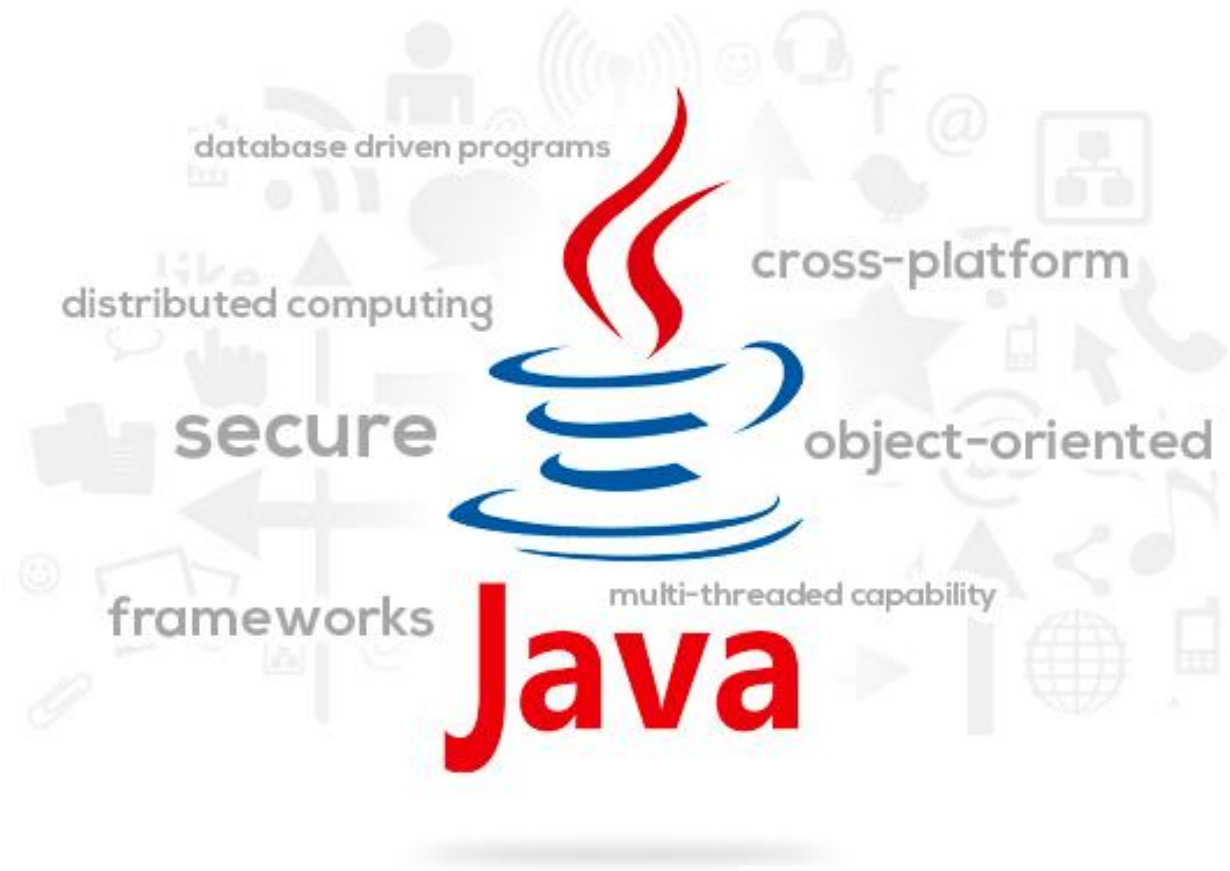


Yadab Raj Ojha

Sr. Java Developer / Lecture

Email: yadabrajojha@gmail.com
Blog: http://yro-tech.blogspot.com/
Java Tutorial: https://github.com/yrojha4ever/JavaStud
LinkedIn: https://www.linkedin.com/in/yrojha
Twitter: https://twitter.com/yrojha4ever
Website: http://javaenvagilist.com/

Spring Framework

- Introduction.
- What is spring framework?
- Advantage of using Spring Framework.
- What spring do.
- Spring modules
- Dependency Injection/Ioc Container
- Bean Autowiring
- Bean Scope /Component Scan
- Annotations(Required, Autowired, Service, Component, Repository, Resource)
- Spring JDBCTemplate
- ORM(Integrate Hibernate with Spring)
- Spring WEB MVC/Transaction Management(@Transaction), Component Scan, View Resolver
- MVC Annotations(Controller, RequestMapping, PathVariable, ModelAttribute)
- Redirect, Multipart file upload, Exceptions
- Spring Form tag library
- Spring Rest

Spring Framework

What Is Spring?

Spring is a lightweight dependency injection(IoC) and aspect-oriented(AOP) container and framework.

- **Lightweight:** Spring is lightweight in terms of both size (single JAR file) and processing overhead(negligible).
- **Dependency Injection(IoC):** Spring promotes loose coupling through a technique known as dependency injection (DI).
- **Aspect oriented (AOP) :** Spring supports Aspect oriented programming and enables cohesive development by separating application business logic from system services(transaction management , logging etc).
- **Container :** Spring contains and manages the life cycle and configuration of application objects.
- **Framework :** Spring provides most of the infrastructural functionality (transaction management, persistence framework integration, etc.), **leaving rest of the coding to the developer.**

Advantages of Spring Framework

1) Predefined Templates

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

2) Loose Coupling

The Spring applications are loosely coupled because of dependency injection.

3) Easy to test

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

4) Lightweight

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

5) Fast Development

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

6) Powerful abstraction

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

7) Declarative support

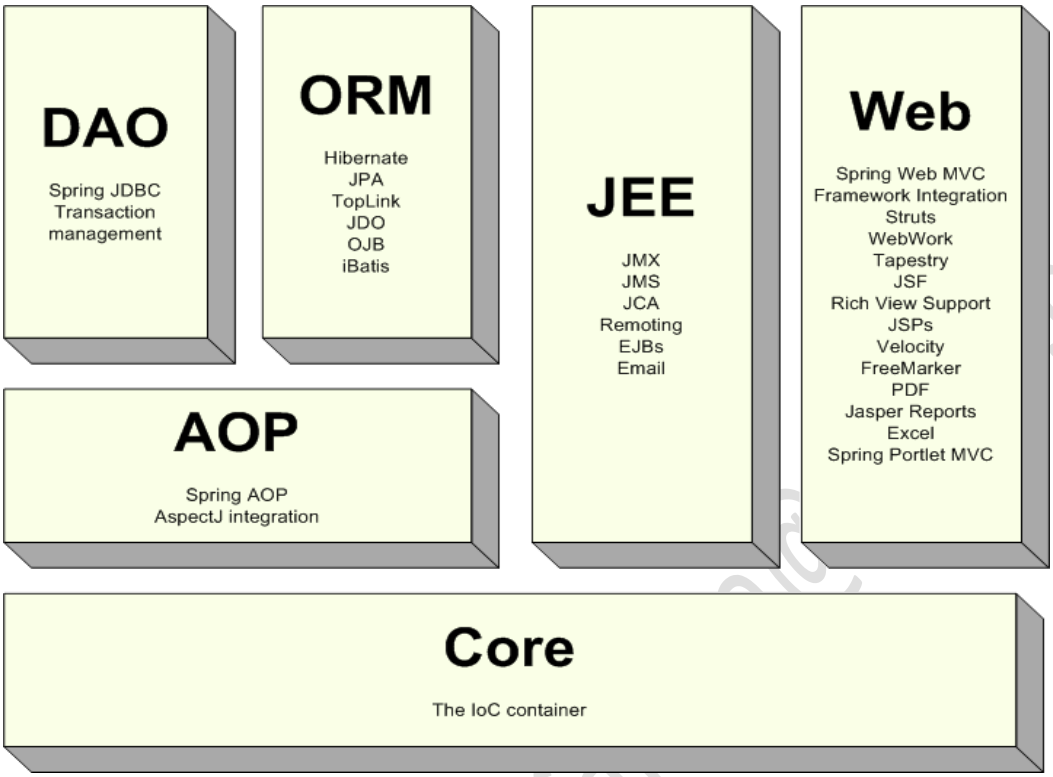
It provides declarative support for caching, validation, transactions and formatting.

What does Spring do?

- Spring's main aim is to **make J2EE easier to use** and promote good programming practice(programming to interfaces, rather than classes). POJO-based programming model!
- Spring Framework does not impose any specific programming model, **it is an alternative to**, replacement for, or even addition to the **Enterprise JavaBean (EJB) model**.
- It makes **Enterprise Application(EE) development Easier**.
- It **help to maintain System Services** such as **logging**, Information Security, **Transaction Processing** & management, Error Detection &Correction(**i.e cross-cutting concerns of program)**
- Spring is **portable between application servers** (support users on WebLogic, **Tomcat**, Resin, JBoss, Jetty, Geronimo, WebSphere and other application servers)

Spring modules (Components)

The Spring Framework is composed of several well-defined modules built on top of the core container.



Dependency Injection(IoC)

Don't call me, I'll call you.....

- **DI** moves the responsibility for **making things happen into the framework**, and away from application code.
- Whereas your code calls a traditional class library, an **IoC framework calls your code**.

Ex: Traditionally: Component might call the container to say "where's object X, which I need to do my work",

With Dependency Injection the container figures out that the component needs an X object, and provides it to it at runtime.

The container figuring out based on method signatures(**JavaBean** properties or **constructors**) & configuration data such as **XML**

Key Benefit of Dependency Injection!

- **Reduced Dependencies(Loose Coupling)**
- **Reduced Dependency Carrying**
- **More Reusable Code**
- **More Testable Code**
- **More Readable Code**

If dependency injection is used then

The class B is given to class A via:

- **The constructor of the class A - this is then called construction injection**
- **A setter Method - this is then called setter injection**

  Ex: Come Later in Bean Wiring?

Spring IoC container

IoC is also known as *dependency injection* (DI). It is a process whereby objects define their dependencies. The container then *injects* those dependencies when it creates the bean. This process is fundamentally the inverse, hence the name *Inversion of Control* (IoC), of the bean itself controlling the instantiation or location of its dependencies by using direct construction of classes, or a mechanism such as the *Service Locator* pattern.

The `org.springframework.beans` and `org.springframework.context` packages are the basis for Spring Framework's IoC container. In short, the `BeanFactory` provides the configuration framework and basic functionality, and the `ApplicationContext` adds more enterprise-specific functionality.

In Spring, the objects that form the backbone of your application and that are managed by the Spring IoC *container* are called *beans*. A bean is an object that is instantiated, assembled, and otherwise managed by a Spring IoC container. Otherwise, a bean is simply one of many objects in your

application. Beans, and the*dependencies* among them, are reflected in the *configuration metadata* used by a container.

Several implementations of the `ApplicationContext` interface are supplied out-of-the-box with Spring. In standalone applications it is common to create an instance of`ClassPathXmlApplicationContext` or `FileSystemXmlApplicationContext`.

The following example shows the basic structure of XML-based configuration metadata:

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

  <bean id="..." class="...">
    <!-- collaborators and configuration for this bean go here -->
  </bean>

  <bean id="..." class="...">
    <!-- collaborators and configuration for this bean go here -->
  </bean>

  <!-- more bean definitions go here -->

</beans>
```

Instantiating a container

```
// create and configure beans
ApplicationContext context =
    new ClassPathXmlApplicationContext(new String[] {"services.xml",
"daos.xml"});

// retrieve configured instance
PetStoreServiceImpl service = context.getBean("petStore",
PetStoreServiceImpl.class);

// use configured instance
List userList = service.getUsernameList();
```

Bean overview

Table 5.1. The bean definition

Property	Detail
class	“Instantiating beans”
name	“Naming beans”
scope	“Bean scopes”
constructor arguments	“Dependency injection”
properties	“Dependency injection”
autowiring mode	“Autowiring collaborators”
lazy-initialization mode	“Lazy-initialized beans”
initialization method	“Initialization callbacks”
destruction method	“Destruction callbacks”

References to other beans (collaborators)

```
<ref bean="someBean"/>
```

Using `depends-on`: If a bean is a dependency of another that usually means that one bean is set as a property of another.

```
<bean id="beanOne" class="ExampleBean" depends-on="manager"/>

<bean id="manager" class="ManagerBean" />
```

Lazy-initialized beans:

A lazy-initialized bean tells the IoC container to create a bean instance when it is first requested, rather than at startup.

By default, `ApplicationContext` implementations eagerly create and configure all singleton beans as part of the initialization process.

```
<bean id="lazy" class="com.foo.ExpensiveToCreateBean" lazy-init="true"/>

<bean name="not.lazy" class="com.foo.AnotherBean"/>
```

Autowiring(`autowire, @Autowired`)

Autowiring modes

Mode	Explanation
no	(Default) No autowiring. Bean references must be defined via a `ref` element. Changing the default setting is not recommended for larger deployments, because specifying collaborators explicitly gives greater control and clarity. To some extent, it documents the structure of a system.
byName	Autowiring by property name. Spring looks for a bean with the same name as the property that needs to be autowired. For example, if a bean definition is set to autowire by name, and it contains a *master* property (that is, it has a *setMaster(..)* method), Spring looks for a bean definition named `master`, and uses it to set the property.
byType	Allows a property to be autowired if exactly one bean of the property type exists in the container. If more than one exists, a fatal exception is thrown, which indicates that you may not use *byType* autowiring for that bean. If there are no matching beans, nothing happens; the property is not set.
constructor	Analogous to *byType*, but applies to constructor arguments. If there is not exactly one bean of the constructor argument type in the container, a fatal error is raised.

The Spring container can *autowire* relationships between collaborating beans. You can allow Spring to resolve collaborators (other beans) automatically for your bean by inspecting the contents of the `ApplicationContext`.

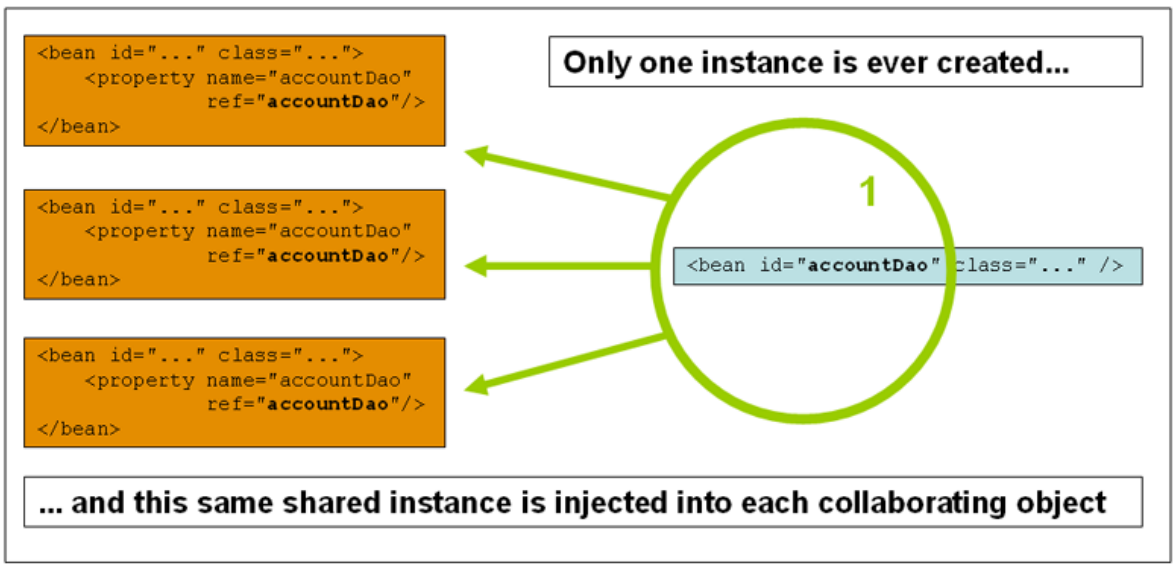
Bean scopes

Table 5.3. Bean scopes

Scope	Description
singleton	(Default) Scopes a single bean definition to a single object instance per Spring IoC container.
prototype	Scopes a single bean definition to any number of object instances.
request	Scopes a single bean definition to the lifecycle of a single HTTP request; that is, each HTTP request has its own instance of a bean created off the back of a single bean definition. Only valid in the context of a web-aware Spring `ApplicationContext`.
session	Scopes a single bean definition to the lifecycle of an HTTP `Session`. Only valid in the context of a web-aware Spring `ApplicationContext`.
global session	Scopes a single bean definition to the lifecycle of a global HTTP `Session`. Typically only valid when used in a portlet context. Only valid in the context of a web-aware Spring `ApplicationContext`.

The singleton scope

```
<bean id="accountService" class="com.foo.DefaultAccountService"
scope="singleton"/>
```



Annotation-based container configuration

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

   <context:annotation-config/>

</beans>
```

@Required

```
@Required
private MovieFinder movieFinder;
```

@Autowired

```
@Autowired(required=false)
private MovieCatalog movieCatalog;
```

@Resource

```
@Resource
private SessionFactory sessionFactory;
```


Classpath scanning and managed components:

Automatically detecting classes and registering bean definitions

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

```
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context.xsd">

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

</beans>
```

In Spring 2.0 and later, the `@Repository` annotation is a marker for any class that fulfills the role or *stereotype* (also known as Data Access Object or DAO) of a repository. Among the uses of this marker is the automatic translation of exceptions.

Spring 2.5 introduces further stereotype annotations: `@Component`, `@Service`, and `@Controller`. `@Component` is a generic stereotype for any Spring-managed component. `@Repository`, `@Service`, and `@Controller` are specializations of `@Component` for more specific use cases, for example, in the persistence, service, and presentation layers, respectively. Therefore, you can annotate your component classes with`@Component`, but by annotating them with `@Repository`, `@Service`, or `@Controller` instead, your classes are more properly suited for processing by tools or associating with aspects. For example, these stereotype annotations make ideal targets for pointcuts. It is also possible that `@Repository`, `@Service`, and `@Controller` may carry additional semantics in future releases of the Spring Framework. Thus, if you are choosing between using `@Component` or `@Service` for your service layer, `@Service` is clearly the better choice. Similarly, as stated above,`@Repository` is already supported as a marker for automatic exception translation in your persistence layer.


```
@Service
public class SimpleMovieLister {

  private MovieFinder movieFinder;

  @Autowired
  public SimpleMovieLister(MovieFinder movieFinder) {
      this.movieFinder = movieFinder;
  }
```

```
}
@Repository
public class JpaMovieFinder implements MovieFinder {
  // implementation elided for clarity
}
```

Spring Framework JDBC

`JdbcTemplate`

The `JdbcTemplate` class is the central class in the JDBC core package. It handles the creation and release of resources, which helps you avoid common errors such as forgetting to close the connection. It performs the basic tasks of the core JDBC workflow such as statement creation and execution, leaving application code to provide SQL and extract results. The `JdbcTemplate` class executes SQL queries, update statements and stored procedure calls, performs iteration over `ResultSet`s and extraction of 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.

When you use the `JdbcTemplate` for your code, you only need to implement callback interfaces, giving them a clearly defined contract. The`PreparedStatementCreator` callback interface creates a prepared statement given a `Connection` provided by this class, providing SQL and any necessary parameters. The same is true for the `CallableStatementCreator` interface, which creates callable statements. The`RowCallbackHandler` interface extracts values from each row of a `ResultSet`.

The `JdbcTemplate` can be used within a DAO implementation through direct instantiation with a `DataSource` reference, or be configured in a Spring IoC container and given to DAOs as a bean reference.

```
int rowCount = this.jdbcTemplate.queryForObject("select count(*) from t_actor", int.class);
```

Querying for a `String`:

```
String lastName = this.jdbcTemplate.queryForObject(
        "select last_name from t_actor where id = ?",
        new Object[]{1212L}, String.class);
```

Querying and populating a *single* domain object:

```
Actor actor = this.jdbcTemplate.queryForObject(
        "select first_name, last_name from t_actor where id = ?",
        new Object[]{1212L},
        new RowMapper<Actor>() {
            public Actor mapRow(ResultSet rs, int rowNum) throws SQLException {
                Actor actor = new Actor();
                actor.setFirstName(rs.getString("first_name"));
                actor.setLastName(rs.getString("last_name"));
                return actor;
            }
        });
```

```
public List<Actor> findAllActors() {
    return this.jdbcTemplate.query( "select first_name, last_name from t_actor", new ActorMapper());
}

private static final class ActorMapper implements RowMapper<Actor> {

    public Actor mapRow(ResultSet rs, int rowNum) throws SQLException {
        Actor actor = new Actor();
        actor.setFirstName(rs.getString("first_name"));
        actor.setLastName(rs.getString("last_name"));
        return actor;
    }
}
```

Updating (INSERT/UPDATE/DELETE) with jdbcTemplate

You use the `update(..)` method to perform insert, update and delete operations. Parameter values are usually provided as var args or alternatively as an object array.

```
this.jdbcTemplate.update(
        "insert into t_actor (first_name, last_name) values (?, ?)",
        "Leonor", "Watling");
this.jdbcTemplate.update(
```

```
        "update t_actor set last_name = ? where id = ?",
        "Banjo", 5276L);
this.jdbcTemplate.update(
        "delete from actor where id = ?",
        Long.valueOf(actorId));
```

ORM Data Access(Hibernate):

`SessionFactory` setup in a Spring container

To avoid tying application objects to hard-coded resource lookups, you can define resources such as a JDBC `DataSource` or a Hibernate`SessionFactory` as beans in the Spring container. Application objects that need to access resources receive references to such predefined instances through bean references, as illustrated in the DAO definition in the next section.

The following excerpt from an XML application context definition shows how to set up a JDBC `DataSource` and a Hibernate `SessionFactory` on top of it:

```
<beans>

  <bean id="myDataSource" class="org.apache.commons.dbcp.BasicDataSource"
destroy-method="close">
    <property name="driverClassName" value="org.hsqldb.jdbcDriver"/>
    <property name="url" value="jdbc:hsqldb:hsql://localhost:9001"/>
    <property name="username" value="sa"/>
    <property name="password" value=""/>
  </bean>

  <bean id="mySessionFactory"
class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
    <property name="dataSource" ref="myDataSource"/>
    <property name="mappingResources">
      <list>
        <value>product.hbm.xml</value>
      </list>
    </property>
    <property name="hibernateProperties">
      <value>
        hibernate.dialect=org.hibernate.dialect.HSQLDialect
      </value>
    </property>
```

```
  </bean>

</beans>
```

Declarative transaction Management:

```
@Transactional(readOnly = true)
    public List<Product> findAllProducts() {
        return this.productDao.findAllProducts();
    }
```

All you need to provide is the TransactionManager implementation and a "<tx:annotation-driven/>" entry.

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

  <!-- SessionFactory, DataSource, etc. omitted -->

  <bean id="transactionManager"

class="org.springframework.orm.hibernate3.HibernateTransactionManager">
    <property name="sessionFactory" ref="sessionFactory"/>
  </bean>

  <tx:annotation-driven/>

  <bean id="myProductService" class="product.SimpleProductService">
    <property name="productDao" ref="myProductDao"/>
  </bean>

</beans>
```

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.

Features of Spring Web MVC

Spring's web module includes many unique web support features:

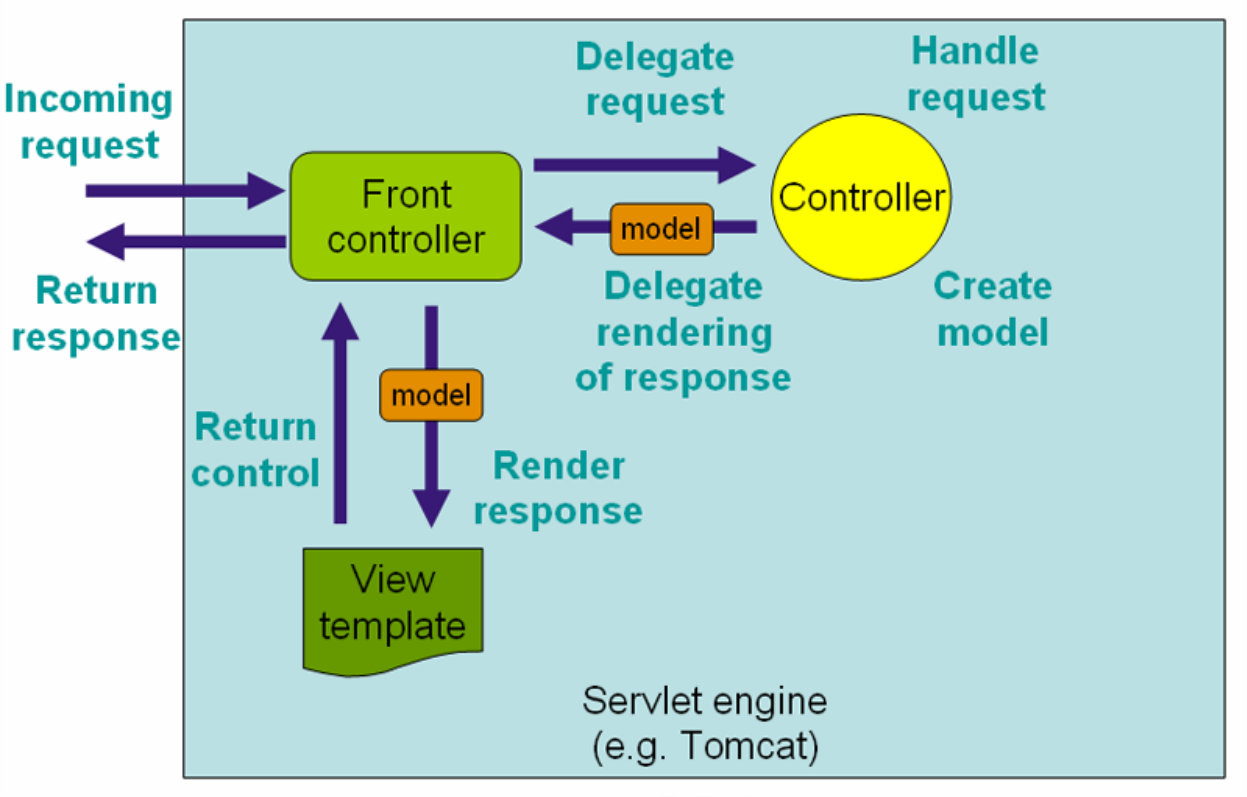
- ***Clear separation of roles***. Each role — controller, validator, command object, form object, model object, `DispatcherServlet`, handler mapping, view resolver, and so on — can be fulfilled by a specialized object.
- ***Powerful and straightforward configuration of both framework and application classes as JavaBeans*.** This configuration capability includes easy referencing across contexts, such as from web controllers to business objects and validators.
- ***Adaptability, non-intrusiveness, and flexibility***. Define any controller method signature you need, possibly using one of the parameter annotations (such as @RequestParam, @RequestHeader, @PathVariable, and more) for a given scenario.
- ***Reusable business code*, *no need for duplication***. Use existing business objects as command or form objects instead of mirroring them to extend a particular framework base class.
- ***Customizable binding and validation***. Type mismatches as application-level validation errors that keep the offending value, localized date and number binding, and so on instead of String-only form objects with manual parsing and conversion to business objects.

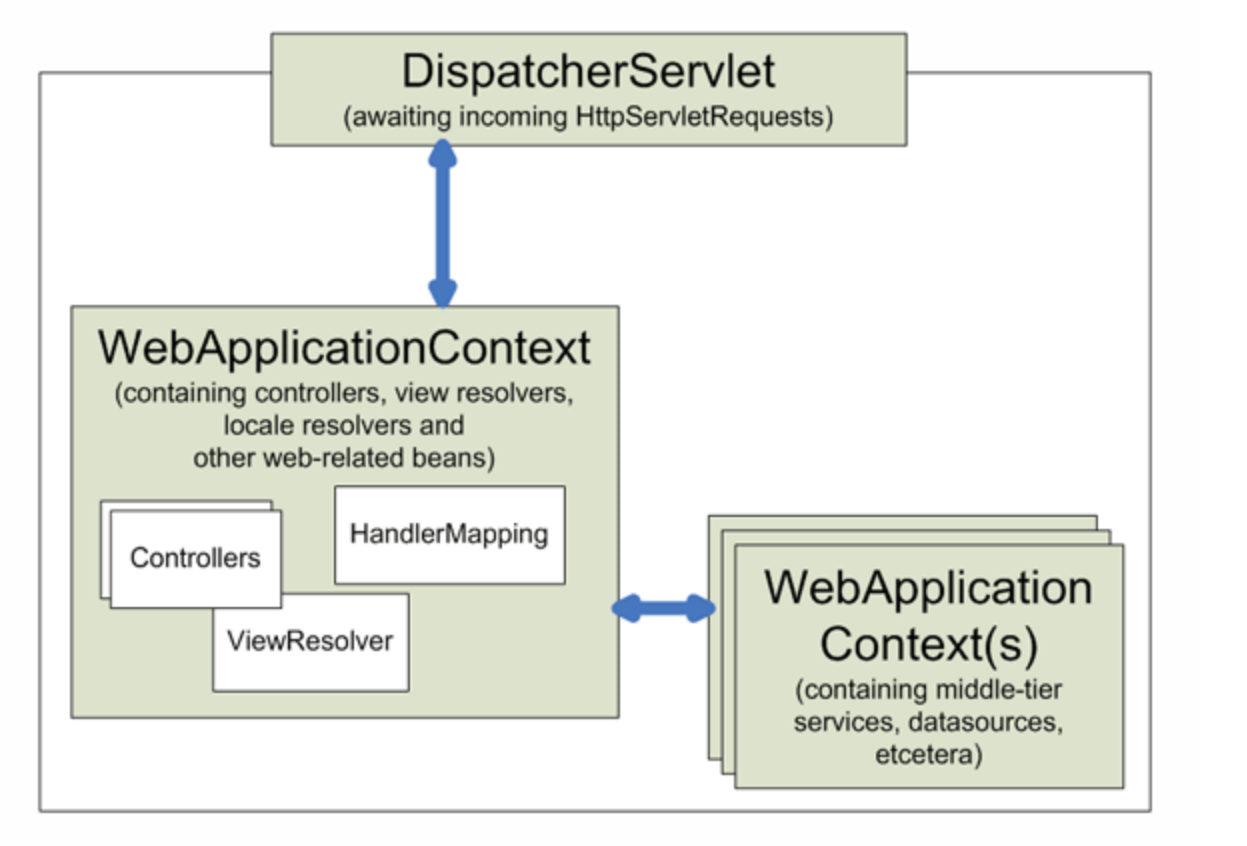
- ***Customizable handler mapping and view resolution***. Handler mapping and view resolution strategies range from simple URL-based configuration, to sophisticated, purpose-built resolution strategies. Spring is more flexible than web MVC frameworks that mandate a particular technique.
- ***Flexible model transfer***. Model transfer with a name/value `Map` supports easy integration with any view technology.
- ***Customizable locale and theme resolution, support for JSPs with or without Spring tag library, support for JSTL, support for Velocity without the need for extra bridges, and so on.***
- ***A simple yet powerful JSP tag library known as the Spring tag library*** *that provides support for features such as data binding and themes*. The custom tags allow for maximum flexibility in terms of markup code. For information on the tag library descriptor, see the appendix entitled Appendix G, *spring.tld*
- *A JSP form tag library, introduced in Spring 2.0, that makes writing forms in JSP pages much easier.* For information on the tag library descriptor, see the appendix entitled Appendix H, *spring-form.tld*
- ***Beans*** *whose lifecycle is scoped to the current HTTP request or HTTP* `Session`*.* This is not a specific feature of Spring MVC itself, but rather of the `WebApplicationContext` container(s) that Spring MVC uses. These bean scopes are described in Section 5.5.4, "Request, session, and global session scopes"

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.

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


Incoming request
Return response
Front controller
Delegate request
Handle request
Controller
model
Delegate rendering of response
Create model
Return control
model
Render response
View template
Servlet engine (e.g. Tomcat)


DispatcherServlet
(awaiting incoming HttpServletRequests)
WebApplicationContext
(containing controllers, view resolvers, locale resolvers and other web-related beans)
Controllers
HandlerMapping
ViewResolver
WebApplication Context(s)
(containing middle-tier services, datasources, etcetera)

```
<web-app>

    <servlet>
        <servlet-name>example</servlet-name>
        <servlet-
class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>

    <servlet-mapping>
        <servlet-name>example</servlet-name>
        <url-pattern>/example/*</url-pattern>
    </servlet-mapping>

</web-app
```

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.

Implementing Controllers

```
@Controller
public class HelloWorldController {

    @RequestMapping("/helloWorld")
    public String helloWorld(Model model) {
        model.addAttribute("message", "Hello World!");
        return "helloWorld";
    }
}
```

```
<?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"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context.xsd">

    <context:component-scan base-
package="org.springframework.samples.petclinic.web"/>

    <!-- ... -->

</beans>
```

Mapping Requests With `@RequestMapping`

```
@Controller
@RequestMapping("/appointments")
public class AppointmentsController {

    private final AppointmentBook appointmentBook;

    @Autowired
    public AppointmentsController(AppointmentBook appointmentBook) {
        this.appointmentBook = appointmentBook;
    }

    @RequestMapping(method = RequestMethod.GET)
    public Map<String, Appointment> get() {
        return appointmentBook.getAppointmentsForToday();
    }

    @RequestMapping(value="/{day}", method = RequestMethod.GET)
```

```
    public Map<String, Appointment> getForDay(@PathVariable
@DateTimeFormat(iso=ISO.DATE) Date day, Model model) {
        return appointmentBook.getAppointmentsForDay(day);
    }

    @RequestMapping(value="/new", method = RequestMethod.GET)
    public AppointmentForm getNewForm() {
        return new AppointmentForm();
    }

    @RequestMapping(method = RequestMethod.POST)
    public String add(@Valid AppointmentForm appointment, BindingResult
result) {
        if (result.hasErrors()) {
            return "appointments/new";
        }
        appointmentBook.addAppointment(appointment);
        return "redirect:/appointments";
    }
}
```

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
  }
}
```

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);
}
```

Resolving views

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

Redirecting to views

```
@RequestMapping(value = "/files/{path}", method = RequestMethod.POST)
public String upload(...) {
    // ...
    return "redirect:files/{path}";
}
```

The `redirect:` prefix

A logical view name such as `redirect:/myapp/some/resource` will redirect relative to the current Servlet context, while a name such as `redirect:http://myhost.com/some/arbitrary/path` will redirect to an absolute URL.

The `forward:` prefix(`RequestDispatcher.forward()`)

As with the `redirect:` prefix, if the view name with the `forward:` prefix is injected into the controller, the controller does not detect that anything special is happening in terms of handling the response.

Spring's multipart (file upload) support

Spring's built-in multipart support handles file uploads in web applications. You enable this multipart support with pluggable`MultipartResolver` objects, defined in the `org.springframework.web.multipart` package. Spring provides one `MultipartResolver`implementation for use with *Commons FileUpload* and another for use with Servlet 3.0 multipart request parsing.

Using a `MultipartResolver` with *Commons FileUpload*

The following example shows how to use the `CommonsMultipartResolver`:

```
<bean id="multipartResolver"

class="org.springframework.web.multipart.commons.CommonsMultipartResolver">

    <!-- one of the properties available; the maximum file size in bytes -->
    <property name="maxUploadSize" value="100000"/>
</bean>
```

Handling a file upload in a form

After the `MultipartResolver` completes its job, the request is processed like any other. First, create a form with a file input that will allow the user to upload a form. The encoding attribute (`enctype="multipart/form-data"`) lets the browser know how to encode the form as multipart request:

```
<html>
    <head>
```

```
        <title>Upload a file please</title>
    </head>
    <body>
        <h1>Please upload a file</h1>
        <form method="post" action="/form" enctype="multipart/form-data">
            <input type="text" name="name"/>
            <input type="file" name="file"/>
            <input type="submit"/>
        </form>
    </body>
</html>
```

The next step is to create a controller that handles the file upload. This controller is very similar to a normal annotated `@Controller`, except that we use `MultipartHttpServletRequest` or `MultipartFile` in the method parameters:

```
@Controller
public class FileUploadController {

    @RequestMapping(value = "/form", method = RequestMethod.POST)
    public String handleFormUpload(@RequestParam("name") String name,
        @RequestParam("file") MultipartFile file) {

        if (!file.isEmpty()) {
            byte[] bytes = file.getBytes();
            // store the bytes somewhere
           return "redirect:uploadSuccess";
       } else {
           return "redirect:uploadFailure";
       }
    }

}
```

Handling Standard Spring MVC Exceptions

Spring MVC may raise a number of exceptions while processing a request. The `SimpleMappingExceptionResolver` can easily map any exception to a default error view as needed. However, when working with clients that interpret responses in an automated way you will want to set specific status code on

the response. Depending on the exception raised the status code may indicate a client error (4xx) or a server error (5xx).

The `DefaultHandlerExceptionResolver` translates Spring MVC exceptions to specific error status codes. It is registered by default with the MVC namespace, the MVC Java config, and also by the the `DispatcherServlet` (i.e. when not using the MVC namespace or Java config). Listed below are some of the exceptions handled by this resolver and the corresponding status codes:

Exception	HTTP Status Code
`BindException`	400 (Bad Request)
`ConversionNotSupportedException`	500 (Internal Server Error)
`HttpMediaTypeNotAcceptableException`	406 (Not Acceptable)
`HttpMediaTypeNotSupportedException`	415 (Unsupported Media Type)
`HttpMessageNotReadableException`	400 (Bad Request)
`HttpMessageNotWritableException`	500 (Internal Server Error)
`HttpRequestMethodNotSupportedException`	405 (Method Not Allowed)
`MethodArgumentNotValidException`	400 (Bad Request)
`MissingServletRequestParameterException`	400 (Bad Request)
`MissingServletRequestPartException`	400 (Bad Request)
`NoSuchRequestHandlingMethodException`	404 (Not Found)
`TypeMismatchException`	400 (Bad Request)

`@ExceptionHandler`

```
@Controller
public class SimpleController {


  // @RequestMapping methods omitted ...


  @ExceptionHandler(IOException.class)
  public ResponseEntity<String> handleIOException(IOException ex) {

    // prepare responseEntity

    return responseEntity;
  }

}
```

Configuring View Controllers

```
<mvc:view-controller path="/" view-name="home"/>
```

Configuring Serving of Resources

```
<mvc:resources mapping="/resources/**" location="/public-resources/"/>
```

```
<mvc:resources mapping="/resources/**" location="/public-resources/" cache-
period="31556926"/>
```


Spring's form tag library

Configuration

The form tag library comes bundled in `spring-webmvc.jar`. The library descriptor is called `spring-form.tld`.

To use the tags from this library, add the following directive to the top of your JSP page:

```
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form" %>
```

The `form` tag

This tag renders an HTML 'form' tag and exposes a binding path to inner tags for binding. It puts the command object in the `PageContext` so that the command object can be accessed by inner tags. *All the other tags in this library are nested tags of the `form` tag*.

Let's assume we have a domain object called `User`. It is a JavaBean with properties such as `firstName` and `lastName`. We will use it as the form backing object of our form controller which returns `form.jsp`. Below is an example of what `form.jsp` would look like:

```
<form:form>
      <table>
          <tr>
              <td>First Name:</td>
              <td><form:input path="firstName" /></td>
          </tr>
          <tr>
              <td>Last Name:</td>
              <td><form:input path="lastName" /></td>
          </tr>
          <tr>
              <td colspan="2">
                  <input type="submit" value="Save Changes" />
              </td>
          </tr>
      </table>
  </form:form>
```

The `firstName` and `lastName` values are retrieved from the command object placed in the `PageContext` by the page controller. Keep reading to see more complex examples of how inner tags are used with the `form` tag.

The generated HTML looks like a standard form:

```
<form method="POST">
      <table>
        <tr>
            <td>First Name:</td>
            <td><input name="firstName" type="text" value="Harry"/></td>
        </tr>
        <tr>
            <td>Last Name:</td>
            <td><input name="lastName" type="text" value="Potter"/></td>
        </tr>
        <tr>
            <td colspan="2">
              <input type="submit" value="Save Changes" />
            </td>
        </tr>
      </table>
  </form>
```

The preceding JSP assumes that the variable name of the form backing object is `'command'`. If you have put the form backing object into the model under another name (definitely a best practice), then you can bind the form to the named variable like so:

```
<form:form commandName="user">
      <table>
          <tr>
              <td>First Name:</td>
              <td><form:input path="firstName" /></td>
          </tr>
          <tr>
              <td>Last Name:</td>
              <td><form:input path="lastName" /></td>
          </tr>
          <tr>
              <td colspan="2">
                  <input type="submit" value="Save Changes" />
              </td>
          </tr>
      </table>
  </form:form>
```

OR you can use: <form:form method="POST" **modelAttribute="student"**> as well.

What is REST ?

REST stands for **RE**presentational **S**tate **T**ransfer. REST is web standards based architecture and uses HTTP Protocol for data communication. It revolves around resource where every component is a resource and a resource is accessed by a common interface using HTTP standard methods. REST was first introduced by Roy Fielding in 2000.

In REST architecture, a REST Server simply provides access to resources and REST client accesses and presents the resources. Here each resource is identified by URIs/ global IDs. REST uses various representations to represent a resource like text, JSON and XML. Now a days JSON is the most popular format being used in web services.

HTTP Methods

Following well known HTTP methods are commonly used in REST based architecture.

- **GET** - Provides a read only access to a resource.
- **PUT** - Used to create a new resource.
- **DELETE** - Used to remove a resource.
- **POST** - Used to update a existing resource or create a new resource.
- **OPTIONS** - Used to get the supported operations on a resource.

RESTFul Web Services

A web service is a collection of open protocols and standards used for exchanging data between applications or systems. Software applications written in various programming languages and running on various platforms can use web services to exchange data over computer networks like the Internet in a manner similar to inter-process communication on a single computer. This interoperability (e.g., between Java and Python, or Windows and Linux applications) is due to the use of open standards.

Web services based on REST Architecture are known as RESTful web services. These web services use HTTP methods to implement the concept of REST architecture. A RESTful web service usually defines a URI, Uniform Resource Identifier a service, provides resource representation such as JSON and set of HTTP Methods.

Creating RESTFul Web Service

This tutorial will create a web service say user management with following functionalities:

Sr. No.	HTTP Method	URI	Operation	Operation Type
1	**GET**	/UserService/users	Get list of users	Read Only
2	**GET**	/UserService/users/1	Get User with Id 1	Read Only
3	**PUT**	/UserService/users/2	Insert User with Id 2	Idempotent
4	**POST**	/UserService/users/2	Update User with Id 2	N/A
5	**DELETE**	/UserService/users/1	Delete User with Id 1	Idempotent
6	**OPTIONS**	/UserService/users	List the supported operations in web service	Read Only

Restfull services in Spring:

1. Make sure you have faster xm libraries added in pom.xml

```
<dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-core</artifactId>
   <version>2.5.4</version>
</dependency>
<dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
```

```
    <artifactId>jackson-databind</artifactId>
    <version>2.5.4</version>
</dependency>
```

2. Create Rest Type Controller:

```
@RestController
@RequestMapping("api/rest")
public class StudentRestController {

    @Autowired
    private StudentDao studentDao;

    @RequestMapping(method = RequestMethod.GET)
    public ResponseEntity<List<Student>> index() {
        ResponseEntity<List<Student>> studList = new
ResponseEntity<List<Student>>( studentDao.getAll(), HttpStatus.OK);
        return studList;
    }
}
```

3. Now you can send application/json request:

```
$.ajax({
            type : "GET",
            contentType : "application/json",
            url : "api/rest",
        }).then(function(data) {
            $('#displayJSON').html(JSON.stringify(data));
        });
```

Just make sure you have jquery :

```
<script
src="https://ajax.googleapis.com/ajax/libs/jquery/2.2.0/jquery.min.js"></script>
```