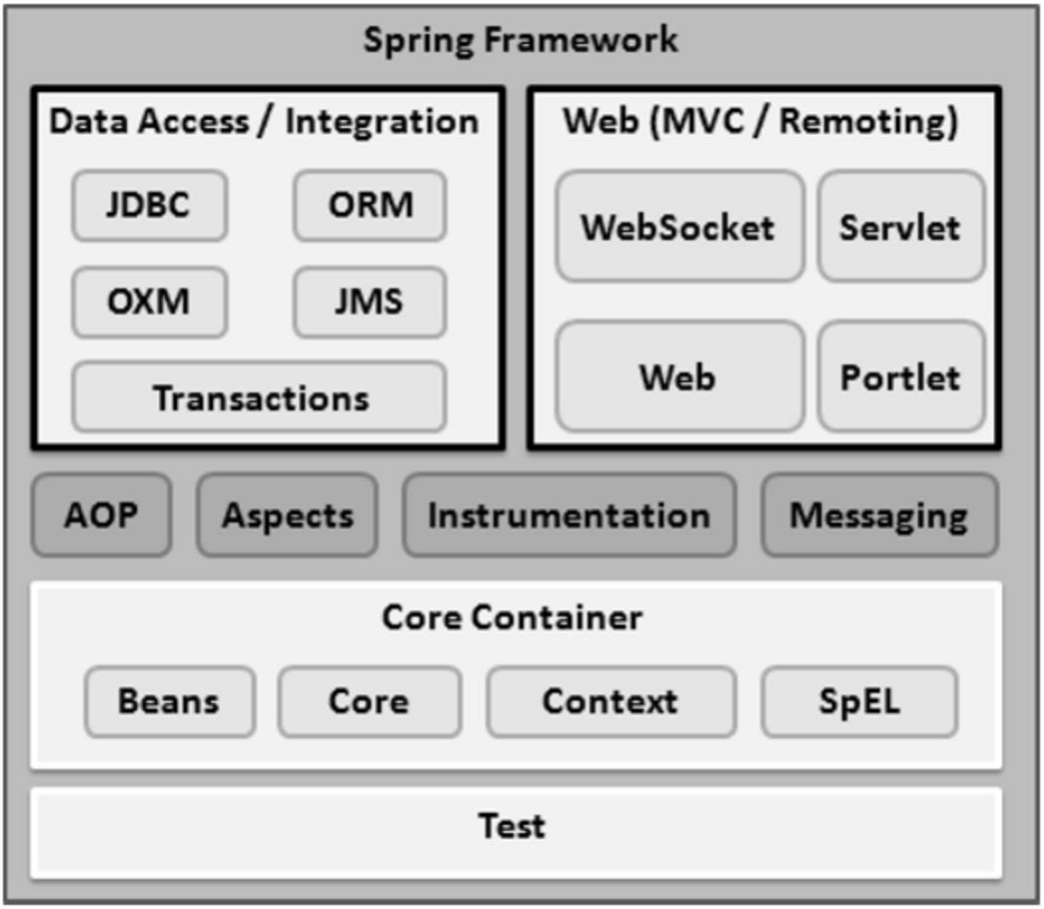
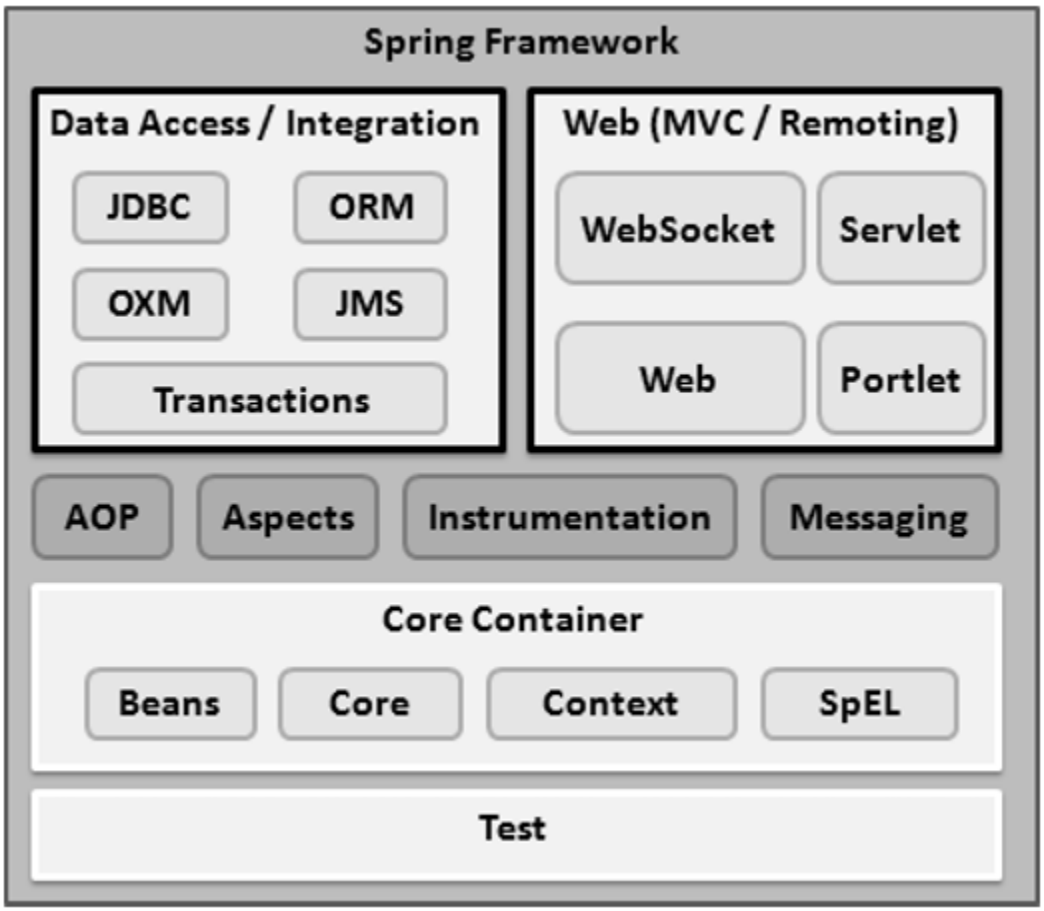
Unit 1 - Introduction to spring framework



Spring Framework - Architecture

Spring could potentially be a one-stop shop for all your enterprise applications. However, Spring is modular, allowing you to pick and choose which modules are applicable to you, without having to bring in the rest. The following section provides details about all the modules available in Spring Framework.

The Spring Framework provides about 20 modules which can be used based on an application requirement.



Core Container

The Core Container consists of the Core, Beans, Context, and Expression Language modules the details of which are as follows −

- The **Core** module provides the fundamental parts of the framework, including the IoC and Dependency Injection features.
- The **Bean** module provides BeanFactory, which is a sophisticated implementation of the

factory pattern.

- The **Context** module builds on the solid base provided by the Core and Beans modules and it is a medium to access any objects defined and configured. The ApplicationContext interface is the focal point of the Context module.
- The **SpEL** module provides a powerful expression language for querying and manipulating an object graph at runtime.

Data Access/Integration

The Data Access/Integration layer consists of the JDBC, ORM, OXM, JMS and Transaction modules whose detail is as follows −

- The **JDBC** module provides a JDBC-abstraction layer that removes the need for tedious JDBC related coding.
- The **ORM** module provides integration layers for popular object-relational mapping APIs, including JPA, JDO, Hibernate, and iBatis.
- The **OXM** module provides an abstraction layer that supports Object/XML mapping implementations for JAXB, Castor, XMLBeans, JiBX and XStream.
- The Java Messaging Service **JMS** module contains features for producing and consuming messages.
- The **Transaction** module supports programmatic and declarative transaction management for classes that implement special interfaces and for all your POJOs.

Web

The Web layer consists of the Web, Web-MVC, Web-Socket, and Web-Portlet modules the details of which are as follows −

- The **Web** module provides basic web-oriented integration features such as multipart file-upload functionality and the initialization of the IoC container using servlet listeners and a web-oriented application context.
- The **Web-MVC** module contains Spring's Model-View-Controller (MVC) implementation for web applications.
- The **Web-Socket** module provides support for WebSocket-based, two-way communication between the client and the server in web applications.
- The **Web-Portlet** module provides the MVC implementation to be used in a portlet environment and mirrors the functionality of Web-Servlet module.

Miscellaneous

There are few other important modules like AOP, Aspects, Instrumentation, Web and Test modules the details of which are as follows −

- The **AOP** module provides an aspect-oriented programming implementation allowing you to define method-interceptors and pointcuts to cleanly decouple code that implements functionality that should be separated.
- The **Aspects** module provides integration with AspectJ, which is again a powerful and mature AOP framework.
- The **Instrumentation** module provides class instrumentation support and class loader implementations to be used in certain application servers.
- The **Messaging** module provides support for STOMP as the WebSocket sub-protocol to use in applications. It also supports an annotation programming model for routing and processing STOMP messages from WebSocket clients.
- The **Test** module supports the testing of Spring components with JUnit or TestNG frameworks.

Useful Video Courses

Mastering Spring Framework

102 Lectures 8 hours

Karthikeya T

More Detail

Spring Boot: A Quick Tutorial Guide

Spring – Understanding Inversion of Control with Example

Last Updated : 18 Feb, 2022

Spring IoC (Inversion of Control) Container is the core of Spring Framework. It creates the objects, configures and assembles their dependencies, manages their entire life cycle. The Container uses Dependency Injection(DI) to manage the components that make up the application. It gets the information about the objects from a configuration file(XML) or Java Code or Java Annotations and Java POJO class. These objects are called Beans. Since the Controlling of Java objects and their lifecycle is not done by the developers, hence the name Inversion Of Control.

There are 2 types of IoC containers:

- BeanFactory
- ApplicationContext

That means if you want to use an IoC container in spring whether we need to use a BeanFactory or ApplicationContext. The BeanFactory is the most basic version of IoC containers, and the ApplicationContext extends the features of BeanFactory. The followings are some of the

- Managing our objects,
- Helping our application to be configurable,
- Managing dependencies

Implementation: So now let's understand what is IoC in Spring with an example. Suppose we have one interface named Sim and it has some abstract methods calling() and data().

Java

```
// Java Program to Illustrate Sim Interface
public interface Sim
{
    void calling();
    void data();
}
```

Now we have created another two classes Airtel and Jio which implement the Sim interface and override the interface methods.

Java

```
// Java Program to Illustrate Airtel Class

// Class
// Implementing Sim interface
public class Airtel implements Sim {

    @Override public void calling()
    {
        System.out.println("Airtel Calling");
    }

    @Override public void data()
```

Java

```
// Java Program to Illustrate Jio Class

// Class
// Implementing Sim interface
public class Jio implements Sim{
    @Override
    public void calling() {
        System.out.println("Jio Calling");
    }

    @Override
    public void data() {
        System.out.println("Jio Data");
    }
}
```

So let's now call these methods inside the main method. So by implementing the Run time polymorphism concept we can do something like this

Java

```
// Java Program to Illustrate Mobile Class

// Class
public class Mobile {

    // Main driver method
    public static void main(String[] args)
    {
```

Further reading:

Wiring in Spring: @Autowired, @Resource and @Inject (/spring-annotations-resource-inject-autowire)

This article will compare and contrast the use of annotations related to dependency injection, namely the @Resource, @Inject, and @Autowired annotations.

Read more (/spring-annotations-resource-inject-autowire) →

@Component vs @Repository and @Service in Spring (/spring-component-repository-service)

Learn about the differences between the @Component, @Repository and @Service annotations and when to use them.

Read more (/spring-component-repository-service) →

2. What Is Inversion of Control?

Inversion of Control is a principle in software engineering which transfers the control of objects or portions of a program to a container or framework. We most often use it in the context of object-oriented programming.

In contrast with traditional programming, in which our custom code makes calls to a library, IoC enables a framework to take control of the flow of a program and make calls to our custom code. To enable this, frameworks use abstractions with additional behavior built in. **If we want to add our own behavior, we need to extend the classes of the framework or plugin our own classes.**

The advantages of this architecture are:

- decoupling the execution of a task from its implementation

- making it easier to switch between different implementations
- greater modularity of a program
- greater ease in testing a program by isolating a component or mocking its dependencies, and allowing components to communicate through contracts

We can achieve Inversion of Control through various mechanisms such as: Strategy design pattern, Service Locator pattern, Factory pattern, and Dependency Injection (DI).

We're going to look at DI next.

3. What Is Dependency Injection?

Dependency injection is a pattern we can use to implement IoC, where the control being inverted is setting an object's dependencies.

Connecting objects with other objects, or “injecting” objects into other objects, is done by an assembler rather than by the objects themselves.

Here's how we would create an object dependency in traditional programming:

```
public class Store {
    private Item item;

    public Store() {
        item = new ItemImpl1();
    }
}
```

In the example above, we need to instantiate an implementation of the *Item* interface within the *Store* class itself.

By using DI, we can rewrite the example without specifying the implementation of the *Item* that we want:

```
public class Store {
    private Item item;
    public Store(Item item) {
        this.item = item;
    }
}
```

In the next sections, we'll look at how we can provide the implementation of *Item* through metadata.

Both IoC and DI are simple concepts, but they have deep implications in the way we structure our systems, so they're well worth understanding fully.

4. The Spring IoC Container

An IoC container is a common characteristic of frameworks that implement IoC.

In the Spring framework, the interface *ApplicationContext* represents the IoC container. The Spring container is responsible for instantiating, configuring and assembling objects known as *beans*, as well as managing their life cycles.

The Spring framework provides several implementations of the *ApplicationContext* interface: *ClassPathXmlApplicationContext* and *FileSystemXmlApplicationContext* for standalone applications, and *WebApplicationContext* for web applications.

In order to assemble beans, the container uses configuration metadata, which can be in the form of XML configuration or annotations.

Here's one way to manually instantiate a container:

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

To set the *item* attribute in the example above, we can use metadata. Then the container will read this metadata and use it to assemble beans at runtime.

Dependency Injection in Spring can be done through constructors, setters or fields.

IoC Container

The IoC container is responsible to instantiate, configure and assemble the objects. The IoC container gets informations from the XML file and works accordingly. The main tasks performed by IoC container are:

- to instantiate the application class
- to configure the object
- to assemble the dependencies between the objects

There are two types of IoC containers. They are:

1. **BeanFactory**
2. **ApplicationContext**

Difference between BeanFactory and the ApplicationContext

The org.springframework.beans.factory.**BeanFactory** and the org.springframework.context.**ApplicationContext** interfaces acts as the IoC container. The ApplicationContext interface is built on top of the BeanFactory interface. It adds some extra functionality than BeanFactory such as simple integration with Spring's AOP, message resource handling (for I18N), event propagation, application layer specific context (e.g. WebApplicationContext) for web application. So it is better to use ApplicationContext than BeanFactory.

Using BeanFactory

The XmlBeanFactory is the implementation class for the BeanFac
to create the instance of XmlBeanFactory class as given below:

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

The constructor of XmlBeanFactory class receives the Resource object so we need to pass the resource object tc create the object of BeanFactory.

Using ApplicationContext

The ClassPathXmlApplicationContext class is the implementation class of ApplicationContext interface. We need tc instantiate the ClassPathXmlApplicationContext class to use the ApplicationContext as given below:

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

The constructor of ClassPathXmlApplicationContext class receives string, so we can pass the name of the xml file tc create the instance of ApplicationContext.

download the example to use ApplicationContext

← Prev Next →

Dependency Injection by Constructor Example

We can inject the dependency by constructor. The **<constructor-arg>** subelement 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:

- Employee.java
- applicationContext.xml
- Test.java

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.javatpoint;

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;}

(int id, String name) {
```

```
    this.id = id;
    this.name = name;
}

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

}
```


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.javatpoint.Employee">
<constructor-arg value="10" type="int"></constructor-arg>
</bean>

</beans>
```

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

```
package com.javatpoint;

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

download this example

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.javatpoint.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 0 10

Output:0 10

You may also pass the string literal as following:

```
....
ss="com.javatpoint.Employee">
```

```
<constructor-arg value="Sonoo"></constructor-arg>
</bean>
....
```

Output:0 Sonoo

You may pass integer literal and string both as following

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

Output:10 Sonoo

download this example (developed using Myeclipse IDE)
download this example (developed using Eclipse IDE)

← Prev Next →

```
1.  Book.java
2.
3.  Book class is a simple class consisting of the book details such title, author,
    publications and its corresponding POJO's.The getBookDetails()method will display the
    book information which is set.
4.
5.  package com.javainterviewpoint;
6.
7.  public class Book
8.  {
9.      private String title;
10.     private String publications;
11.     private String author;
12.
13.     public Book()
14.     {
15.         super();
16.     }
17.
18.     public Book(String title, String publications, String author)
19.     {
20.         super();
21.         this.title = title;
22.         this.publications = publications;
23.         this.author = author;
24.     }
25.
26.     public String getTitle()
27.     {
28.         return title;
29.     }
30.
31.     public void setTitle(String title)
32.     {
33.         this.title = title;
34.     }
35.
36.     public String getPublications()
37.     {
38.         return publications;
39.     }
40.
41.     public void setPublications(String publications)
42.     {
43.         this.publications = publications;
44.     }
45.
46.     public String getAuthor()
47.     {
48.         return author;
49.     }
50.
51.     public void setAuthor(String author)
52.     {
53.         this.author = author;
54.     }
55.
56.     public void getBookDetails()
57.     {
58.         System.out.println("**Published Book Details**");
59.         System.out.println("Book Title        : " + title);
60.         System.out.println("Book Author       : " + author);
61.         System.out.println("Book Publications : " + publications);
62.     }
63. }
64.
65. Library.java
66.
```

```
Library class has the Book class instance as a property and its corresponding getters and setters. The book property will get its value through our configuration file.

package com.javainterviewpoint;

public class Library
{
    private Book book;

    public void setBook(Book book)
    {
        this.book = book;
    }

    public Book getBook()
    {
        return book;
    }
}

SpringConfig.xml

In our configuration file we have defined seperate id for each bean Library and Book classes

<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="library" class="com.javainterviewpoint.Library">
         <property name="book" ref="book"></property>
    </bean>

    <bean id="book" class="com.javainterviewpoint.Book">
           <property name="title" value="Spring XML Configuration"></property>
           <property name="author" value="JavaInterviewPoint"></property>
           <property name="publications" value="JIP Publication"></property>
      </bean>
 </beans>

     We inject primitives to the Book class properties such as title, author, publications.

     <bean id="book" class="com.javainterviewpoint.Book">
              <property name="title" value="Spring XML Configuration"></property>
              <property name="author" value="JavaInterviewPoint"></property>
              <property name="publications" value="JIP"></property>
      </bean>

     We are referencing the Book class bean id to the property book of the Library class

 <property name="book" ref="book"></property>

     The ref passed to the property book should be the bean id of the Book Class. In short

 ref =<<bean id of Book class>>
 Application.java

 package com.javainterviewpoint;

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

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

```
132.          // Read the Spring configuration file [SpringConfig.xml]
133.          ApplicationContext appContext = new
    ClassPathXmlApplicationContext("SpringConfig.xml");
134.          // Get the Library instance
135.          Library library = (Library) appContext.getBean("library");
136.          // Get the Book Details
137.          library.getBook().getBookDetails();
138.      }
139.  }
140.
```

```
1.  Book.java
2.
3.  Book class is a simple class consisting of the book details such title, author,
    publications and its corresponding POJO's.The getBookDetails()method will display the
    book information which is set.
4.
5.  package com.javainterviewpoint;
6.
7.  public class Book
8.  {
9.      private String title;
10.     private String publications;
11.     private String author;
12.
13.     public Book()
14.     {
15.         super();
16.     }
17.
18.     public Book(String title, String publications, String author)
19.     {
20.         super();
21.         this.title = title;
22.         this.publications = publications;
23.         this.author = author;
24.     }
25.
26.     public String getTitle()
27.     {
28.         return title;
29.     }
30.
31.     public void setTitle(String title)
32.     {
33.         this.title = title;
34.     }
35.
36.     public String getPublications()
37.     {
38.         return publications;
39.     }
40.
41.     public void setPublications(String publications)
42.     {
43.         this.publications = publications;
44.     }
45.
46.     public String getAuthor()
47.     {
48.         return author;
49.     }
50.
51.     public void setAuthor(String author)
52.     {
53.         this.author = author;
54.     }
55.
56.     public void getBookDetails()
```

```
    {
        System.out.println("**Published Book Details**");
        System.out.println("Book Title        : " + title);
        System.out.println("Book Author       : " + author);
        System.out.println("Book Publications : " + publications);
    }
}

Library.java

Library class has the Book class instance as a property and its corresponding getters and setters. The book property will get its value through our configuration file.

package com.javainterviewpoint;

public class Library
{
    private Book book;

    public void setBook(Book book)
    {
        this.book = book;
    }

    public Book getBook()
    {
        return book;
    }
}

SpringConfig.xml

In our configuration file we have defined seperate id for each bean Library and Book classes

<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="library" class="com.javainterviewpoint.Library">
         <property name="book" ref="book"></property>
    </bean>

    <bean id="book" class="com.javainterviewpoint.Book">
           <property name="title" value="Spring XML Configuration"></property>
           <property name="author" value="JavaInterviewPoint"></property>
           <property name="publications" value="JIP Publication"></property>
       </bean>
  </beans>

      We inject primitives to the Book class properties such as title, author, publications.

      <bean id="book" class="com.javainterviewpoint.Book">
               <property name="title" value="Spring XML Configuration"></property>
               <property name="author" value="JavaInterviewPoint"></property>
               <property name="publications" value="JIP"></property>
       </bean>

      We are referencing the Book class bean id to the property book of the Library class

  <property name="book" ref="book"></property>

      The ref passed to the property book should be the bean id of the Book Class. In short

  ref =<<bean id of Book class>>
  Application.java
```

```
122.
123.  package com.javainterviewpoint;
124.
125.  import org.springframework.context.ApplicationContext;
126.  import org.springframework.context.support.ClassPathXmlApplicationContext;
127.
128.  public class Application
129.  {
130.      public static void main(String args[])
131.      {
132.          // Read the Spring configuration file [SpringConfig.xml]
133.          ApplicationContext appContext = new
    ClassPathXmlApplicationContext("SpringConfig.xml");
134.          // Get the Library instance
135.          Library library = (Library) appContext.getBean("library");
136.          // Get the Book Details
137.          library.getBook().getBookDetails();
138.      }
139.  }
140.
```

17. Web MVC framework

17.1 Introduction to Spring Web MVC framework

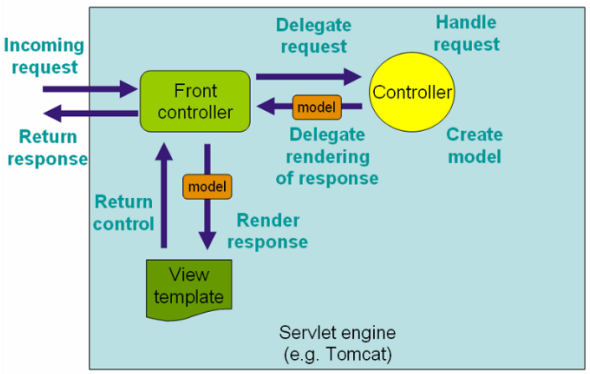
The Spring Web model-view-controller (MVC) framework is designed around a `DispatcherServlet` that dispatches requests to handlers, with configurable handler mappings, view resolution, locale and theme resolution as well as support for uploading files. The default handler is based on the `@Controller` and `@RequestMapping` annotations, offering a wide range of flexible handling methods. With the introduction of Spring 3.0, the `@Controller` mechanism also allows you to create RESTful Web sites and applications, through the `@PathVariable` annotation and other features.

Spring's view resolution is extremely flexible. A `Controller` is typically responsible for preparing a model `Map` with data and selecting a view name but it can also write directly to the response stream and complete the request. View name resolution is highly configurable through file extension or Accept header content type negotiation, through bean names, a properties file, or even a custom `ViewResolver` implementation. The model (the M in MVC) is a `Map` interface, which allows for the complete abstraction of the view technology. You can integrate directly with template based rendering technologies such as JSP, Velocity and Freemarker, or directly generate XML, JSON, Atom, and many other types of content. The model `Map` is simply transformed into an appropriate format, such as JSP request attributes, a Velocity template model.

17.2 The `DispatcherServlet`

Spring's web MVC framework is, like many other web MVC frameworks, request-driven, designed around a central Servlet that dispatches requests to controllers and offers other functionality that facilitates the development of web applications. Spring's `DispatcherServlet` however, does more than just that. It is completely integrated with the Spring IoC container and as such allows you to use every other feature that Spring has.

The request processing workflow of the Spring Web MVC `DispatcherServlet` is illustrated in the following diagram. The pattern-savvy reader will recognize that the `DispatcherServlet` is an expression of the “Front Controller” design pattern (this is a pattern that Spring Web MVC shares with many other leading web frameworks).



The request processing workflow in Spring Web MVC (high level)

The `DispatcherServlet` is an actual `Servlet` (it inherits from the `HttpServlet` base class), and as such is declared in the `web.xml` of your web application. You need to map requests that you want the `DispatcherServlet` to handle, by using a URL mapping in the same `web.xml` file. This is standard Java EE Servlet configuration; the following example shows such a `DispatcherServlet` declaration and mapping:

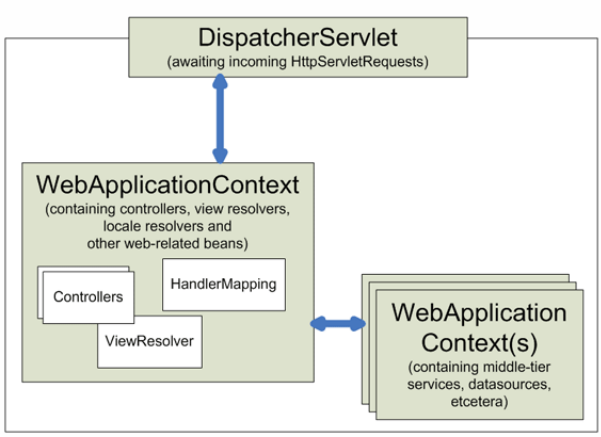
```
1.  <web-app>
2.
3.      <servlet>
4.          <servlet-name>example</servlet-name>
5.          <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
6.          <load-on-startup>1</load-on-startup>
7.      </servlet>
8.
9.      <servlet-mapping>
10.         <servlet-name>example</servlet-name>
11.         <url-pattern>/example/*</url-pattern>
12.     </servlet-mapping>
13.
14. </web-app>
15.
```

In the preceding example, all requests starting with `/example` will be handled by the `DispatcherServlet` instance named `example`.

`WebApplicationInitializer` is an interface provided by Spring MVC that ensures your code-based configuration is detected and automatically used to initialize any Servlet 3 container. An abstract base class implementation of this interace named `AbstractDispatcherServletInitializer` makes it even easier to register the `DispatcherServlet` by simply specifying its servlet mapping. See Code-based Servlet container initialization for more details.

The above is only the first step in setting up Spring Web MVC. You now need to configure the various beans used by the Spring Web MVC framework (over and above the `DispatcherServlet` itself).

As detailed in Section 5.14, “Additional Capabilities of the `ApplicationContext`”, `ApplicationContext` instances in Spring can be scoped. In the Web MVC framework, each `DispatcherServlet` has its own `WebApplicationContext`, which inherits all the beans already defined in the root `WebApplicationContext`. These inherited beans can be overridden in the servlet-specific scope, and you can define new scope-specific beans local to a given Servlet instance.



Context hierarchy in Spring Web MVC

Upon initialization of a `DispatcherServlet`, Spring MVC looks for a file named *`[servlet-name]-servlet.xml`* in the `WEB-INF` directory of your web application and creates the beans defined there, overriding the definitions of any beans defined with the same name in the global scope.

Consider the following `DispatcherServlet` Servlet configuration (in the `web.xml` file):

```
1.  <web-app>
2.
3.      <servlet>
4.          <servlet-name>golfing</servlet-name>
5.          <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
6.          <load-on-startup>1</load-on-startup>
7.      </servlet>
8.
9.      <servlet-mapping>
10.         <servlet-name>golfing</servlet-name>
11.         <url-pattern>/golfing/*</url-pattern>
12.     </servlet-mapping>
13.
14. </web-app>
15.
```

With the above Servlet configuration in place, you will need to have a file called `/WEB-INF/`**golfing**`-servlet.xml` in your application; this file will contain all of your Spring Web MVC-specific components (beans). You can change the exact location of this configuration file through a Servlet initialization parameter (see below for details).

The `WebApplicationContext` is an extension of the plain `ApplicationContext` that has some extra features necessary for web applications. It differs from a normal `ApplicationContext` in that it is capable of resolving themes (see Section 17.9, “Using themes”), and that it knows which Servlet it is associated with (by having a link to the `ServletContext`). The `WebApplicationContext` is bound in the `ServletContext`, and by using static methods on the `RequestContextUtils` class you can always look up the `WebApplicationContext` if you need access to it.

17.2.1 Special Bean Types In the `WebApplicationContext`

The Spring `DispatcherServlet` uses special beans to process requests and render the appropriate views. These beans are part of Spring MVC. You can choose which special beans to use by simply configuring one or more of them in the `WebApplicationContext`. However, you don't need to do that initially since Spring MVC maintains a list of default beans to use if you don't configure any. More on that in the next section. First see the table below listing the special bean types the `DispatcherServlet` relies on.

Table 17.1. Special bean types in the `WebApplicationContext`

Bean type	Explanation
HandlerMapping	Maps incoming requests to handlers and a list of pre- and post-processors (handler interceptors) based on some criteria the details of which vary by `HandlerMapping` implementation. The most popular implementation supports annotated controllers but other implementations exists as well.
HandlerAdapter	Helps the `DispatcherServlet` to invoke a handler mapped to a request regardless of the handler is actually invoked. For example, invoking an annotated controller requires resolving various annotations. Thus the main purpose of a `HandlerAdapter` is to shield the `DispatcherServlet` from such details.
HandlerExceptionResolver	Maps exceptions to views also allowing for more complex exception handling code.
ViewResolver	Resolves logical String-based view names to actual `View` types.
LocaleResolver	Resolves the locale a client is using, in order to be able to offer internationalized views
ThemeResolver	Resolves themes your web application can use, for example, to offer personalized layouts
MultipartResolver	Parses multi-part requests for example to support processing file uploads from HTML forms.
FlashMapManager	Stores and retrieves the "input" and the "output" `FlashMap` that can be used to pass attributes from one request to another, usually across a redirect.

17.2.2 Default DispatcherServlet Configuration

As mentioned in the previous section for each special bean the `DispatcherServlet` maintains a list of implementations to use by default. This information is kept in the file `DispatcherServlet.properties` in the package `org.springframework.web.servlet`.

All special beans have some reasonable defaults of their own. Sooner or later though you'll need to customize one or more of the properties these beans provide. For example it's quite common to configure an `InternalResourceViewResolver` settings its `prefix` property to the parent location of view files.

Regardless of the details, the important concept to understand here is that once you configure a special bean such as an `InternalResourceViewResolver` in your `WebApplicationContext`, you effectively override the list of default implementations that would have been used otherwise for that special bean type. For example if you configure an

`InternalResourceViewResolver`, the default list of `ViewResolver` implementations is ignored.

In Section 17.15, "Configuring Spring MVC" you'll learn about other options for configuring Spring MVC including MVC Java config and the MVC XML namespace both of which provide a simple starting point and assume little knowledge of how Spring MVC works. Regardless of how you choose to configure your application, the concepts explained in this section are fundamental should be of help to you.

17.2.3 DispatcherServlet Processing Sequence

After you set up a `DispatcherServlet`, and a request comes in for that specific `DispatcherServlet`, the `DispatcherServlet` starts processing the request as follows:

1. The `WebApplicationContext` is searched for and bound in the request as an attribute that the controller and other elements in the process can use. It is bound by default under the key `DispatcherServlet.WEB_APPLICATION_CONTEXT_ATTRIBUTE`.
2. The locale resolver is bound to the request to enable elements in the process to resolve the locale to use when processing the request (rendering the view, preparing data, and so on). If you do not need locale resolving, you do not need it.
3. The theme resolver is bound to the request to let elements such as views determine which theme to use. If you do not use themes, you can ignore it.
4. If you specify a multipart file resolver, the request is inspected for multiparts; if multiparts are found, the request is wrapped in a `MultipartHttpServletRequest` for further processing by other elements in the process. See Section 17.10, "Spring's multipart (file upload) support" for further information about multipart handling.
5. An appropriate handler is searched for. If a handler is found, the execution chain associated with the handler (preprocessors, postprocessors, and controllers) is executed in order to prepare a model or rendering.
6. If a model is returned, the view is rendered. If no model is returned, (may be due to a preprocessor or postprocessor intercepting the request, perhaps for security reasons), no view is rendered, because the request could already have been fulfilled.

Handler exception resolvers that are declared in the `WebApplicationContext` pick up exceptions that are thrown during processing of the request. Using these exception resolvers allows you to define custom behaviors to address exceptions.

The Spring `DispatcherServlet` also supports the return of the *last-modification-date*, as specified by the Servlet API. The process of determining the last modification date for a specific request is straightforward: the `DispatcherServlet` looks up an appropriate handler mapping and tests whether the handler that is found implements the *`LastModified`* interface. If so, the value of the `long getLastModified(request)` method of the `LastModified` interface is returned to the client.

You can customize individual `DispatcherServlet` instances by adding Servlet initialization parameters (`init-param` elements) to the Servlet declaration in the `web.xml` file. See the following table for the list of supported parameters.

Table 17.2. `DispatcherServlet` initialization parameters

Parameter	Explanation
`contextClass`	Class that implements `WebApplicationContext`, which instantiates the context used by this Servlet. By default, the `XmlWebApplicationContext` is used.
`contextConfigLocation`	String that is passed to the context instance (specified by `contextClass`) to indicate where context(s) can be found. The string consists potentially of multiple strings (using a comma as a delimiter) to support multiple contexts. In case of multiple context locations with beans that are defined twice, the latest location takes precedence.
`namespace`	Namespace of the `WebApplicationContext`. Defaults to `[servlet-name]-servlet`.

17.3 Implementing Controllers

Controllers provide access to the application behavior that you typically define through a service interface. Controllers interpret user input and transform it into a model that is represented to the user by the view. Spring implements a controller in a very abstract way, which enables you to create a wide variety of controllers.

Spring 2.5 introduced an annotation-based programming model for MVC controllers that uses annotations such as `@RequestMapping`, `@RequestParam`, `@ModelAttribute`, and so on. This annotation support is available for both Servlet MVC and Portlet MVC. Controllers implemented in this style do not have to extend specific base classes or implement specific interfaces. Furthermore, they do not usually have direct dependencies on Servlet or Portlet APIs, although you can easily configure access to Servlet or Portlet facilities.

Available in the samples repository, a number of web applications leverage the annotation support described in this section including *MvcShowcase*, *MvcAjax*, *MvcBasic*, *PetClinic*, *PetCare*, and others.

```
1.  @Controller
2.  public class HelloWorldController {
3.
4.      @RequestMapping("/helloWorld")
5.      public String helloWorld(Model model) {
6.          model.addAttribute("message", "Hello World!");
7.          return "helloWorld";
8.      }
9.  }
10.
```

As you can see, the `@Controller` and `@RequestMapping` annotations allow flexible method names and signatures. In this particular example the method accepts a `Model` and returns a view name as a `String`, but various other method parameters and return values can be used as explained later in this section. `@Controller` and `@RequestMapping` and a number of other annotations form the basis for the Spring MVC implementation. This section documents these annotations and how they are most commonly used in a Servlet environment.

17.3.1 Defining a controller with `@Controller`

The `@Controller` annotation indicates that a particular class serves the role of a *controller*. Spring does not require you to extend any controller base class or reference the Servlet API. However, you can still reference Servlet-specific features if you need to.

The `@Controller` annotation acts as a stereotype for the annotated class, indicating its role. The dispatcher scans such annotated classes for mapped methods and detects `@RequestMapping` annotations (see the next section).

You can define annotated controller beans explicitly, using a standard Spring bean definition in the dispatcher's context. However, the `@Controller` stereotype also allows for autodetection, aligned with Spring general support for detecting component classes in the classpath and auto-registering bean definitions for them.

To enable autodetection of such annotated controllers, you add component scanning to your configuration. Use the *spring-context* schema as shown in the following XML snippet:

```
1.  <?xml version="1.0" encoding="UTF-8"?>
2.  <beans xmlns="http://www.springframework.org/schema/beans"
3.      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4.      xmlns:p="http://www.springframework.org/schema/p"
5.      xmlns:context="http://www.springframework.org/schema/context"
6.      xsi:schemaLocation="
7.          http://www.springframework.org/schema/beans
8.          http://www.springframework.org/schema/beans/spring-beans.xsd
9.          http://www.springframework.org/schema/context
10.         http://www.springframework.org/schema/context/spring-context.xsd">
11.
12.     <context:component-scan base-package="org.springframework.samples.petclinic.web"/>
13.
14.     <!-- ... -->
15.
16. </beans>
17.
```

17.3.2 Mapping Requests With `@RequestMapping`

You use the `@RequestMapping` annotation to map URLs such as `/appointments` onto an entire class or a particular handler method. Typically the class-level annotation maps a specific request path (or path pattern) onto a form controller, with additional method-level annotations narrowing the primary mapping for a specific HTTP method request method ("GET", "POST", etc.) or an HTTP request parameter condition.

The following example from the *Petcare* sample shows a controller in a Spring MVC application that uses this annotation:

```
1.  @Controller
2.  @RequestMapping("/appointments")
3.  public class AppointmentsController {
4.
5.      private final AppointmentBook appointmentBook;
6.
7.      @Autowired
8.      public AppointmentsController(AppointmentBook appointmentBook) {
9.          this.appointmentBook = appointmentBook;
```

```
10.     }
11.
12.     @RequestMapping(method = RequestMethod.GET)
13.     public Map<String, Appointment> get() {
14.         return appointmentBook.getAppointmentsForToday();
15.     }
16.
17.     @RequestMapping(value="/{day}", method = RequestMethod.GET)
18.     public Map<String, Appointment> getForDay(@PathVariable
    @DateTimeFormat(iso=ISO.DATE) Date day, Model model) {
19.         return appointmentBook.getAppointmentsForDay(day);
20.     }
21.
22.     @RequestMapping(value="/new", method = RequestMethod.GET)
23.     public AppointmentForm getNewForm() {
24.         return new AppointmentForm();
25.     }
26.
27.     @RequestMapping(method = RequestMethod.POST)
28.     public String add(@Valid AppointmentForm appointment, BindingResult result) {
29.         if (result.hasErrors()) {
30.             return "appointments/new";
31.         }
32.         appointmentBook.addAppointment(appointment);
33.         return "redirect:/appointments";
34.     }
35. }
36.
```

In the example, the `@RequestMapping` is used in a number of places. The first usage is on the type (class) level, which indicates that all handling methods on this controller are relative to the `/appointments` path. The `get()` method has a further `@RequestMapping` refinement: it only accepts GET requests, meaning that an HTTP GET for `/appointments` invokes this method. The `post()` has a similar refinement, and the `getNewForm()` combines the definition of HTTP method and path into one, so that GET requests for `appointments/new` are handled by that method.

The `getForDay()` method shows another usage of `@RequestMapping`: URI templates. (See the next section).

A `@RequestMapping` on the class level is not required. Without it, all paths are simply absolute, and not relative. The following example from the *PetClinic* sample application shows a multi-action controller using `@RequestMapping`:

```
1.  @Controller
2.  public class ClinicController {
3.
4.      private final Clinic clinic;
5.
6.      @Autowired
7.      public ClinicController(Clinic clinic) {
8.          this.clinic = clinic;
9.      }
10.
11.     @RequestMapping("/")
12.     public void welcomeHandler() {
13.     }
14.
15.     @RequestMapping("/vets")
16.     public ModelMap vetsHandler() {
17.         return new ModelMap(this.clinic.getVets());
18.     }}
```

Request Parameters and Header Values

You can narrow request matching through request parameter conditions such as `"myParam"`, `"!myParam"`, or `"myParam=myValue"`. The first two test for request parameter presence/absence and the third for a specific parameter value. Here is an example with a request parameter value condition:

```
@Controller
@RequestMapping("/owners/{ownerId}")
public class RelativePathUriTemplateController {

  @RequestMapping(value = "/pets/{petId}", method = RequestMethod.GET,
params="myParam=myValue")
  public void findPet(@PathVariable String ownerId, @PathVariable String petId, Model
model) {
    // implementation omitted
  }
}

```

The same can be done to test for request header presence/absence or to match based on a specific request header value:

```
@Controller
@RequestMapping("/owners/{ownerId}"
)
public class
RelativePathUriTemplateController {

@RequestMapping(value = "/pets",
method = RequestMethod.GET,
headers="myHeader=myValue")
  public void findPet(@PathVariable
String ownerId, @PathVariable
String petId, Model model) {
    // implementation omitted
  }
}

```

Although you can match to *Content-Type* and *Accept* header values using media type wild cards (for example *"content-type=text/*"* will match to *"text/plain"* and *"text/html"*), it is recommended to use the *consumes* and *produces* conditions respectively instead. They are intended specifically for that purpose.

17.3.3 Defining `@RequestMapping` handler methods

An `@RequestMapping` handler method can have a very flexible signatures. The supported method arguments and return values are described in the following section. Most arguments can be used in arbitrary order with the only exception of `BindingResult` arguments. This is described in the next section.

Spring 3.1 introduced a new set of support classes for `@RequestMapping` methods called `RequestMappingHandlerMapping` and `RequestMappingHandlerAdapter` respectively. They are recommended for use and even required to take advantage of new features in Spring MVC 3.1 and going forward. The new support classes are enabled by default from the MVC namespace and with use of the MVC Java config but must be configured explicitly if using neither.

Supported method argument types

Example 17.1. Invalid ordering of BindingResult and @ModelAttribute

```
@RequestMapping(method = RequestMethod.POST)
public String processSubmit(@ModelAttribute("pet") Pet pet,
    Model model, BindingResult result) { … }
```

Note, that there is a `Model` parameter in between `Pet` and `BindingResult`. To get this working you have to reorder the parameters as follows:

```
@RequestMapping(method = RequestMethod.POST)
public String processSubmit(@ModelAttribute("pet") Pet pet,
    BindingResult result, Model model) { … }
```

Supported method return types

The following are the supported return types:

- A `ModelAndView` object, with the model implicitly enriched with command objects and the results of `@ModelAttribute` annotated reference data accessor methods.
- A `Model` object, with the view name implicitly determined through a `RequestToViewNameTranslator` and the model implicitly enriched with command objects and the results of `@ModelAttribute` annotated reference data accessor methods.
- A `Map` object for exposing a model, with the view name implicitly determined through a `RequestToViewNameTranslator` and the model implicitly enriched with command objects and the results of `@ModelAttribute` annotated reference data accessor methods.
- A `View` object, with the model implicitly determined through command objects and `@ModelAttribute` annotated reference data accessor methods. The handler method may also programmatically enrich the model by declaring a `Model` argument (see above).
- A `String` value that is interpreted as the logical view name, with the model implicitly determined through command objects and `@ModelAttribute` annotated reference data accessor methods. The handler method may also programmatically enrich the model by declaring a `Model` argument (see above).
- `void` if the method handles the response itself (by writing the response content directly, declaring an argument of type `ServletResponse` / `HttpServletResponse` for that purpose) or if the view name is supposed to be implicitly determined through a `RequestToViewNameTranslator` (not declaring a response argument in the handler method signature).
- If the method is annotated with `@ResponseBody`, the return type is written to the response HTTP body. The return value will be converted to the declared method argument type using `HttpMessageConverter`s. See the section called “Mapping the response body with the `@ResponseBody` annotation”.
- A `HttpEntity<?>` or `ResponseEntity<?>` object to provide access to the Servlet response HTTP headers and contents. The entity body will be converted to the response stream using `HttpMessageConverter`s. See the section called “Using `HttpEntity<?>`”.
- A `Callable<?>` can be returned when the application wants to produce the return value asynchronously in a thread managed by Spring MVC.

- A `DeferredResult<?>` can be returned when the application wants to produce the return value from a thread of its own choosing.
- Any other return type is considered to be a single model attribute to be exposed to the view, using the attribute name specified through `@ModelAttribute` at the method level (or the default attribute name based on the return type class name). The model is implicitly enriched with command objects and the results of `@ModelAttribute` annotated reference data accessor methods.

Binding request parameters to method parameters with `@RequestParam`

Use the `@RequestParam` annotation to bind request parameters to a method parameter in your controller.

The following code snippet shows the usage:

```
1.  @Controller
2.  @RequestMapping("/pets")
3.  @SessionAttributes("pet")
4.  public class EditPetForm {
5.
6.      // ...
7.
8.      @RequestMapping(method = RequestMethod.GET)
9.      public String setupForm(@RequestParam("petId") int petId, ModelMap model) {
10.         Pet pet = this.clinic.loadPet(petId);
11.         model.addAttribute("pet", pet);
12.         return "petForm";
13.     }
14.
15.     // ...
16.
```

Parameters using this annotation are required by default, but you can specify that a parameter is optional by setting `@RequestParam`'s `required` attribute to `false` (e.g., `@RequestParam(value="id", required=false)`).

Type conversion is applied automatically if the target method parameter type is not `String`. See the section called “Method Parameters And Type Conversion”.

Mapping the request body with the @RequestBody annotation

The `@RequestBody` method parameter annotation indicates that a method parameter should be bound to the value of the HTTP request body. For example:

```
@RequestMapping(value = "/something", method = RequestMethod.PUT)
public void handle(@RequestBody String body, Writer writer) throws
IOException {
  writer.write(body);
}
```

You convert the request body to the method argument by using an `HttpMessageConverter`. `HttpMessageConverter` is responsible for converting from the HTTP request message to an object and converting from an object to the HTTP response body. The `RequestMappingHandlerAdapter` supports the `@RequestBody` annotation with the following default `HttpMessageConverters`:

- `ByteArrayHttpMessageConverter` converts byte arrays.
- `StringHttpMessageConverter` converts strings.
- `FormHttpMessageConverter` converts form data to/from a MultiValueMap<String, String>.
- `SourceHttpMessageConverter` converts to/from a javax.xml.transform.Source.

For more information on these converters, see Message Converters. Also note that if using the MVC namespace or the MVC Java config, a wider range of message converters are registered by default. See Enabling the MVC Java Config or the MVC XML Namespace for more information.

If you intend to read and write XML, you will need to configure the `MarshallingHttpMessageConverter` with a specific `Marshaller` and an `Unmarshaller` implementation from the `org.springframework.oxm` package. The example below shows how to do that directly in your configuration but if your application is configured through the MVC namespace or the MVC Java config see Enabling the MVC Java Config or the MVC XML Namespace instead.

```
<bean
class="org.springframework.web.servlet.mvc.method.annotation.RequestMapping
HandlerAdapter">
    <property name="messageConverters">
      <util:list id="beanList">
        <ref bean="stringHttpMessageConverter"/>
        <ref bean="marshallingHttpMessageConverter"/>
      </util:list>
    </property>
</bean>

<bean id="stringHttpMessageConverter"

class="org.springframework.http.converter.StringHttpMessageConverter"/>

<bean id="marshallingHttpMessageConverter"

class="org.springframework.http.converter.xml.MarshallingHttpMessageConvert
er">
  <property name="marshaller" ref="castorMarshaller" />
  <property name="unmarshaller" ref="castorMarshaller" />
</bean>

<bean id="castorMarshaller"
class="org.springframework.oxm.castor.CastorMarshaller"/>
```

An `@RequestBody` method parameter can be annotated with `@Valid`, in which case it will be validated using the configured `Validator` instance. When using the MVC namespace or the MVC Java config, a JSR-303 validator is configured automatically assuming a JSR-303 implementation is available on the classpath.

Just like with `@ModelAttribute` parameters, an `Errors` argument can be used to examine the errors. If such an argument is not declared, a `MethodArgumentNotValidException` will be raised. The exception is handled in the `DefaultHandlerExceptionResolver`, which sends a `400` error back to the client.

Also see Enabling the MVC Java Config or the MVC XML Namespace for information on configuring message converters and a validator through the MVC namespace or the MVC Java config.

Mapping the response body with the `@ResponseBody` annotation

The `@ResponseBody` annotation is similar to `@RequestBody`. This annotation can be put on a method and indicates that the return type should be written straight to the HTTP response body (and not placed in a Model, or interpreted as a view name). For example:

```
@RequestMapping(value = "/something", method = RequestMethod.PUT)
@ResponseBody
public String helloWorld()  {
  return "Hello World";
}
```

The above example will result in the text `Hello World` being written to the HTTP response stream.

As with `@RequestBody`, Spring converts the returned object to a response body by using an `HttpMessageConverter`. For more information on these converters, see the previous section and Message Converters.

Using `HttpEntity<?>`

The `HttpEntity` is similar to `@RequestBody` and `@ResponseBody`. Besides getting access to the request and response body, `HttpEntity` (and the response-specific subclass `ResponseEntity`) also allows access to the request and response headers, like so:

```
@RequestMapping("/something")
public ResponseEntity<String> handle(HttpEntity<byte[]> requestEntity)
throws UnsupportedEncodingException {
  String requestHeader =
requestEntity.getHeaders().getFirst("MyRequestHeader"));
  byte[] requestBody = requestEntity.getBody();
  // do something with request header and body

  HttpHeaders responseHeaders = new HttpHeaders();
  responseHeaders.set("MyResponseHeader", "MyValue");
  return new ResponseEntity<String>("Hello World", responseHeaders,
HttpStatus.CREATED);
}
```

The above example gets the value of the `MyRequestHeader` request header, and reads the body as a byte array. It adds the `MyResponseHeader` to the response, writes `Hello World` to the response stream, and sets the response status code to 201 (Created).

As with `@RequestBody` and `@ResponseBody`, Spring uses `HttpMessageConverter` to convert from and to the request and response streams. For more information on these converters, see the previous section and Message Converters.

Using `@ModelAttribute` on a method

The `@ModelAttribute` annotation can be used on methods or on method arguments. This section explains its usage on methods while the next section explains its usage on method arguments.

An `@ModelAttribute` on a method indicates the purpose of that method is to add one or more model attributes. Such methods support the same argument types as `@RequestMapping` methods but cannot be mapped directly to requests. Instead `@ModelAttribute` methods in a controller are invoked before `@RequestMapping` methods, within the same controller. A couple of examples:

```
// Add one attribute
// The return value of the method is added to the model under the name
"account"
// You can customize the name via @ModelAttribute("myAccount")

@ModelAttribute
public Account addAccount(@RequestParam String number) {
    return accountManager.findAccount(number);
}

// Add multiple attributes

@ModelAttribute
public void populateModel(@RequestParam String number, Model model) {
    model.addAttribute(accountManager.findAccount(number));
    // add more ...
}
```

`@ModelAttribute` methods are used to populate the model with commonly needed attributes for example to fill a drop-down with states or with pet types, or to retrieve a command object like Account in order to use it to represent the data on an HTML form. The latter case is further discussed in the next section.

Note the two styles of `@ModelAttribute` methods. In the first, the method adds an attribute implicitly by returning it. In the second, the method accepts a `Model` and adds any number of model attributes to it. You can choose between the two styles depending on your needs.

A controller can have any number of `@ModelAttribute` methods. All such methods are invoked before `@RequestMapping` methods of the same controller.

`@ModelAttribute` methods can also be defined in an `@ControllerAdvice`-annotated class and such methods apply to all controllers. The `@ControllerAdvice` annotation is a component annotation allowing implementation classes to be autodetected through classpath scanning.

What happens when a model attribute name is not explicitly specified? In such cases a default name is assigned to the model attribute based on its type. For example if the method returns an object of type `Account`, the default name used is "account". You can change that through the value of the `@ModelAttribute` annotation. If adding attributes directly to the `Model`, use the appropriate overloaded `addAttribute(..)` method - i.e., with or without an attribute name.

The `@ModelAttribute` annotation can be used on `@RequestMapping` methods as well. In that case the return value of the `@RequestMapping` method is interpreted as a model attribute rather than as a view name. The view name is derived from view name conventions instead much like for methods returning void — see Section 17.12.3, “The View - `RequestToViewNameTranslator`”.

Using `@ModelAttribute` on a method argument

As explained in the previous section `@ModelAttribute` can be used on methods or on method arguments. This section explains its usage on method arguments.

An `@ModelAttribute` on a method argument indicates the argument should be retrieved from the model. If not present in the model, the argument should be instantiated first and then added to the model. Once present in the model, the argument's fields should be populated from all request parameters that have matching names. This is known as data binding in Spring MVC, a very useful mechanism that saves you from having to parse each form field individually.

```
@RequestMapping(value="/owners/{ownerId}/pets/{petId}/edit", method =
RequestMethod.POST)
public String processSubmit(@ModelAttribute Pet pet) {

}
```

Spring MVC Validation

The Spring MVC Validation is used to restrict the input provided by the user. To validate the user's input, the Spring 4 or higher version supports and use Bean Validation API. It can validate both server-side as well as client-side applications.

Bean Validation API

The Bean Validation API is a Java specification which is used to apply constraints on object model via annotations. Here, we can validate a length, number, regular expression, etc. Apart from that, we can also provide custom validations.

As Bean Validation API is just a specification, it requires an implementation. So, for that, it uses Hibernate Validator. The Hibernate Validator is a fully compliant JSR-303/309 implementation that allows to express and validate application constraints.

Validation Annotations

Let's see some frequently used validation annotations.

Annotation	Description
@NotNull	It determines that the value can't be null.
@Min	It determines that the number must be equal or greater than the specified value.
@Max	It determines that the number must be equal or less than the specified value.
@Size	It determines that the size must be equal to the specified value.
@Pattern	It determines that the sequence follows the specified regular expression.

Spring MVC Validation Example

In this example, we create a simple form that contains the input fields. Here, (*) me
field. Otherwise, the form generates an error.

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/org.hibernate.validator/hibernate-validator -->
<dependency>
    <groupId>org.hibernate.validator</groupId>
    <artifactId>hibernate-validator</artifactId>
    <version>6.0.13.Final</version>
</dependency>
```

2. Create the bean class

Employee.java

```
package com.javatpoint;
import javax.validation.constraints.Size;

public class Employee {

    private String name;
    @Size(min=1,message="required")
    private String pass;
```

```
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public String getPass() {
        return pass;
    }
    public void setPass(String pass) {
        this.pass = pass;
    }
}
```

3. Create the controller class

In controller class:

- The **@Valid** annotation applies validation rules on the provided object.
- The **BindingResult** interface contains the result of validation.

```
package com.javatpoint;

import javax.validation.Valid;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.validation.BindingResult;
import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.web.bind.annotation.RequestMapping;

@Controller
public class EmployeeController {

    @RequestMapping("/hello")
    public String display(Model m)
    {
        m.addAttribute("emp", new Employee());
        return "viewpage";
    }
    @RequestMapping("/helloagain")
    public String submitForm( @Valid @ModelAttribute("emp") Employee e, Bindin
    {
        if(br.hasErrors())
        {
            return "viewpage";
```

```
    }
    else
    {
    return "final";
    }
  }
}
```

4. 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/xm
/javaee xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/w
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>
```

5. 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">
   <!-- Provide support for component scanning -->
```

```
    <context:component-scan base-package="com.javatpoint" />
    <!--Provide support for conversion, formatting and validation -->
    <mvc:annotation-driven/>
    <!-- Define Spring MVC view resolver -->
     <bean id="viewResolver" class="org.springframework.web.servlet.view.InternalResourceViewResolver">
        <property name="prefix" value="/WEB-INF/jsp/"></property>
        <property name="suffix" value=".jsp"></property>
     </bean>
</beans>
```

6. Create the requested page

index.jsp

```
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form" %>
<html>
<body>
<a href="hello">Click here...</a>
</body>
</html>
```

7. Create the other view components

viewpage.jsp

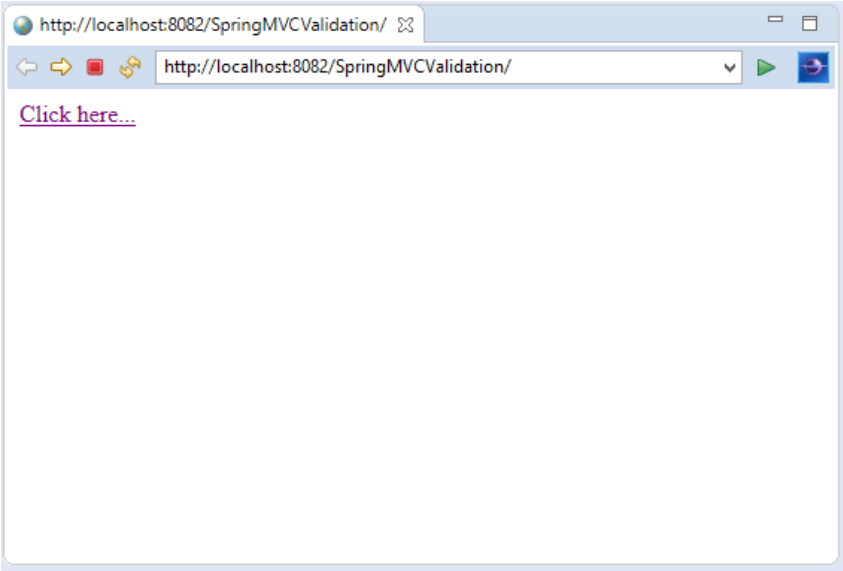
```
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form" %>
<html>
<head>
<style>
.error{color:red}
</style>
</head>
<body>
<form:form action="helloagain" modelAttribute="emp">
Username: <form:input path="name"/> <br><br>
Password(*): <form:password path="pass"/>
```

```
<form:errors path="pass" cssClass="error"/><br><br>
<input type="submit" value="submit">
</form:form>
</body>
</html>
```

final.jsp

```
<html>
<body>
Username: ${emp.name} <br><br>
Password: ${emp.pass}
</body>
</html>
```

Output:



Let's submit the form without entering the password.

Spring - JDBC Framework Overview

While working with the database using plain old JDBC, it becomes cumbersome to write unnecessary code to handle exceptions, opening and closing database connections, etc. However, Spring JDBC Framework takes care of all the low-level details starting from opening the connection, prepare and execute the SQL statement, process exceptions, handle transactions and finally close the connection.

So what you have to do is just define the connection parameters and specify the SQL statement to be executed and do the required work for each iteration while fetching data from the database.

Spring JDBC provides several approaches and correspondingly different classes to interface with the database. I'm going to take classic and the most popular approach which makes use of **JdbcTemplate** class of the framework. This is the central framework class that manages all the database communication and exception handling.

JdbcTemplate Class

The JDBC Template class executes SQL queries, updates statements, stores procedure calls, performs iteration over ResultSets, and extracts returned parameter values. It also catches JDBC exceptions and translates them to the generic, more informative, exception hierarchy defined in the org.springframework.dao package.

Instances of the *JdbcTemplate* class are *threadsafe* once configured. So you can configure a single instance of a *JdbcTemplate* and then safely inject this shared reference into multiple DAOs.

A common practice when using the JDBC Template class is to configure a *DataSource* in your Spring configuration file, and then dependency-inject that shared DataSource bean into your DAO classes, and the JdbcTemplate is created in the setter for the DataSource.

Configuring Data Source

Let us create a database table **Student** in our database **TEST**. We assume you are working with MySQL database, if you work with any other database then you can change your DDL and SQL queries accordingly.

```
CREATE TABLE Student(
   ID   INT NOT NULL AUTO_INCREMENT,
   NAME VARCHAR(20) NOT NULL,
```

```
   AGE  INT NOT NULL,
   PRIMARY KEY (ID)
);
```

Now we need to supply a DataSource to the JDBC Template so it can configure itself to get database access. You can configure the DataSource in the XML file with a piece of code as shown in the following code snippet −

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

Data Access Object (DAO)

DAO stands for Data Access Object, which is commonly used for database interaction. DAOs exist to provide a means to read and write data to the database and they should expose this functionality through an interface by which the rest of the application will access them.

The DAO support in Spring makes it easy to work with data access technologies like JDBC, Hibernate, JPA, or JDO in a consistent way.

Executing SQL statements

Let us see how we can perform CRUD (Create, Read, Update and Delete) operation on database tables using SQL and JDBC Template object.

Querying for an integer

```
String SQL = "select count(*) from Student";
int rowCount = jdbcTemplateObject.queryForInt( SQL );
```

Querying for a long

```
String SQL = "select count(*) from Student";
long rowCount = jdbcTemplateObject.queryForLong( SQL );
```

A simple query using a bind variable

```
String SQL = "select age from Student where id = ?";
int age = jdbcTemplateObject.queryForInt(SQL, new Object[]{10});
```

Querying for a String

```
String SQL = "select name from Student where id = ?";
String name = jdbcTemplateObject.queryForObject(SQL, new Object[]{10}, String.class
```

Querying and returning an object

```
String SQL = "select * from Student where id = ?";
Student student = jdbcTemplateObject.queryForObject(
   SQL, new Object[]{10}, new StudentMapper());

public class StudentMapper implements RowMapper<Student> {
   public Student mapRow(ResultSet rs, int rowNum) throws SQLException {
      Student student = new Student();
      student.setID(rs.getInt("id"));
      student.setName(rs.getString("name"));
      student.setAge(rs.getInt("age"));

      return student;
   }
}
```

Querying and returning multiple objects

```
String SQL = "select * from Student";
List<Student> students = jdbcTemplateObject.query(
   SQL, new StudentMapper());

public class StudentMapper implements RowMapper<Student> {
   public Student mapRow(ResultSet rs, int rowNum) throws SQLException {
      Student student = new Student();
      student.setID(rs.getInt("id"));
      student.setName(rs.getString("name"));
      student.setAge(rs.getInt("age"));

      return student;
   }
}
```

Inserting a row into the table

```
String SQL = "insert into Student (name, age) values (?, ?)";
jdbcTemplateObject.update( SQL, new Object[]{"Zara", 11} );
```

Updating a row into the table

```
String SQL = "update Student set name = ? where id = ?";
jdbcTemplateObject.update( SQL, new Object[]{"Zara", 10} );
```

Deleting a row from the table

```
String SQL = "delete Student where id = ?";
jdbcTemplateObject.update( SQL, new Object[]{20} );
```

Executing DDL Statements

You can use the **execute(..)** method from *jdbcTemplate* to execute any SQL statements or DDL statements. Following is an example to use CREATE statement to create a table −

```
String SQL = "CREATE TABLE Student( " +
   "ID   INT NOT NULL AUTO_INCREMENT, " +
   "NAME VARCHAR(20) NOT NULL, " +
   "AGE  INT NOT NULL, " +
   "PRIMARY KEY (ID));"

jdbcTemplateObject.execute( SQL );
```

Spring JDBC Framework Examples

Based on the above concepts, let us check few important examples which will help you in understanding usage of JDBC framework in Spring −

Sr.No.	Example & Description
1	Spring JDBC Example This example will explain how to write a simple JDBC-based Spring application.
2	SQL Stored Procedure in Spring Learn how to call SQL stored procedure while using JDBC in Spring.

Useful Video Courses

Spring MVC CRUD Example

CRUD (Create, Read, Update and Delete) application is the most important application for creating any project. It provides an idea to develop a large project. In spring MVC, we can develop a simple CRUD application.

Here, we are using **JdbcTemplate** for database interaction.

Create a table

Here, we are using emp99 table present in the 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

Spring MVC CRUD Example

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='
//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 nam

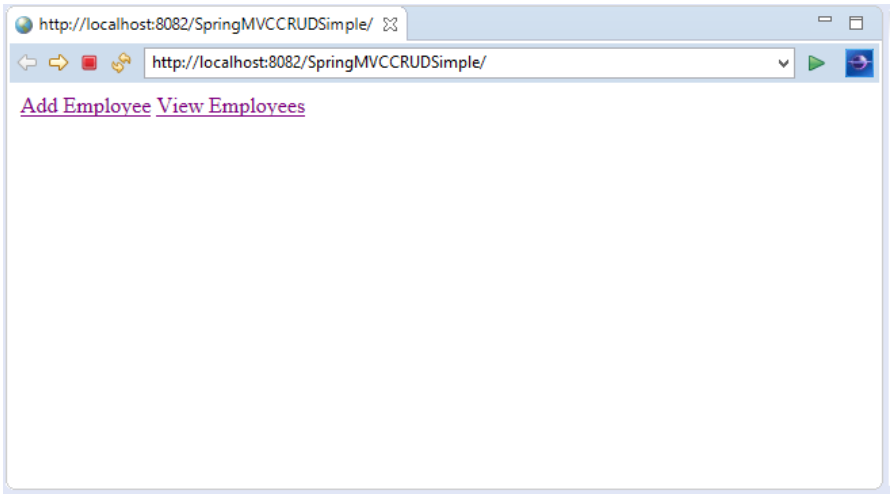
```
<%@ taglib uri="http://www.springframework.org/tags/form" prefix="form"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>

    <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>
```

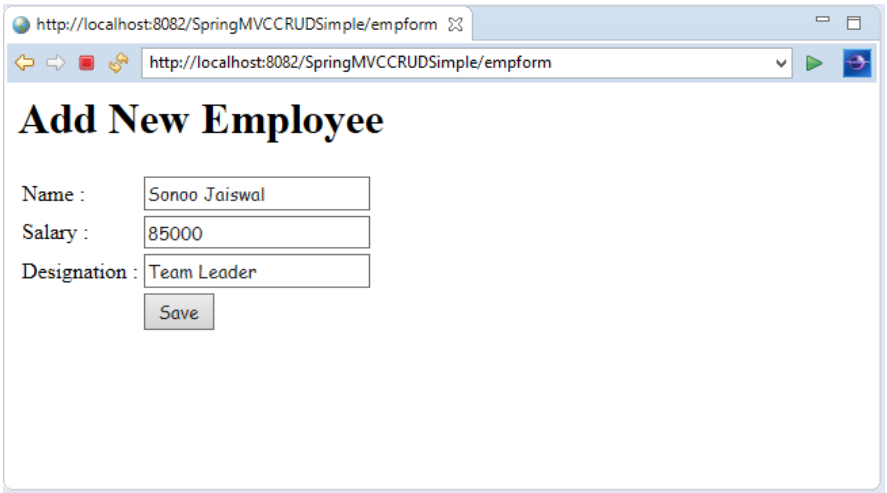
viewemp.jsp

```
<%@ 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>
```

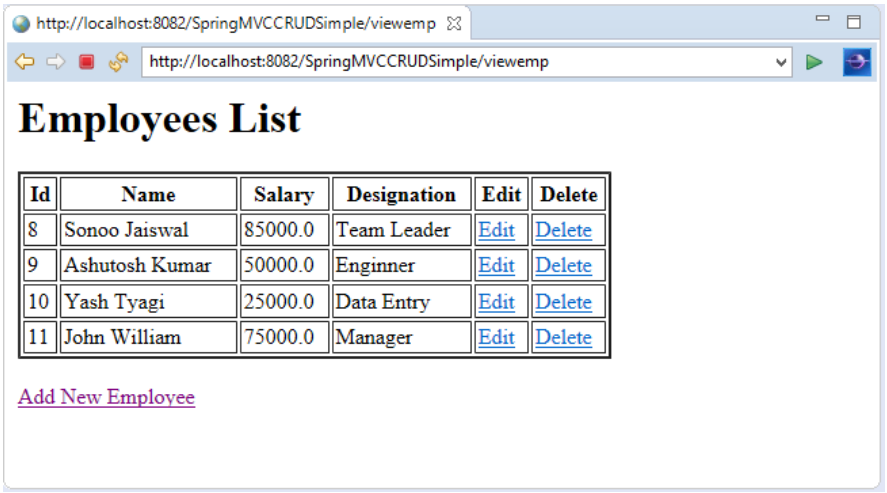
Output:



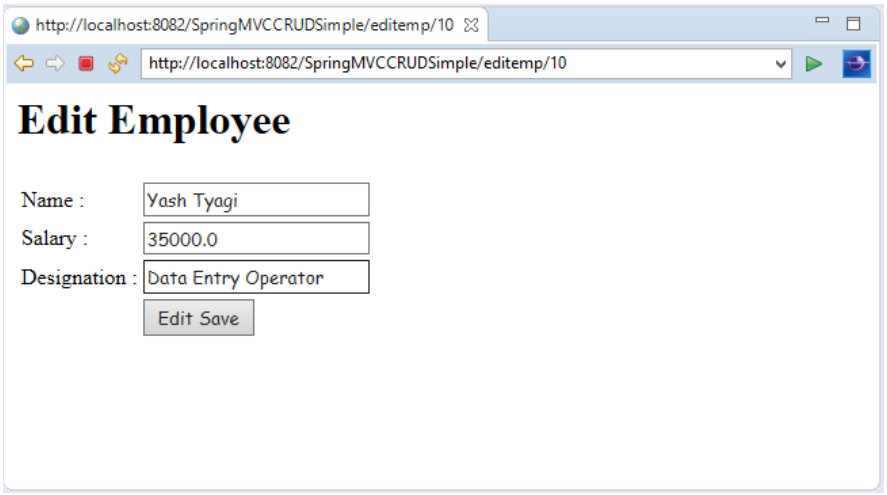
On clicking **Add Employee**, you will see the following form.



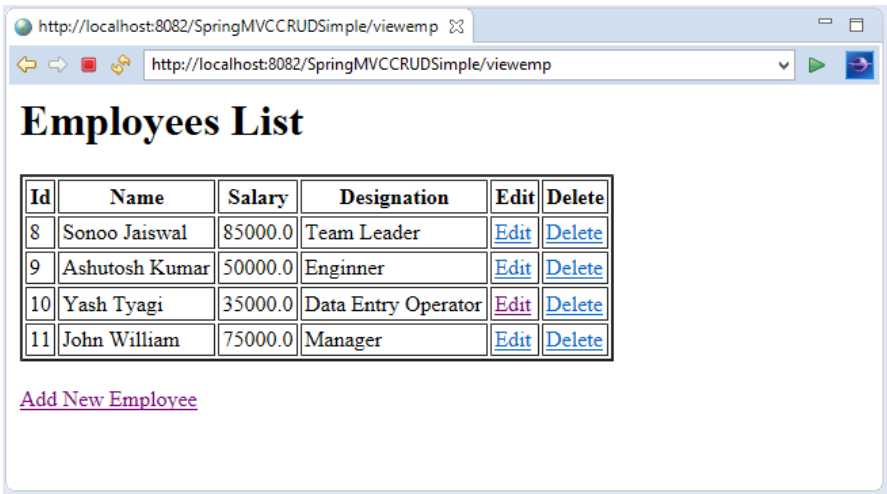
Fill the form and **click Save** to add the entry into the database.



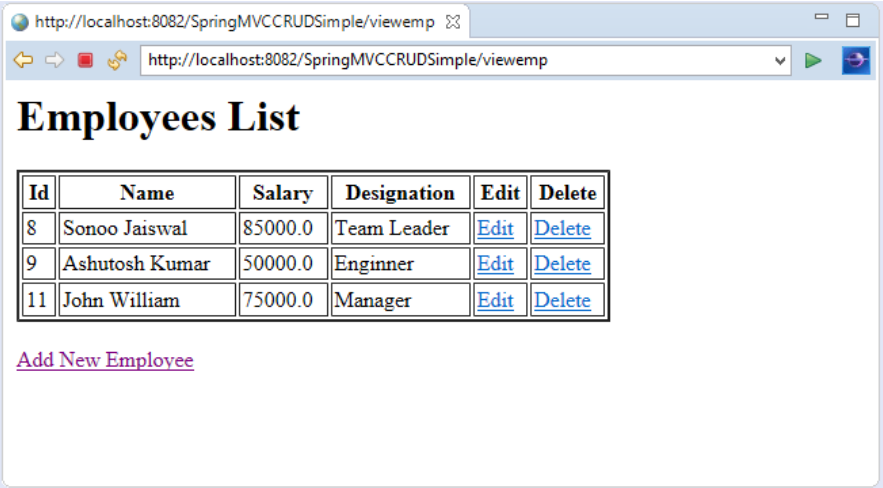
Now, click **Edit** to make some changes in the provided data.



Now, click **Edit Save** to add the entry with changes into the database.



Now, click **Delete** to delete the entry from the database.



Download this example (developed using Eclipse)

Download SQL File

Download SQL File

Download MYSQL-connector.jar file

If you are not using maven, download MYSQL-connector.jar.

← Prev

For Videos Join Our Youtube Channel: Join Now

Feedback

- Send your Feedback to feedback@javatpoint.com

Help Others, Please Share

Spring Security Architecture

This guide is a primer for Spring Security, offering insight into the design and basic building blocks of the framework. We cover only the very basics of application security. However, in doing so, we can clear up some of the confusion experienced by developers who use Spring Security. To do this, we take a look at the way security is applied in web applications by using filters and, more generally, by using method annotations. Use this guide when you need a high-level understanding of how a secure application works, how it can be customized, or if you need to learn how to think about application security.

This guide is not intended as a manual or recipe for solving more than the most basic problems (there are other sources for those), but it could be useful for beginners and experts alike. Spring Boot is also often referenced, because it provides some default behaviour for a secure application, and it can be useful to understand how that fits in with the overall architecture.

Note All of the principles apply equally well to applications that do not use Spring Boot.

Authentication and Access Control

Application security boils down to two more or less independent problems: authentication (who are you?) and authorization (what are you allowed to do?). Sometimes people say “access control” instead of "authorization", which can get confusing, but it can be helpful to think of it that way because “authorization” is overloaded in other places. Spring Security has an architecture that is designed to separate authentication from authorization and has strategies and extension points for both.

Authentication

The main strategy interface for authentication is `AuthenticationManager`, which has only one method:

```
public interface AuthenticationManager {

  Authentication authenticate(Authentication authentication)
    throws AuthenticationException;
}
```

An `AuthenticationManager` can do one of 3 things in its `authenticate()` method:

- Return an `Authentication` (normally with `authenticated=true`) if it can verify that the input represents a valid principal.
- Throw an `AuthenticationException` if it believes that the input represents an invalid principal.
- Return `null` if it cannot decide.

`AuthenticationException` is a runtime exception. It is usually handled by an application in a generic way, depending on the style or purpose of the application. In other words, user code is not normally expected to catch and handle it. For example, a web UI might render a page that says that the authentication failed, and a backend HTTP service might send a 401 response, with or without a `WWW-Authenticate` header depending on the context.

The most commonly used implementation of `AuthenticationManager` is `ProviderManager`, which delegates to a chain of `AuthenticationProvider` instances. An `AuthenticationProvider` is a bit like an `AuthenticationManager`, but it has an extra method to allow the caller to query whether it supports a given `Authentication` type:

```
public interface AuthenticationProvider {

	Authentication authenticate(Authentication authentication)
			throws AuthenticationException;

	boolean supports(Class<?> authentication);
}
```

The `Class<?>` argument in the `supports()` method is really `Class<? extends Authentication>` (it is only ever asked if it supports something that

is passed into the `authenticate()` method). A `ProviderManager` can support multiple different authentication mechanisms in the same application by delegating to a chain of `AuthenticationProviders`. If a `ProviderManager` does not recognize a particular `Authentication` instance type, it is skipped.

A `ProviderManager` has an optional parent, which it can consult if all providers return `null`. If the parent is not available, a `null` `Authentication` results in an `AuthenticationException`.

Sometimes, an application has logical groups of protected resources (for example, all web resources that match a path pattern, such as `/api/**`), and each group can have its own dedicated `AuthenticationManager`. Often, each of those is a `ProviderManager`, and they share a parent. The parent is then a kind of "global" resource, acting as a fallback for all providers.

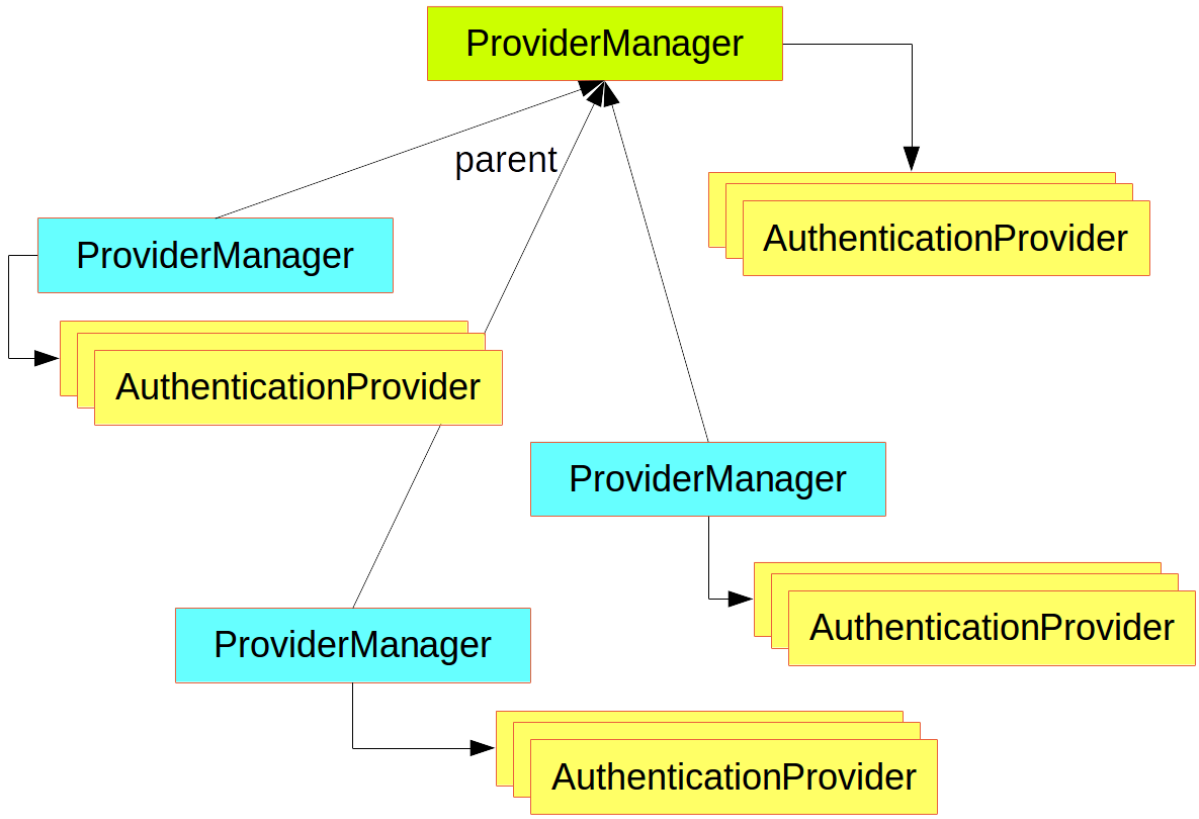


Figure 1. An `AuthenticationManager` hierarchy using `ProviderManager`

Customizing Authentication Managers

Spring Security provides some configuration helpers to quickly get common authentication manager features set up in your application. The most commonly used helper is the `AuthenticationManagerBuilder`, which is great for setting up in-memory, JDBC, or LDAP user details or for adding a custom `UserDetailsService`. The following example shows an application that configures the global (parent) `AuthenticationManager`:

```
@Configuration
public class ApplicationSecurity extends
WebSecurityConfigurerAdapter {

   ... // web stuff here

  @Autowired
  public void initialize(AuthenticationManagerBuilder builder,
DataSource dataSource) {

builder.jdbcAuthentication().dataSource(dataSource).withUser("dav
e")
       .password("secret").roles("USER");
  }

}
```

This example relates to a web application, but the usage of `AuthenticationManagerBuilder` is more widely applicable (see Web Security for more detail on how web application security is implemented). Note that the `AuthenticationManagerBuilder` is `@Autowired` into a method in a `@Bean` — that is what makes it build the global (parent) `AuthenticationManager`. In contrast, consider the following example:

```
@Configuration
public class ApplicationSecurity extends
WebSecurityConfigurerAdapter {

  @Autowired
  DataSource dataSource;

   ... // web stuff here

  @Override
  public void configure(AuthenticationManagerBuilder builder) {
```

```
builder.jdbcAuthentication().dataSource(dataSource).withUser("dav
e")
       .password("secret").roles("USER");
  }

}
```

If we had used an `@Override` of a method in the configurer, the `AuthenticationManagerBuilder` would be used only to build a "local" `AuthenticationManager`, which would be a child of the global one. In a Spring Boot application, you can `@Autowired` the global one into another bean, but you cannot do that with the local one unless you explicitly expose it yourself.

Spring Boot provides a default global `AuthenticationManager` (with only one user) unless you pre-empt it by providing your own bean of type `AuthenticationManager`. The default is secure enough on its own for you not to have to worry about it much, unless you actively need a custom global `AuthenticationManager`. If you do any configuration that builds an `AuthenticationManager`, you can often do it locally to the resources that you are protecting and not worry about the global default.

Authorization or Access Control

Once authentication is successful, we can move on to authorization, and the core strategy here is `AccessDecisionManager`. There are three implementations provided by the framework and all three delegate to a chain of `AccessDecisionVoter` instances, a bit like the `ProviderManager` delegates to `AuthenticationProviders`.

An `AccessDecisionVoter` considers an `Authentication` (representing a principal) and a secure `Object`, which has been decorated with `ConfigAttributes`:

```
boolean supports(ConfigAttribute attribute);

boolean supports(Class<?> clazz);

int vote(Authentication authentication, S object,
```

```
        Collection<ConfigAttribute> attributes);
```

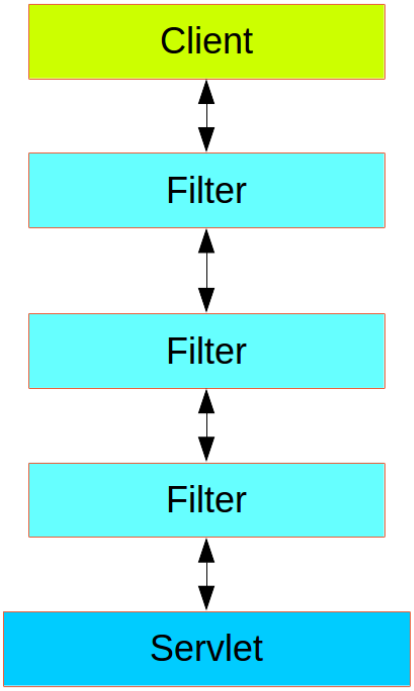
The `Object` is completely generic in the signatures of the `AccessDecisionManager` and `AccessDecisionVoter`. It represents anything that a user might want to access (a web resource or a method in a Java class are the two most common cases). The `ConfigAttributes` are also fairly generic, representing a decoration of the secure `Object` with some metadata that determines the level of permission required to access it. `ConfigAttribute` is an interface. It has only one method (which is quite generic and returns a `String`), so these strings encode in some way the intention of the owner of the resource, expressing rules about who is allowed to access it. A typical `ConfigAttribute` is the name of a user role (like `ROLE_ADMIN` or `ROLE_AUDIT`), and they often have special formats (like the `ROLE_` prefix) or represent expressions that need to be evaluated.

Most people use the default `AccessDecisionManager`, which is `AffirmativeBased` (if any voters return affirmatively, access is granted). Any customization tends to happen in the voters, either by adding new ones or modifying the way that the existing ones work.

It is very common to use `ConfigAttributes` that are Spring Expression Language (SpEL) expressions — for example, `isFullyAuthenticated() && hasRole('user')`. This is supported by an `AccessDecisionVoter` that can handle the expressions and create a context for them. To extend the range of expressions that can be handled requires a custom implementation of `SecurityExpressionRoot` and sometimes also `SecurityExpressionHandler`.

Web Security

Spring Security in the web tier (for UIs and HTTP back ends) is based on Servlet `Filters`, so it is helpful to first look at the role of `Filters` generally. The following picture shows the typical layering of the handlers for a single HTTP request.



The client sends a request to the application, and the container decides which filters and which servlet apply to it based on the path of the request URI. At most, one servlet can handle a single request, but filters form a chain, so they are ordered. In fact, a filter can veto the rest of the chain if it wants to handle the request itself. A filter can also modify the request or the response used in the downstream filters and servlet. The order of the filter chain is very important, and Spring Boot manages it through two mechanisms: `@Beans` of type `Filter` can have an `@Order` or implement `Ordered`, and they can be part of a `FilterRegistrationBean` that itself has an order as part of its API. Some off-the-shelf filters define their own constants to help signal what order they like to be in relative to each other (for example, the `SessionRepositoryFilter` from Spring Session has a `DEFAULT_ORDER` of `Integer.MIN_VALUE + 50`, which tells us it likes to be early in the chain, but it does not rule out other filters coming before it).

Spring Security is installed as a single `Filter` in the chain, and its concrete type is `FilterChainProxy`, for reasons that we cover soon. In a Spring Boot application, the security filter is a `@Bean` in the `ApplicationContext`, and it is

installed by default so that it is applied to every request. It is installed at a position defined by `SecurityProperties.DEFAULT_FILTER_ORDER`, which in turn is anchored by `FilterRegistrationBean.REQUEST_WRAPPER_FILTER_MAX_ORDER` (the maximum order that a Spring Boot application expects filters to have if they wrap the request, modifying its behavior). There is more to it than that, though: From the point of view of the container, Spring Security is a single filter, but, inside of it, there are additional filters, each playing a special role. The following image shows this relationship:

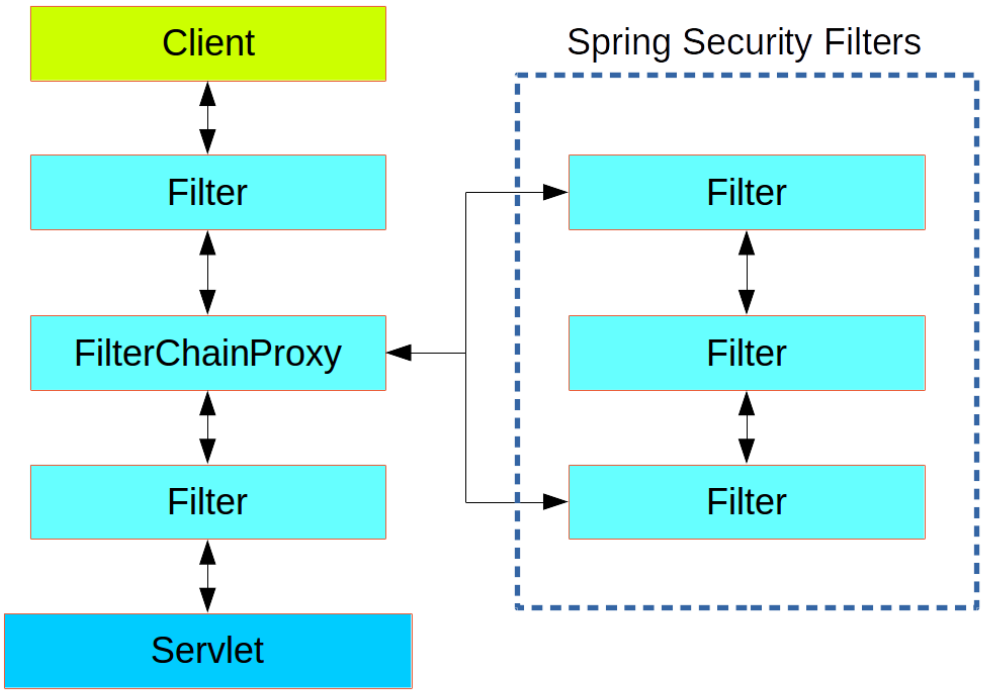


Figure 2. Spring Security is a single physical `Filter` but delegates processing to a chain of internal filters

In fact, there is even one more layer of indirection in the security filter: It is usually installed in the container as a `DelegatingFilterProxy`, which does not have to be a Spring `@Bean`. The proxy delegates to a `FilterChainProxy`, which is always a `@Bean`, usually with a fixed name of `springSecurityFilterChain`. It is the `FilterChainProxy` that contains all

the security logic arranged internally as a chain (or chains) of filters. All the filters have the same API (they all implement the `Filter` interface from the Servlet specification), and they all have the opportunity to veto the rest of the chain.

There can be multiple filter chains all managed by Spring Security in the same top level `FilterChainProxy` and all are unknown to the container. The Spring Security filter contains a list of filter chains and dispatches a request to the first chain that matches it. The following picture shows the dispatch happening based on matching the request path (`/foo/**` matches before `/**`). This is very common but not the only way to match a request. The most important feature of this dispatch process is that only one chain ever handles a request.

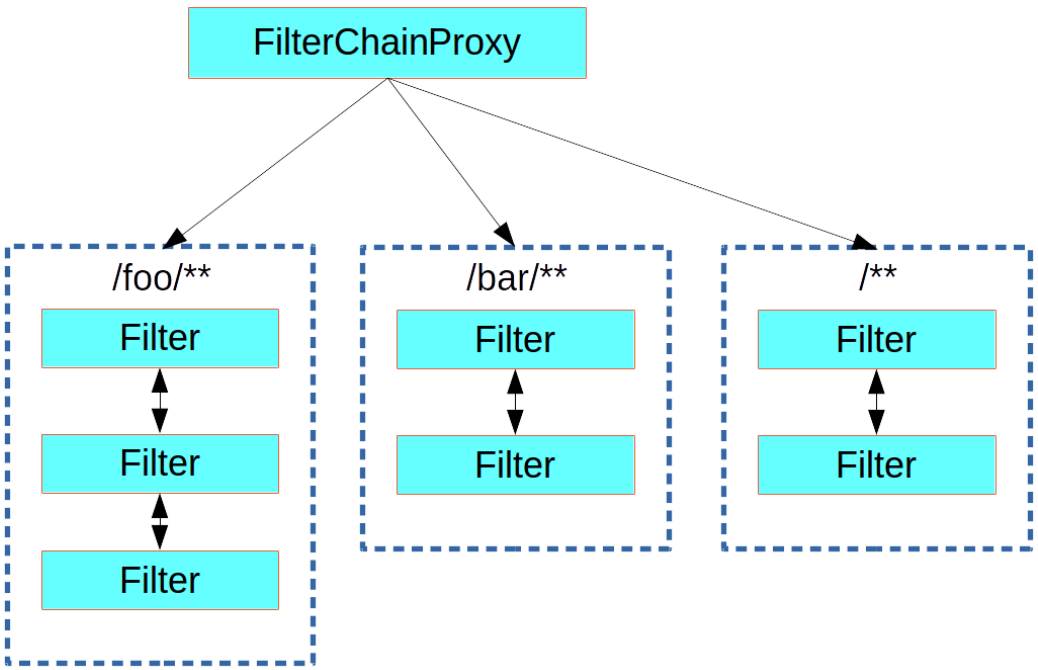


Figure 3. The Spring Security `FilterChainProxy` dispatches requests to the first chain that matches.

A vanilla Spring Boot application with no custom security configuration has a several (call it n) filter chains, where usually n=6. The first (n-1) chains are there just to ignore static resource patterns, like `/css/**` and `/images/**`, and the error view: `/error`. (The paths can be controlled by the user

with `security.ignored` from the `SecurityProperties` configuration bean.) The last chain matches the catch-all path (`/**`) and is more active, containing logic for authentication, authorization, exception handling, session handling, header writing, and so on. There are a total of 11 filters in this chain by default, but normally it is not necessary for users to concern themselves with which filters are used and when.

Note The fact that all filters internal to Spring Security are unknown to the container is important, especially in a Spring Boot application, where, by default, all `@Beans` of type `Filter` are registered automatically with the container. So if you want to add a custom filter to the security chain, you need to either not make it be a `@Bean` or wrap it in a `FilterRegistrationBean` that explicitly disables the container registration.

Creating and Customizing Filter Chains

The default fallback filter chain in a Spring Boot application (the one with the `/**` request matcher) has a predefined order of `SecurityProperties.BASIC_AUTH_ORDER`. You can switch it off completely by setting `security.basic.enabled=false`, or you can use it as a fallback and define other rules with a lower order. To do the latter, add a `@Bean` of type `WebSecurityConfigurerAdapter` (or `WebSecurityConfigurer`) and decorate the class with `@Order`, as follows:

```
@Configuration
@Order(SecurityProperties.BASIC_AUTH_ORDER - 10)
public class ApplicationConfigurerAdapter extends
WebSecurityConfigurerAdapter {
  @Override
  protected void configure(HttpSecurity http) throws Exception {
    http.antMatcher("/match1/**")
     ...;
  }
}
```

This bean causes Spring Security to add a new filter chain and order it before the fallback.

Many applications have completely different access rules for one set of resources compared to another. For example, an application that hosts a UI and a backing API might support cookie-based authentication with a redirect to a login page for the UI parts and token-based authentication with a 401 response to unauthenticated requests for the API parts. Each set of resources has its own `WebSecurityConfigurerAdapter` with a unique order and its own request matcher. If the matching rules overlap, the earliest ordered filter chain wins.

Request Matching for Dispatch and Authorization

A security filter chain (or, equivalently, a `WebSecurityConfigurerAdapter`) has a request matcher that is used to decide whether to apply it to an HTTP request. Once the decision is made to apply a particular filter chain, no others are applied. However, within a filter chain, you can have more fine-grained control of authorization by setting additional matchers in the `HttpSecurity` configurer, as follows:

```
@Configuration
@Order(SecurityProperties.BASIC_AUTH_ORDER - 10)
public class ApplicationConfigurerAdapter extends
WebSecurityConfigurerAdapter {
  @Override
  protected void configure(HttpSecurity http) throws Exception {
    http.antMatcher("/match1/**")
      .authorizeRequests()
        .antMatchers("/match1/user").hasRole("USER")
        .antMatchers("/match1/spam").hasRole("SPAM")
        .anyRequest().isAuthenticated();
  }
}
```

One of the easiest mistakes to make when configuring Spring Security is to forget that these matchers apply to different processes. One is a request matcher for the whole filter chain, and the other is only to choose the access rule to apply.

Combining Application Security Rules with Actuator Rules

If you use the Spring Boot Actuator for management endpoints, you probably want them to be secure, and, by default, they are. In fact, as soon as you add the Actuator to a secure application, you get an additional filter chain that applies only to the actuator endpoints. It is defined with a request matcher that matches only actuator endpoints and it has an order of `ManagementServerProperties.BASIC_AUTH_ORDER`, which is 5 fewer than the default `SecurityProperties` fallback filter, so it is consulted before the fallback.

If you want your application security rules to apply to the actuator endpoints, you can add a filter chain that is ordered earlier than the actuator one and that has a request matcher that includes all actuator endpoints. If you prefer the default security settings for the actuator endpoints, the easiest thing is to add your own filter later than the actuator one, but earlier than the fallback (for example, `ManagementServerProperties.BASIC_AUTH_ORDER + 1`), as follows:

```
@Configuration
@Order(ManagementServerProperties.BASIC_AUTH_ORDER + 1)
public class ApplicationConfigurerAdapter extends
WebSecurityConfigurerAdapter {
  @Override
  protected void configure(HttpSecurity http) throws Exception {
    http.antMatcher("/foo/**")
      ...;
  }
}
```

Note Spring Security in the web tier is currently tied to the Servlet API, so it is only really applicable when running an application in a servlet container, either embedded or otherwise. It is not, however, tied to Spring MVC or the rest of the Spring web stack, so it can be used in any servlet application — for instance, one using JAX-RS.

Method Security

As well as support for securing web applications, Spring Security offers support for applying access rules to Java method executions. For Spring Security, this is just a different type of “protected resource”. For users, it means the access rules are declared using the same format of `ConfigAttribute` strings (for example,

roles or expressions) but in a different place in your code. The first step is to enable method security — for example, in the top level configuration for our application:

```
@SpringBootApplication
@EnableGlobalMethodSecurity(securedEnabled = true)
public class SampleSecureApplication {
}
```

Then we can decorate the method resources directly:

```
@Service
public class MyService {

  @Secured("ROLE_USER")
  public String secure() {
    return "Hello Security";
  }

}
```

This example is a service with a secure method. If Spring creates a `@Bean` of this type, it is proxied and callers have to go through a security interceptor before the method is actually executed. If access is denied, the caller gets an `AccessDeniedException` instead of the actual method result.

There are other annotations that you can use on methods to enforce security constraints, notably `@PreAuthorize` and `@PostAuthorize`, which let you write expressions containing references to method parameters and return values, respectively.

Tip: It is not uncommon to combine Web security and method security. The filter chain provides the user experience features, such as authentication and redirect to login pages and so on, and the method security provides protection at a more granular level.

Working with Threads

Spring Security is fundamentally thread-bound, because it needs to make the current authenticated principal available to a wide variety of downstream

consumers. The basic building block is the `SecurityContext`, which may contain an `Authentication` (and when a user is logged in it is an `Authentication` that is explicitly `authenticated`). You can always access and manipulate the `SecurityContext` through static convenience methods in `SecurityContextHolder`, which, in turn, manipulate a `ThreadLocal`. The following example shows such an arrangement:

```
SecurityContext context = SecurityContextHolder.getContext();
Authentication authentication = context.getAuthentication();
assert(authentication.isAuthenticated);
```

It is **not** common for user application code to do this, but it can be useful if you, for instance, need to write a custom authentication filter (although, even then, there are base classes in Spring Security that you can use so that you could avoid needing to use the `SecurityContextHolder`).

If you need access to the currently authenticated user in a web endpoint, you can use a method parameter in a `@RequestMapping`, as follows:

```
@RequestMapping("/foo")
public String foo(@AuthenticationPrincipal User user) {
  ... // do stuff with user
}
```

This annotation pulls the current `Authentication` out of the `SecurityContext` and calls the `getPrincipal()` method on it to yield the method parameter. The type of the `Principal` in an `Authentication` is dependent on the `AuthenticationManager` used to validate the authentication, so this can be a useful little trick to get a type-safe reference to your user data.

If Spring Security is in use, the `Principal` from the `HttpServletRequest` is of type `Authentication`, so you can also use that directly:

```
@RequestMapping("/foo")
public String foo(Principal principal) {
  Authentication authentication = (Authentication) principal;
  User = (User) authentication.getPrincipal();
```

```
  ... // do stuff with user
}
```

This can sometimes be useful if you need to write code that works when Spring Security is not in use (you would need to be more defensive about loading the `Authentication` class).

Processing Secure Methods Asynchronously

Since the `SecurityContext` is thread-bound, if you want to do any background processing that calls secure methods (for example, with `@Async`), you need to ensure that the context is propagated. This boils down to wrapping the `SecurityContext` with the task (`Runnable`, `Callable`, and so on) that is executed in the background. Spring Security provides some helpers to make this easier, such as wrappers for `Runnable` and `Callable`. To propagate the `SecurityContext` to `@Async` methods, you need to supply an `AsyncConfigurer` and ensure the `Executor` is of the correct type:

```
@Configuration
public class ApplicationConfiguration extends
AsyncConfigurerSupport {

  @Override
  public Executor getAsyncExecutor() {
    return new
DelegatingSecurityContextExecutorService(Executors.newFixedThread
Pool(5));
  }

}
```

Spring Security Tutorial

Spring Security Tutorial provides basic and advanced concepts of Spring Security. Our Spring Security Tutorial is designed for beginners and professionals both.

Our Spring Security Tutorial includes all topics of Spring Security such as spring security introduction, features, project modules, xml example, java example, login logout, spring boot etc.

Prerequisite

To learn Spring Security, you must have the basic knowledge of HTML and CSS

Introduction

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.

Technologies that support Spring Security Integration

Spring Security framework supports wide range of authentication models. These models either provided by third parties or framework itself. Spring Security supports integration with all of these technologies.

- HTTP BASIC authentication headers
- HTTP 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 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 History

In late 2003, a project **Acegi Security System for Spring** started with the intention to develop a Spring-based security system. So, a simple security system was implemented but not released officially. Developers used that code internally for their solutions and by 2004 about 20 developers were using that.

Initially, authentication module was not part of the project, around a year after, module was added and complete project was reconfigure to support more technologies.

After some time this project became a subproject of Spring framework and released as 1.0.0 in 2006.

in 2007, project is renamed to Spring Security and widely accepted. Currently, it is recognized and supported by developers open community world wide.

Spring Security Features

- LDAP (Lightweight Directory Access Protocol)
- Single sign-on
- JAAS (Java Authentication and Authorization Service) LoginModule
- Basic Access Authentication
- Digest Access Authentication
- Remember-me
- Web Form Authentication
- Authorization
- Software Localization
- HTTP Authorization

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 to 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 remember to the user and avoid login again from the same machine until the user logout.

Web Form Authentication

In this process, web form 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 the 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.

Features added in Spring Security 5.0

OAuth 2.0 Login

This feature provides the facility to the user to login into the application by using their existing account at GitHub or Google. This feature is implemented by using the Authorization Code Grant that is specified in the OAuth 2.0 Authorization Framework.

Reactive Support

From version Spring Security 5.0, it provides reactive programming and reactive web runtime support and even, we can integrate with Spring WebFlux.

Modernized Password Encoding

Spring Security 5.0 introduced new Password encoder **DelegatingPasswordEncoder** which is more modernize and solve all the problems of previous encoder **NoOpPasswordEncoder**.

Spring Project Modules

In Spring Security 3.0, the Security module is divided into separate jar files. The purpose was to divide jar files based on their functionalities, so, the developer can integrate according to their requirement.

It also helps to set required dependency into pom.xml file of maven project.

The following are the jar files that are included into Spring Security module.

- spring-security-core.jar
- spring-security-remoting.jar

- spring-security-web.jar
- spring-security-config.jar
- spring-security-ldap.jar
- spring-security-oauth2-core.jar
- spring-security-oauth2-client.jar
- spring-security-oauth2-jose.jar
- spring-security-acl.jar
- spring-security-cas.jar
- spring-security-openid.jar
- spring-security-test.jar

Core - spring-security-core.jar

This is core jar file and required for every application that wants to use Spring Security. This jar file includes core access-control and core authentication classes and interfaces. We can use it in standalone applications or remote clients applications.

It contains top level packages:

- org.springframework.security.core
- org.springframework.security.access
- org.springframework.security.authentication
- org.springframework.security.provisioning

Remoting - spring-security-remoting.jar

This jar is used to integrate security feature into the Spring remote application. We don't need it until or unless we are creating remote application. All the classes and interfaces are located into **org.springframework.security.remoting** package.

Web - spring-security-web.jar

This jar is useful for Spring Security web authentication and URL-based access control. It includes filters and web-security infrastructure.

All the classes and interfaces are located into the **org.springframework.security.web** package.

Config - spring-security-config.jar

This jar file is required for Spring Security configuration using XML and Java both. It includes Java configuration code and security namespace parsing code. All the classes and interfaces are stored in **org.springframework.security.config** package.

LDAP - spring-security-ldap.jar

This jar file is required only if we want to use LDAP (Lighweight Directory Access Protocol). It includes authentication and provisioning code. All the classes and interfaces are stored into **org.springframework.security.ldap** package.

OAuth 2.0 Core - spring-security-oauth2-core.jar

This jar is required to integrate Oauth 2.0 Authorization Framework and OpenID Connect Core 1.0 into the application. This jar file includes the core classes for OAuth 2.0 and classes are stored into the **org.springframework.security.oauth2.core** package.

OAuth 2.0 Client - spring-security-oauth2-client.jar

This jar file is required to get client support for OAuth 2.0 Authorization Framework and OpenID Connect Core 1.0. This module provides OAuth login and OpenID client support. All the classes and interfaces are available from **org.springframework.security.oauth2.client** package.

OAuth 2.0 JOSE - spring-security-oauth2-jose.jar

It provides Spring Security's support for the JOSE (Javascript Object Signing and Encryption) framework. The JOSE framework provides methods to establish secure connection between clients. It contains following collection of specifications:

- JWT (JSON Web Token)
- JWS (JSON Web Signature)
- JWE (JSON Web Encryption)
- JWK (JSON Web Key)

All the classes and interfaces are available into these two packages:

org.springframework.security.oauth2.jwt and **org.springframework.security.oauth2.jose.**

ACL - spring-security-acl.jar

This jar is used to apply security to domain object in the application. We can access classes and code from the **org.springframework.security.acls** package.

CAS - spring-security-cas.jar

It is required for Spring Security?s CAS client integration. We can use it to integrate Spring Security web authentication with CAS single sign-on server. The source code is located into **org.springframework.security.cas** package.

OpenID - spring-security-openid.jar

This jar is used for OpenID web authentication support. We can use it to authenticate users against an external OpenID server. It requires OpenID4Java and top level package is **org.springframework.security.openid**.

Test - spring-security-test.jar

This jar provides support for testing Spring Security application.

---

1. **Cross Site Request Forgery (CSRF)**

Before we discuss how Spring Security can protect applications from CSRF attacks, we will explain what a CSRF attack is. Let's take a look at a concrete example to get a better understanding.

Assume that your bank's website provides a form that allows transferring money from the currently logged in user to another bank account. For example, the HTTP request might look like:

```
POST /transfer HTTP/1.1
Host: bank.example.com
Cookie: JSESSIONID=randomid; Domain=bank.example.com; Secure; HttpOnly
Content-Type: application/x-www-form-urlencoded

amount=100.00&routingNumber=1234&account=9876
```

Now pretend you authenticate to your bank's website and then, without logging out, visit an evil website. The evil website contains an HTML page with the following form:

```
<form action="https://bank.example.com/transfer" method="post">
<input type="hidden"
        name="amount"
        value="100.00"/>
<input type="hidden"
        name="routingNumber"
        value="evilsRoutingNumber"/>
<input type="hidden"
```

```
        name="account"
        value="evilsAccountNumber"/>
<input type="submit"
        value="Win Money!"/>
</form>
```

You like to win money, so you click on the submit button. In the process, you have unintentionally transferred $100 to a malicious user. This happens because, while the evil website cannot see your cookies, the cookies associated with your bank are still sent along with the request.

Worst yet, this whole process could have been automated using JavaScript. This means you didn't even need to click on the button. So how do we protect ourselves from such attacks?

1.2 Synchronizer Token Pattern

The issue is that the HTTP request from the bank's website and the request from the evil website are exactly the same. This means there is no way to reject requests coming from the evil website and allow requests coming from the bank's website. To protect against CSRF attacks we need to ensure there is something in the request that the evil site is unable to provide.

One solution is to use the Synchronizer Token Pattern. This solution is to ensure that each request requires, in addition to our session cookie, a randomly generated token as an HTTP parameter. When a request is submitted, the server must look up the expected value for the parameter and compare it against the actual value in the request. If the values do not match, the request should fail.

We can relax the expectations to only require the token for each HTTP request that updates state. This can be safely done since the same origin policy ensures the evil site cannot read the response. Additionally, we do not want to include the random token in HTTP GET as this can cause the tokens to be leaked.

Let's take a look at how our example would change. Assume the randomly generated token is present in an HTTP parameter named _csrf. For example, the request to transfer money would look like this:

```
POST /transfer HTTP/1.1
Host: bank.example.com
Cookie: JSESSIONID=randomid; Domain=bank.example.com; Secure; HttpOnly
Content-Type: application/x-www-form-urlencoded

amount=100.00&routingNumber=1234&account=9876&_csrf=<secure-random>
```

You will notice that we added the _csrf parameter with a random value. Now the evil website will not be able to guess the correct value for the _csrf parameter (which must be explicitly provided on the evil website) and the transfer will fail when the server compares the actual token to the expected token.

1.3 When to use CSRF protection

When should you use CSRF protection? Our recommendation is to use CSRF protection for any request that could be processed by a browser by normal users. If you are only creating a service that is used by non-browser clients, you will likely want to disable CSRF protection.

1.3.1 CSRF protection and JSON

A common question is "do I need to protect JSON requests made by javascript?" The short answer is, it depends. However, you must be very careful as there are CSRF exploits that can impact JSON requests. For example, a malicious user can create a CSRF with JSON using the following form:

```
<form action="https://bank.example.com/transfer" method="post"
enctype="text/plain">
<input
name='{"amount":100,"routingNumber":"evilsRoutingNumber","account":"evilsAcco
untNumber", "ignore_me":"' value='test"}' type='hidden'>
<input type="submit"
        value="Win Money!"/>
</form>
```

This will produce the following JSON structure

```
{ "amount": 100,
"routingNumber": "evilsRoutingNumber",
"account": "evilsAccountNumber",
"ignore_me": "=test"
}
```

If an application were not validating the Content-Type, then it would be exposed to this exploit. Depending on the setup, a Spring MVC application that validates the Content-Type could still be exploited by updating the URL suffix to end with ".json" as shown below:

```
<form action="https://bank.example.com/transfer.json" method="post"
enctype="text/plain">
<input
name='{"amount":100,"routingNumber":"evilsRoutingNumber","account":"evilsAcco
untNumber", "ignore_me":"' value='test"}' type='hidden'>
<input type="submit"
        value="Win Money!"/>
</form>
```

1.3.2 CSRF and Stateless Browser Applications

What if my application is stateless? That doesn't necessarily mean you are protected. In fact, if a user does not need to perform any actions in the web browser for a given request, they are likely still vulnerable to CSRF attacks.

For example, consider an application uses a custom cookie that contains all the state within it for authentication instead of the JSESSIONID. When the CSRF attack is made the custom cookie will be sent with the request in the same manner that the JSESSIONID cookie was sent in our previous example.

Users using basic authentication are also vulnerable to CSRF attacks since the browser will automatically include the username password in any requests in the same manner that the JSESSIONID cookie was sent in our previous example.

1.4 Using Spring Security CSRF Protection

So what are the steps necessary to use Spring Security's to protect our site against CSRF attacks? The steps to using Spring Security's CSRF protection are outlined below:

- **Use proper HTTP verbs**
- **Configure CSRF Protection**
- **Include the CSRF Token**

1.4.1 Use proper HTTP verbs

The first step to protecting against CSRF attacks is to ensure your website uses proper HTTP verbs. Specifically, before Spring Security's CSRF support can be of use, you need to be certain that your application is using PATCH, POST, PUT, and/or DELETE for anything that modifies state.

This is not a limitation of Spring Security's support, but instead a general requirement for proper CSRF prevention. The reason is that including private information in an HTTP GET can cause the information to be leaked. See RFC 2616 Section 15.1.3 Encoding Sensitive Information in URI's for general guidance on using POST instead of GET for sensitive information.

1.4.2 Configure CSRF Protection

The next step is to include Spring Security's CSRF protection within your application. Some frameworks handle invalid CSRF tokens by invalidating the user's session, but this causes its own problems. Instead by default Spring Security's CSRF protection will produce an HTTP 403 access denied. This can be customized by configuring the AccessDeniedHandler to process `InvalidCsrfTokenException` differently.

As of Spring Security 4.0, CSRF protection is enabled by default with XML configuration. If you would like to disable CSRF protection, the corresponding XML configuration can be seen below.

```
<http>
        <!-- ... -->
        <csrf disabled="true"/>
```

```
</http>
```

CSRF protection is enabled by default with Java Configuration. If you would like to disable CSRF, the corresponding Java configuration can be seen below. Refer to the Javadoc of csrf() for additional customizations in how CSRF protection is configured.

```
@EnableWebSecurity
public class WebSecurityConfig extends
WebSecurityConfigurerAdapter {

	@Override
	protected void configure(HttpSecurity http) throws Exception {
		http
			.csrf().disable();
	}
}
```

1.4.3 Include the CSRF Token

Form Submissions

The last step is to ensure that you include the CSRF token in all PATCH, POST, PUT, and DELETE methods. One way to approach this is to use the `_csrf` request attribute to obtain the current `CsrfToken`. An example of doing this with a JSP is shown below:

```
<c:url var="logoutUrl" value="/logout"/>
<form action="${logoutUrl}"
	method="post">
<input type="submit"
	value="Log out" />
<input type="hidden"
	name="${_csrf.parameterName}"
	value="${_csrf.token}"/>
</form>
```

An easier approach is to use the csrfInput tag from the Spring Security JSP tag library.

If you are using Spring MVC `<form:form>` tag or Thymeleaf 2.1+ and are using `@EnableWebSecurity`, the `CsrfToken` is automatically included for you (using the `CsrfRequestDataValueProcessor`).

Ajax and JSON Requests

If you are using JSON, then it is not possible to submit the CSRF token within an HTTP parameter. Instead you can submit the token within a HTTP header. A typical pattern

would be to include the CSRF token within your meta tags. An example with a JSP is shown below:

```
<html>
<head>
	<meta name="_csrf" content="${_csrf.token}"/>
	<!-- default header name is X-CSRF-TOKEN -->
	<meta name="_csrf_header" content="${_csrf.headerName}"/>
	<!-- ... -->
</head>
<!-- ... -->
```

Instead of manually creating the meta tags, you can use the simpler csrfMetaTags tag from the Spring Security JSP tag library.

You can then include the token within all your Ajax requests. If you were using jQuery, this could be done with the following:

```
$(function () {
var token = $("meta[name='_csrf']").attr("content");
var header = $("meta[name='_csrf_header']").attr("content");
$(document).ajaxSend(function(e, xhr, options) {
	xhr.setRequestHeader(header, token);
});
});
```

As an alternative to jQuery, we recommend using cujoJS's rest.js. The rest.js module provides advanced support for working with HTTP requests and responses in RESTful ways. A core capability is the ability to contextualize the HTTP client adding behavior as needed by chaining interceptors on to the client.

```
var client = rest.chain(csrf, {
token: $("meta[name='_csrf']").attr("content"),
name: $("meta[name='_csrf_header']").attr("content")
});
```

The configured client can be shared with any component of the application that needs to make a request to the CSRF protected resource. One significant difference between rest.js and jQuery is that only requests made with the configured client will contain the CSRF token, vs jQuery where all requests will include the token. The ability to scope which requests receive the token helps guard against leaking the CSRF token to a third party. Please refer to the rest.js reference documentation for more information on rest.js.

CookieCsrfTokenRepository

There can be cases where users will want to persist the `CsrfToken` in a cookie. By default the `CookieCsrfTokenRepository` will write to a cookie named `XSRF-TOKEN` and

read it from a header named `X-XSRF-TOKEN` or the HTTP parameter `_csrf`. These defaults come from AngularJS

You can configure `CookieCsrfTokenRepository` in XML using the following:

```
<http>
        <!-- ... -->
        <csrf token-repository-ref="tokenRepository"/>
</http>
<b:bean id="tokenRepository"
        class="org.springframework.security.web.csrf.CookieCsrfTokenRepository"
        p:cookieHttpOnly="false"/>
```

> The sample explicitly sets `cookieHttpOnly=false`. This is necessary to allow JavaScript (i.e. AngularJS) to read it. If you do not need the ability to read the cookie with JavaScript directly, it is recommended to omit `cookieHttpOnly=false` to improve security.

You can configure `CookieCsrfTokenRepository` in Java Configuration using:

```
@EnableWebSecurity
public class WebSecurityConfig extends
                WebSecurityConfigurerAdapter {

        @Override
        protected void configure(HttpSecurity http) throws Exception {
                http
                        .csrf()

        .csrfTokenRepository(CookieCsrfTokenRepository.withHttpOnlyFalse());
        }
}
```

> The sample explicitly sets `cookieHttpOnly=false`. This is necessary to allow JavaScript (i.e. AngularJS) to read it. If you do not need the ability to read the cookie with JavaScript directly, it is recommended to omit `cookieHttpOnly=false` (by using `new CookieCsrfTokenRepository()` instead) to improve security.

19.5 CSRF Caveats

There are a few caveats when implementing CSRF.

19.5.1 Timeouts

One issue is that the expected CSRF token is stored in the HttpSession, so as soon as the HttpSession expires your configured `AccessDeniedHandler` will receive a InvalidCsrfTokenException. If you are using the default `AccessDeniedHandler`, the browser will get an HTTP 403 and display a poor error message.

> One might ask why the expected `CsrfToken` isn't stored in a cookie by default. This is because there are known exploits in which headers (i.e. specify the cookies) can be set by another domain. This is the same reason Ruby on Rails no longer skips CSRF checks when the header X-Requested-With is present. See this webappsec.org thread for details on how to perform the exploit. Another disadvantage is that by removing the state (i.e. the timeout) you lose the ability to forcibly terminate the token if it is compromised.

A simple way to mitigate an active user experiencing a timeout is to have some JavaScript that lets the user know their session is about to expire. The user can click a button to continue and refresh the session.

Alternatively, specifying a custom `AccessDeniedHandler` allows you to process the `InvalidCsrfTokenException` any way you like. For an example of how to customize the `AccessDeniedHandler` refer to the provided links for both xml and Java configuration.

Finally, the application can be configured to use CookieCsrfTokenRepository which will not expire. As previously mentioned, this is not as secure as using a session, but in many cases can be good enough.

1.5.2 Logging In

In order to protect against forging log in requests the log in form should be protected against CSRF attacks too. Since the `CsrfToken` is stored in HttpSession, this means an HttpSession will be created as soon as `CsrfToken` token attribute is accessed. While this sounds bad in a RESTful / stateless architecture the reality is that state is necessary to implement practical security. Without state, we have nothing we can do if a token is compromised. Practically speaking, the CSRF token is quite small in size and should have a negligible impact on our architecture.

A common technique to protect the log in form is by using a JavaScript function to obtain a valid CSRF token before the form submission. By doing this, there is no need to think about session timeouts (discussed in the previous section) because the session is created right before the form submission (assuming that CookieCsrfTokenRepository isn't configured instead), so the user can stay on the login page and submit the username/password when he wants. In order to achieve this, you can take advantage of the `CsrfTokenArgumentResolver` provided by Spring Security and expose an endpoint like it's described on here.

1.5.3 Logging Out

Adding CSRF will update the LogoutFilter to only use HTTP POST. This ensures that log out requires a CSRF token and that a malicious user cannot forcibly log out your users.

One approach is to use a form for log out. If you really want a link, you can use JavaScript to have the link perform a POST (i.e. maybe on a hidden form). For browsers with JavaScript that is disabled, you can optionally have the link take the user to a log out confirmation page that will perform the POST.

If you really want to use HTTP GET with logout you can do so, but remember this is generally not recommended. For example, the following Java Configuration will perform logout with the URL /logout is requested with any HTTP method:

```
@EnableWebSecurity
public class WebSecurityConfig extends
WebSecurityConfigurerAdapter {

        @Override
        protected void configure(HttpSecurity http) throws Exception {
                http
                        .logout()
                                .logoutRequestMatcher(new
AntPathRequestMatcher("/logout"));
        }
}
```

1.5.4 Multipart (file upload)

There are two options to using CSRF protection with multipart/form-data. Each option has its tradeoffs.

- **Placing MultipartFilter before Spring Security**
- **Include CSRF token in action**

Before you integrate Spring Security's CSRF protection with multipart file upload, ensure that you can upload without the CSRF protection first. More information about using multipart forms with Spring can be found within the 17.10 Spring's multipart (file upload) support section of the Spring reference and the MultipartFilter javadoc.

Placing MultipartFilter before Spring Security

The first option is to ensure that the `MultipartFilter` is specified before the Spring Security filter. Specifying the `MultipartFilter` before the Spring Security filter means that there is no authorization for invoking the `MultipartFilter` which means anyone can place temporary files on your server. However, only authorized users will be able to submit a File that is processed by your application. In general, this is the recommended approach because the temporary file upload should have a negligble impact on most servers.

To ensure `MultipartFilter` is specified before the Spring Security filter with java configuration, users can override beforeSpringSecurityFilterChain as shown below:

```
public class SecurityApplicationInitializer extends
AbstractSecurityWebApplicationInitializer {

	@Override
	protected void beforeSpringSecurityFilterChain(ServletContext
servletContext) {
		insertFilters(servletContext, new MultipartFilter());
	}
}
```

To ensure `MultipartFilter` is specified before the Spring Security filter with XML configuration, users can ensure the <filter-mapping> element of the `MultipartFilter` is placed before the springSecurityFilterChain within the web.xml as shown below:

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

Include CSRF token in action

If allowing unauthorized users to upload temporariy files is not acceptable, an alternative is to place the `MultipartFilter` after the Spring Security filter and include the CSRF as a query parameter in the action attribute of the form. An example with a jsp is shown below

```
<form action="./upload?${_csrf.parameterName}=${_csrf.token}" method="post"
enctype="multipart/form-data">
```

The disadvantage to this approach is that query parameters can be leaked. More genearlly, it is considered best practice to place sensitive data within the body or headers to ensure it is not leaked. Additional information can be found in RFC 2616 Section 15.1.3 Encoding Sensitive Information in URI's.

1.5.5 HiddenHttpMethodFilter

The HiddenHttpMethodFilter should be placed before the Spring Security filter. In general this is true, but it could have additional implications when protecting against CSRF attacks.

Note that the HiddenHttpMethodFilter only overrides the HTTP method on a POST, so this is actually unlikely to cause any real problems. However, it is still best practice to ensure it is placed before Spring Security's filters.

1.6 Overriding Defaults

Spring Security's goal is to provide defaults that protect your users from exploits. This does not mean that you are forced to accept all of its defaults.

For example, you can provide a custom CsrfTokenRepository to override the way in which the `CsrfToken` is stored.

You can also specify a custom RequestMatcher to determine which requests are protected by CSRF (i.e. perhaps you don't care if log out is exploited). In short, if Spring Security's CSRF protection doesn't behave exactly as you want it, you are able to customize the behavior. Refer to the Section 43.1.18, "<csrf>" documentation for details on how to make these customizations with XML and the `CsrfConfigurer` javadoc for details on how to make these customizations when using Java configuration.
