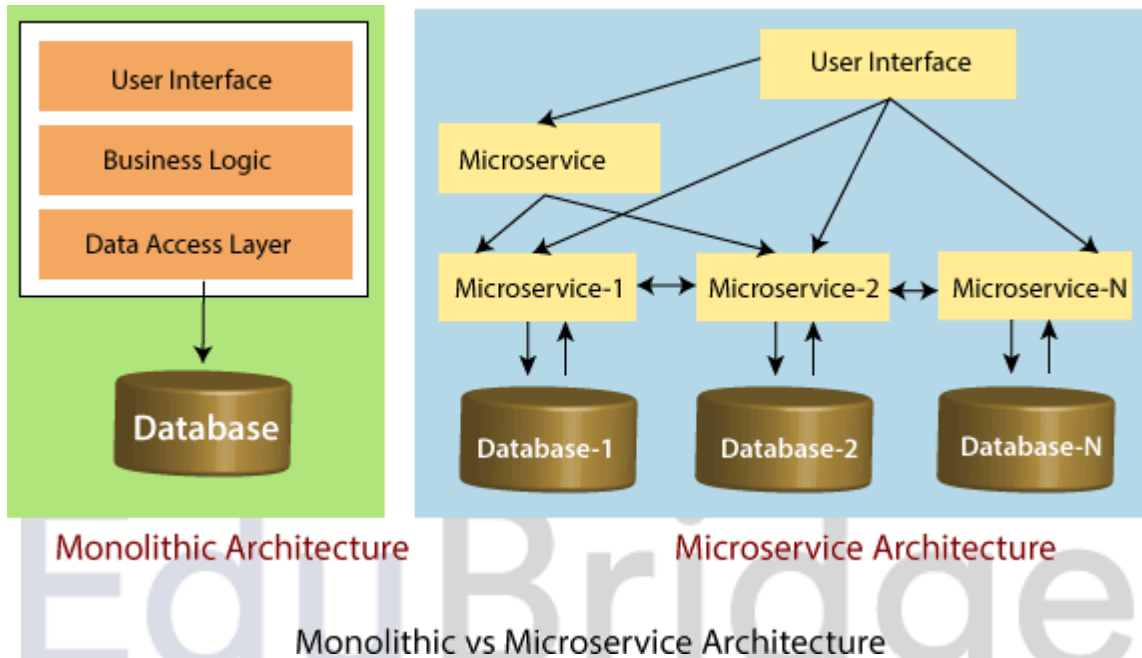
There is a bare minimum of centralized management of these services, which may be written in different programming language and use different data storage technologies.

Points to remember

- These are the services which are exposed by REST.
- These are small well-chosen deployable units.
- The services must be cloud-enabled.

The microservice defines an approach to the architecture that divides an application into a pool of loosely coupled services that implements business requirements. It is next to Service-Oriented Architecture (SOA). The most important feature of the microservice-based architecture is that it can perform continuous delivery of a large and complex application.

Microservice helps in breaking the application and build a logically independent smaller applications. For example, we can build a cloud application with the help of Amazon AWS with minimum efforts.



In the above figure, each microservice has its own business layer and database. If we change in one microservice, it does not affect the other services. These services communicate with each other by using lightweight protocols such as HTTP or REST or messaging protocols.

Principles of Microservices

There are the following principles of Microservices:

Single Responsibility Principle

The single responsibility principle states that a class or a module in a program should have only one responsibility. Any microservice cannot serve more than one responsibility, at a time.

Modeled around business domain

Microservice never restrict itself from accepting appropriate technology stack or database. The stack or database is most suitable for solving the business purpose.

Isolated Failure

The large application can remain mostly unaffected by the failure of a single module. It is possible that a service can fail at any time. So, it is important to detect failure quickly, if possible, automatically restore failure.



Infrastructure Automation

The infrastructure automation is the process of scripting environments. With the help of scripting environment, we can apply the same configuration to a single node or thousands of nodes. It is also known as configuration management, scripted infrastructures, and system configuration management.

Deploy independently

Microservices are platform agnostic. It means we can design and deploy them independently without affecting the other services.

Advantages of Microservices

- Microservices are self-contained, independent deployment module.
- The cost of scaling is comparatively less than the monolithic architecture.
- Microservices are independently manageable services. It can enable more and more services as the need arises. It minimizes the impact on existing service.
- It is possible to change or upgrade each service individually rather than upgrading in the entire application.
- Microservices allows us to develop an application which is organic (an application which latterly upgrades by adding more functions or modules) in nature.
- It enables event streaming technology to enable easy integration in comparison to heavyweight interprocess communication.
- Microservices follows the single responsibility principle.
- The demanding service can be deployed on multiple servers to enhance performance.
- Less dependency and easy to test.
- Dynamic scaling.
- Faster release cycle.

Disadvantages of Microservices

- Microservices has all the associated complexities of the distributed system.
- There is a higher chance of failure during communication between different services.
- Difficult to manage a large number of services.
- The developer needs to solve the problem, such as network latency and load balancing.
- Complex testing over a distributed environment.

Components of Microservices

There are the following components of microservices:



Spring Cloud Config Server

Spring Cloud Config Server provides the HTTP resource-based API for external configuration in the distributed system. We can enable the Spring Cloud Config Server by using the annotation `@EnableConfigServer`.

Netflix Eureka Naming Server

Netflix Eureka Server is a discovery server. It provides the REST interface to the outside for communicating with it. A microservice after coming up, register itself as a discovery client. The Eureka server also has another software module called Eureka Client. Eureka client interacts with the Eureka server for service discovery. The Eureka client also balances the client requests.

Hystrix Server

Hystrix server acts as a fault-tolerance robust system. It is used to avoid complete failure of an application. It does this by using the Circuit Breaker mechanism. If the application is running without any issue, the circuit remains closed. If there is an error encountered in the application, the Hystrix Server opens the circuit. The Hystrix server stops the further request to calling service. It provides a highly robust system.

Netflix Zuul API Gateway Server

Netflix Zuul Server is a gateway server from where all the client request has passed through. It acts as a unified interface to a client. It also has an inbuilt load balancer to load the balance of all incoming request from the client.

Netflix Ribbon

Netflix Ribbon is the client-side Inter-Process Communication (IPC) library. It provides the client-side balancing algorithm. It uses a Round Robin Load Balancing:

- Load balancing
- Fault tolerance
- Multiple protocols(HTTP, TCP, UDP)
- Caching and Batching

Zipkin Distributed Server

Zipkin is an open-source project in project. That provides a mechanism for sending, receiving, and visualization traces.

One thing you need to be focused on that is port number.

| Application | Port |
|------------------------------|------|
| Spring Cloud Config Server | 8888 |
| Netflix Eureka Naming Server | 8761 |

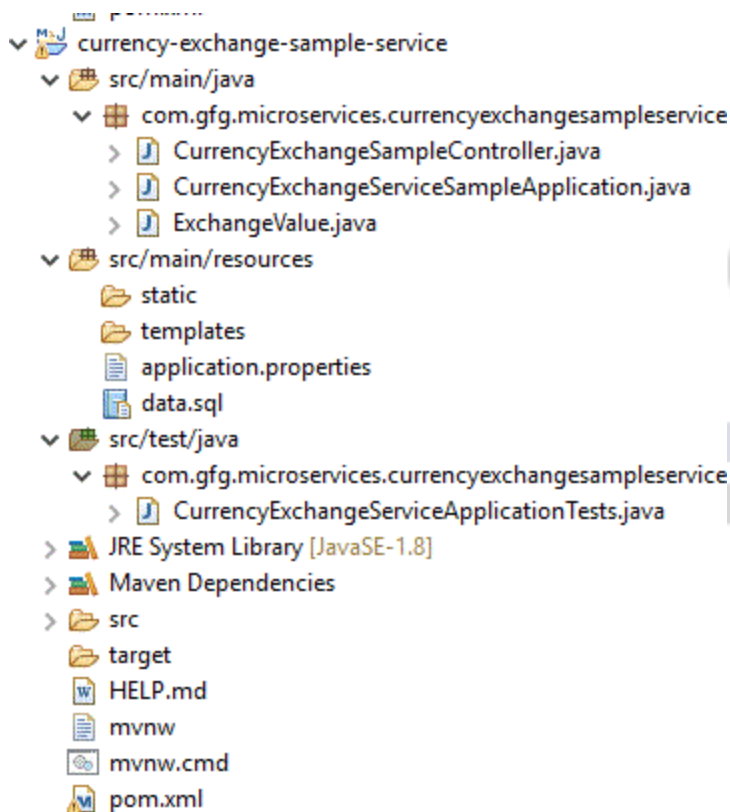


| | |
|-----------------------------------|------|
| Netflix Zuul API gateway Server | 8765 |
| Zipkin distributed Tracing Server | 9411 |

Creating a Simple

MicroserviceProject

Structure



CurrencyExchangeServiceSampleApplication.java



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

@SpringBootApplication
// It is equivalent to using @Configuration,
// @EnableAutoConfiguration and @ComponentScan with their
// default attributes:
public class CurrencyExchangeServiceSampleApplication {

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

CurrencyExchangeSampleController.java

```
@GetMapping("/currency-exchange-sample/fromCurrency/{fromCurrency}/toCurrency/{toCurrency}")
// where {fromCurrency} and {toCurrency} are path variable
// fromCurrency can be USD,EUR,AUD,INR and toCurrency can be the opposite of any fromCurrency
```

```
import java.math.BigDecimal;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.core.env.Environment;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RestController;
```

```
@SpringBootApplication
```

```
@RestController
```

```
// The @RestController annotation in Spring is essentially
```

```
// just a combination of
```

```
// @Controller and @ResponseBody. This annotation was added
```

```
// during Spring 4.0 to remove the redundancy of declaring
```

```
// the @ResponseBody annotation in your controller
```

```
public class CurrencyExchangeSampleController {
```

```
    @Autowired private Environment environment;
```

```
    @GetMapping(
```

```
        "/currency-exchange-sample/fromCurrency/{fromCurrency}/toCurrency/{toCurrency}")
```

```
    // where {fromCurrency} and {toCurrency} are path
```

```
    // variable
```

```
    // fromCurrency can be USD,EUR,AUD,INR and toCurrency
```

```
    // can be the opposite of any fromCurrency
```

```
    public ExchangeValue
```

```
    retrieveExchangeValue(@PathVariable String fromCurrency,
```

```
        @PathVariable String toCurrency)
```

```
    {
```

```
        // Here we need to write all of our business logic
```

```
        BigDecimal conversionMultiple = null;
```

```
        ExchangeValue exchangeValue = new ExchangeValue();
```

```
        if (fromCurrency != null && toCurrency != null) {
```

```
            if (fromCurrency.equalsIgnoreCase("USD")
```

```
                && toCurrency.equalsIgnoreCase("INR")) {
```

```
                conversionMultiple = BigDecimal.valueOf(78);
```

```
            }
```



```

if (fromCurrency.equalsIgnoreCase("INR")
    && toCurrency.equalsIgnoreCase("USD")) {
    conversionMultiple
        = BigDecimal.valueOf(0.013);
}
if (fromCurrency.equalsIgnoreCase("EUR")
    && toCurrency.equalsIgnoreCase("INR")) {
    conversionMultiple = BigDecimal.valueOf(82);
}
if (fromCurrency.equalsIgnoreCase("AUD")
    && toCurrency.equalsIgnoreCase("INR")) {
    conversionMultiple = BigDecimal.valueOf(54);
}
}
// setting the port
exchangeValue = new ExchangeValue(
    1000L, fromCurrency, toCurrency,
    conversionMultiple);
exchangeValue.setPort(Integer.parseInt(
    environment.getProperty("local.server.port")));
return exchangeValue;
}
}
    
```

ExchangeValue.java



```
import java.math.BigDecimal;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.Table;

// @Entity annotation defines that a class can be mapped to
// a table
@Entity

// Representation of the table name
@Table(name = "Exchange_Value")
public class ExchangeValue {
    // The @Id annotation is inherited from
    // javax.persistence.Id, indicating the member field
    // below is the primary key of the current entity
    @Id @Column(name = "id") private Long id;
    @Column(name = "currency_from")
    private String fromCurrency;
    @Column(name = "currency_to") private String toCurrency;
    @Column(name = "conversion_multiple")
    private BigDecimal conversionMultiple;
    @Column(name = "port") private int port;

    public ExchangeValue() {}

    // generating constructor using fields
    public ExchangeValue(Long id, String fromCurrency,
        String toCurrency,
        BigDecimal conversionMultiple)
    {
        super();
        this.id = id;
        this.fromCurrency = fromCurrency;
        this.toCurrency = toCurrency;
        this.conversionMultiple = conversionMultiple;
    }
}
```



```
// generating getters
public int getPort() { return port; }

public void setPort(int port) { this.port = port; }

public Long getId() { return id; }

public String getFrom() { return fromCurrency; }

public String getTo() { return toCurrency; }

public BigDecimal getConversionMultiple()
{
    return conversionMultiple;
}
}
```

application.properties

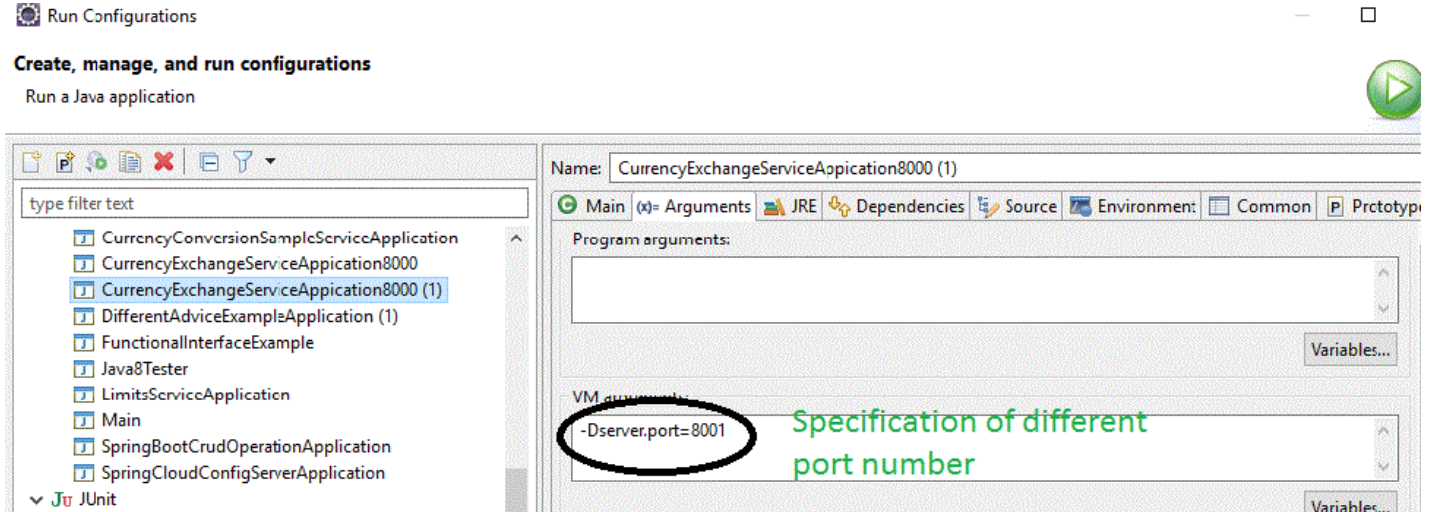
```
spring.application.name=currency-exchange-sample-service
server.port=8000 #Representation of the port number . We can set different port number in run
configuration also
spring.jpa.show-sql=true #To display the SQL
spring.h2.console.enabled=true
spring.datasource.platform=h2 #As we are using h2 datasource
spring.datasource.url=jdbc:h2:mem:gfg
```

data.sql

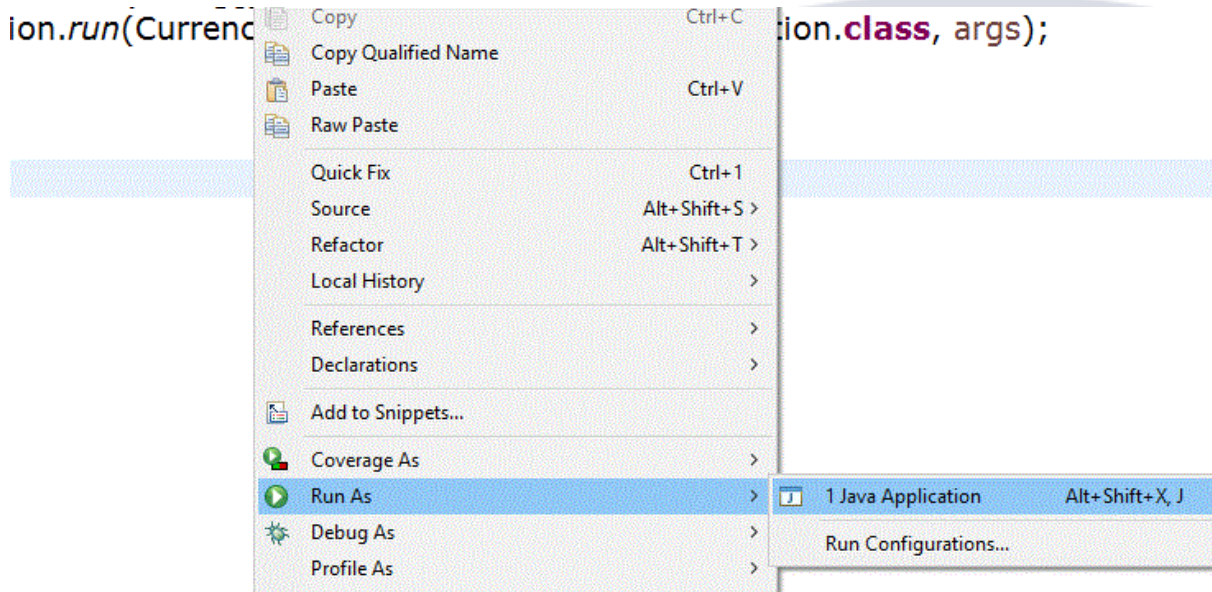
```
insert into exchange_value(id,currency_from,currency_to,conversion_multiple,port)
values(10001,'USD', 'INR' ,65,0);
insert into exchange_value(id,currency_from,currency_to,conversion_multiple,port)
values(10002,'EUR', 'INR' ,82,0);
insert into exchange_value(id,currency_from,currency_to,conversion_multiple,port)
values(10003,'AUD', 'INR' ,53,0);
```

By default, it has been set to run on port 8000. We can create another instance and can make the project run on

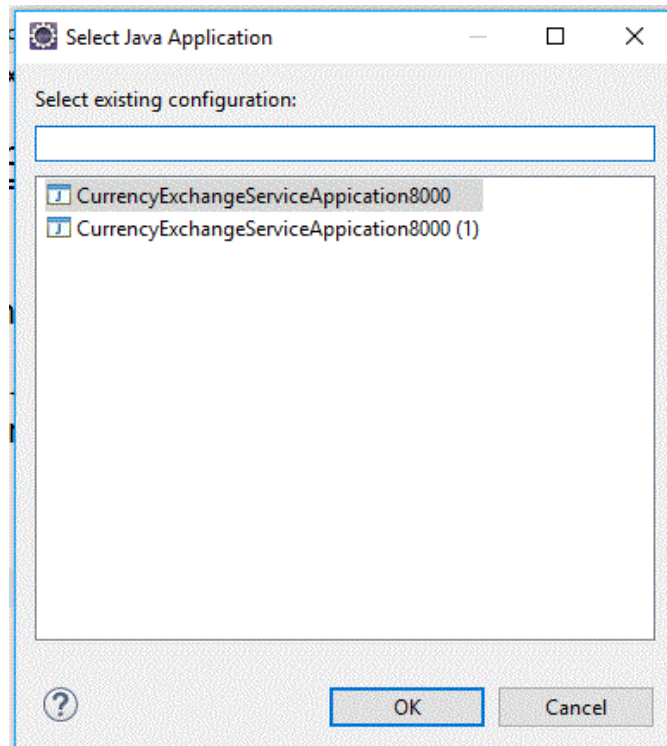
port8001 in the below ways



As this is the spring boot application, it can be normally run as Java application



If we set to run the application on two different ports, we will get the below options



Let us select the first one. On running the application, in the console, we see as


```

2022-05-11 17:04:12.822 INFO 8320 --- [ restartedMain] o.s.cloud.context.scope.GenericScope : BeanFactory id=9d1fb960-e659-3
2022-05-11 17:04:13.119 INFO 8320 --- [ restartedMain] trationDelegate$BeanPostProcessorChecker : Bean 'org.springframework.tran
2022-05-11 17:04:13.172 INFO 8320 --- [ restartedMain] trationDelegate$BeanPostProcessorChecker : Bean 'org.springframework.clou
2022-05-11 17:04:13.991 INFO 8320 --- [ restartedMain] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat initialized with port(s)
2022-05-11 17:04:14.014 INFO 8320 --- [ restartedMain] o.apache.catalina.core.StandardService : Starting service [Tomcat]
2022-05-11 17:04:14.014 INFO 8320 --- [ restartedMain] org.apache.catalina.core.StandardEngine : Starting Servlet engine: [Apache
2022-05-11 17:04:14.234 INFO 8320 --- [ restartedMain] o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring embedded WebAp
2022-05-11 17:04:14.234 INFO 8320 --- [ restartedMain] o.s.web.context.ContextLoader : Root WebApplicationContext: initial
2022-05-11 17:04:14.990 INFO 8320 --- [ restartedMain] com.zaxxer.hikari.HikariDataSource : HikariPool-1 - Starting...
2022-05-11 17:04:15.283 INFO 8320 --- [ restartedMain] com.zaxxer.hikari.HikariDataSource : HikariPool-1 - Start completed.
2022-05-11 17:04:15.392 INFO 8320 --- [ restartedMain] o.hibernate.jpa.internal.util.LogHelper : HHH000204: Processing Persistence
2022-05-11 17:04:15.502 INFO 8320 --- [ restartedMain] org.hibernate.Version : HHH000412: Hibernate Core {5.4.8.Fin
2022-05-11 17:04:15.715 INFO 8320 --- [ restartedMain] o.hibernate.annotations.common.Version : HCANN000001: Hibernate Comm
2022-05-11 17:04:16.000 INFO 8320 --- [ restartedMain] org.hibernate.dialect.Dialect : HHH000490: Using dialect: org.hibernate
Hibernate: drop table exchange_value if exists
Hibernate: create table exchange_value (id bigint not null, conversion_multiple decimal(19,2), currency_from varchar(255), port integer
2022-05-11 17:04:17.343 INFO 8320 --- [ restartedMain] o.h.e.t.j.p.i.JtaPlatformInitiator : HHH000490: Using JtaPlatform: implm
2022-05-11 17:04:17.353 INFO 8320 --- [ restartedMain] j.LocalContainerEntityManagerFactoryBean : Initialized JPA EntityManagerFac
2022-05-11 17:04:17.379 INFO 8320 --- [ restartedMain] o.s.b.d.a.OptionalLiveReloadServer : LiveReload server is running on po
2022-05-11 17:04:17.467 WARN 8320 --- [ restartedMain] JpaBaseConfiguration$JpaWebConfiguration : spring.jpa.open-in-view is en
2022-05-11 17:04:17.658 INFO 8320 --- [ restartedMain] o.s.concurrent.ThreadPoolTaskExecutor : Initializing ExecutorService 'appl
2022-05-11 17:04:18.173 INFO 8320 --- [ restartedMain] o.s.b.a.e.web.EndpointLinksResolver : Exposing 2 endpoint(s) beneath b
2022-05-11 17:04:18.314 INFO 8320 --- [ restartedMain] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s): 80
2022-05-11 17:04:18.369 INFO 8320 --- [ restartedMain] CurrencyExchangeServiceSampleApplication : Started CurrencyExchangeServ
  
```

From the console, we can see that it used default Tomcat and the project is running on port 8080. As we have used 3 insert scripts, automatically table is created and the data is inserted. We can able to do the following

<http://localhost:8000/currency-exchange-sample/fromCurrency/USD/toCurrency/INR>

```

{"id":1000,"conversionMultiple":78,"port":8000,"from":"USD","to":"INR"}
  
```

When this URL is hit, it will be redirected to the controller, and fromCurrency is taken as “USD” and toCurrency is taken as “INR”

Because of this,



```
// Below set of code is executed and hence we are seeing the result like above
if (fromCurrency != null && toCurrency != null) {
    if (fromCurrency.equalsIgnoreCase("USD") && toCurrency.equalsIgnoreCase("INR")) {
        conversionMultiple = BigDecimal.valueOf(78);
    }
}
```

Similarly, we can able to execute the below following URLs

```
http://localhost:8000/currency-exchange-sample/fromCurrency/EUR/toCurrency/INR
```

```
{ "id": 1000, "conversionMultiple": 82, "port": 8000, "from": "EUR", "to": "INR" }
```

```
http://localhost:8000/currency-exchange-sample/fromCurrency/AUD/toCurrency/INR
```

```
{ "id": 1000, "conversionMultiple": 54, "port": 8000, "from": "AUD", "to": "INR" }
```

Hence according to our business needs, we can add business logic to the controller file. Let us see how the above service is getting called in the currency-conversion project