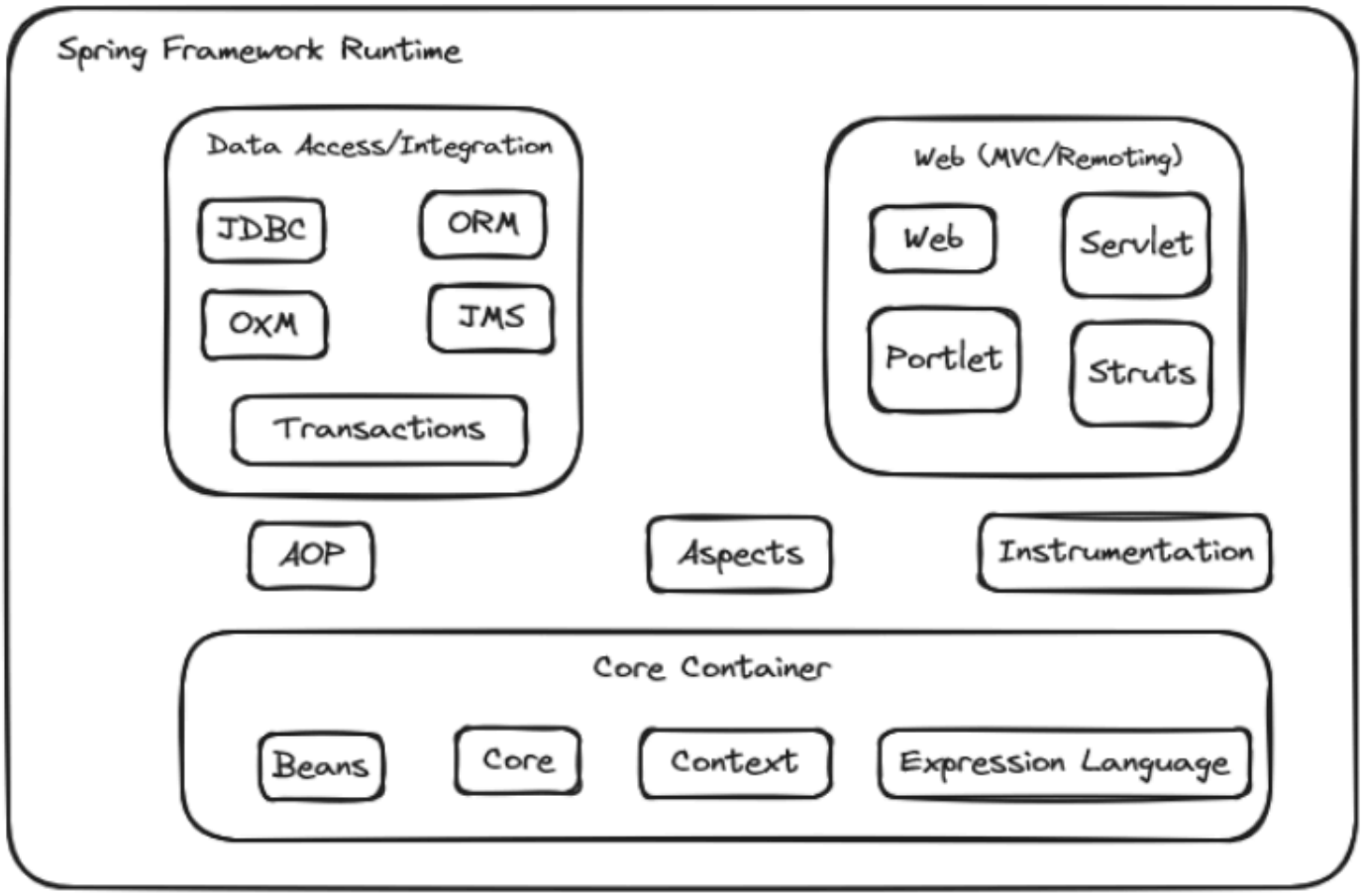
# Spring MVC Concepts

## Spring Framework Overview




### Core Container

The Core Container contains the following modules:

- **Core Module**
  - Provides IoC (Inversion of Control) and DI (Dependency Injection) features.
  - These are fundamental parts of the framework.
- **Beans Module**
  - Contains BeanFactory which helps in implementing the factory pattern.
  - Along with Core Module, it provides a strong base for building the Context module.
- **Context Module**
  - Acts as a middleman to access any defined and configured object.
  - Works as the focal point of Application Context.
- **SpEL Module**
  - Provides a powerful expression language for querying and manipulating an object graph at runtime.

## Data Access/Integration

This layer involves the following modules:

- **JDBC Module**
  - Provides a JDBC-abstraction layer, eliminating the need for extensive JDBC coding.
- **ORM Module**
  - Supports integration with popular object-relational mapping APIs like iBatis, Hibernate, JDO, and JPA.
- **OXM Module**
  - Supports Object/XML mapping implementations for APIs like XStream, JiBX, XMLBeans, Castor, and JAXB.

-**JMS Module**

- Includes features for producing and consuming messages via Java Messaging Service (JMS).
- **Transaction Module**
  - Provides programmatic and declarative transaction management for classes implementing special interfaces and all POJOs.

## Web

The Web layer includes the following modules:

- **Web Module**
  - Provides basic features for multipart file-upload functionality.
  - Initializes the IoC container using servlet listeners and a web-oriented application context.
- **Web-MVC Module**
  - Covers Spring's Model-View-Controller (MVC) implementation for web applications.
- **Web-Socket Module**
  - Supports WebSocket-based, two-way communication between the client and the server in web applications.
- **Web-Portlet Module**
  - Provides MVC implementation for portlet setups, emulating the functionality of the Web-Servlet module.

## Miscellaneous Modules

Additional important modules include:

- **AOP Module**
  - Provides an aspect-oriented programming implementation to define method-interceptors and pointcuts, decoupling code.
- **Aspects Module**
  - Supports AspectJ integration, a mature and powerful AOP Framework.
- **Instrumentation Module**
  - Supports class loader and class instrumentation, mainly for application servers.
- **Messaging Module**
  - Supports Streaming Text Oriented Messaging Protocol (STOMP) as the WebSocket sub-protocol for use in applications.

## Test Module

- Supports testing Spring Components with JUnit or TestNG frameworks.

# Hello World using Spring

1. **Create a Maven Project**
   - Open your IDE and create a new Maven project.
   - Set the Group Id and Artifact Id. For example:
     - Group Id: com.example
     - Artifact Id: HelloWorldSpringMVC
2. **Add Dependencies to** `pom.xml`

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.
 <modelVersion>4.0.0</modelVersion>
 <groupId>com.example</groupId>
 <artifactId>HelloWorldSpringMVC</artifactId>
 <version>1.0-SNAPSHOT</version>
 <packaging>war</packaging>

 <dependencies>
     <!-- Spring MVC Dependency -->
     <dependency>
         <groupId>org.springframework</groupId>
         <artifactId>spring-webmvc</artifactId>
         <version>5.3.10</version>
     </dependency>

     <!-- Servlet API Dependency -->
     <dependency>
         <groupId>javax.servlet</groupId>
         <artifactId>javax.servlet-api</artifactId>
         <version>4.0.1</version>
         <scope>provided</scope>
     </dependency>

     <!-- JSTL Dependency -->
     <dependency>
         <groupId>javax.servlet</groupId>
         <artifactId>jstl</artifactId>
         <version>1.2</version>
     </dependency>
 </dependencies>

 <build>
     <finalName>HelloWorldSpringMVC</finalName>
 </build>
 </project>
```

3. **Configure Spring MVC**
   - In the src/main/webapp/WEB-INF directory, create a web.xml file to configure the Spring Dispatcher Servlet:

```
    <web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
                            http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd"
        version="3.1">
    <servlet>
       <servlet-name>dispatcher</servlet-name>
       <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
       <load-on-startup>1</load-on-startup>
    </servlet>


     dispatcher / index.jsp
```

4. **Create Spring Configuration File: In the src/main/webapp/WEB-INF directory, create a Spring configuration file named dispatcher-servlet.xml:**

```
<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.example" />
 <mvc:annotation-driven />

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

5. **Create Controller**
   - Create a controller class to handle requests. In src/main/java/com/example/controller, create a file named HelloController.java:

```
package com.example.controller;

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

@Controller
public class HelloController {

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

```
    }
}
```

6. **Create View**
    - Create a JSP file to display the message. In src/main/webapp/WEB-INF/views, create a file named hello.jsp:

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<!DOCTYPE html>
<html>
<head>
    <title>Hello World</title>
</head>
<body>
    <h1>${message}</h1>
</body>
</html>
```

7. **Run the ap**p: ▶ `mvn clean package` => Deploy the generated HelloWorldSpringMVC.war file to your Tomcat server => Access the application by navigating to `http://localhost:8080/HelloWorldSpringMVC/hello.`

# Inversion of Control and Dependency Injection

- **Inversion of Control (IoC)** is a design principle in software development where the control of object creation and management is transferred from the application code to a container or framework. In the context of Spring, the IoC container is responsible for instantiating, configuring, and managing the lifecycle of beans (objects).
- Key Concepts:
    - *IoC Container*: The core of the Spring Framework, which initializes the application and manages the lifecycle and configuration of application objects.
    - *Beans*: Objects that are managed by the Spring IoC container.
    - *Configuration Metadata*: The configuration information that the container uses to create and configure beans. This can be provided via XML, annotations, or Java configuration.
- **Dependency injection**: Dependency Injection (DI) is a pattern that allows an object (the client) to receive other objects that it depends on (dependencies). Instead of the client creating its dependencies, they are injected into the client by an external entity, typically the IoC container.
    - Types of Dependency Injection:
        - *Constructor Injection*: Dependencies are provided through the constructor of the class.
        - *Setter Injection*: Dependencies are provided through setter methods of the class.
        - *Field Injection*: Dependencies are injected directly into the fields of the class. This is less commonly used due to issues with testability and immutability.
           IOC flow

```
import...

@Component
public class Wheel{
    public String fun(){
        return "Something"
    }
```

```
}
```

```
import...

@RestController
public class Car{

   @Autowired
   private Wheel wheel;

   @GetMapping("/ok)
   public String ok(){
       return wheel.fun();
   }
}
```

# Creating Controller and Views

- In Spring MVC, controllers handle incoming requests from clients. Upon receiving a request, the controller invokes a business class to process business-related tasks. Once the tasks are completed, the controller redirects the client request to a logical view name, resolved by Spring's DispatcherServlet to render results or output.

 high level spring flow

## Responsibilities of a Controller

- **Intercept Incoming Requests**: Controllers are responsible for intercepting and handling incoming requests.
- **Request Mapping**: Convert the payload of the request to the internal structure of the data.
- **Data Processing**: Send data to the model for further processing after initial handling.
- **View Rendering**: Once processed, the data is referred to the view for rendering or display.

## Types of Controllers

- **Traditional MVC Controllers**: Not service-oriented and rely on view technology to redirect data received from a controller to views.
- **RESTful Controllers**: Designed as service-oriented, returning raw/response data typically in the form of JSON/XML, without views.

1. **Update web.xml**: The web.xml file configures the DispatcherServlet.

```
<?xml version="1.0" encoding="UTF-8"?>
 <web-app xmlns:xsi="https://www.w3.org/2001/XMLSchema-instance"
         xmlns="https://java.sun.com/xml/ns/javaee"
         xsi:schemaLocation="https://java.sun.com/xml/ns/javaee
                             https://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
         id="WebApp_ID" version="3.0">
     <display-name>IGNOU_Example</display-name>

     <!-- Adding DispatcherServlet as front controller -->
```

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

    <servlet-mapping>
        <servlet-name>ignou-serv</servlet-name>
        <url-pattern>/</url-pattern>
    </servlet-mapping>
 </web-app>
```

2. **Create Spring Bean Configuration**

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

    <!-- Enables @Controller programming model -->
    <mvc:annotation-driven />
    <context:component-scan base-package="com.ignou" />

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

3. **Write the Controller Class**

```
package com.ignou.controller;

 import java.text.DateFormat;
 import java.util.Date;
 import java.util.Locale;
 import org.springframework.stereotype.Controller;
 import org.springframework.ui.Model;
 import org.springframework.validation.annotation.Validated;
 import org.springframework.web.bind.annotation.RequestMapping;
 import org.springframework.web.bind.annotation.RequestMethod;

 @Controller
 public class IgnouController {
     @RequestMapping(value = "/", method = RequestMethod.GET)
```

```
    public String ignou(Locale locale, Model model) {
        System.out.println("Landing IGNOU Page Requested = " + locale);
        Date date = new Date();
        DateFormat dateFormat = DateFormat.getDateTimeInstance(DateFormat.LONG, DateFormat.LONG,
        String formattedDate = dateFormat.format(date);
        model.addAttribute("serverTime", formattedDate);
        return "ignou";
    }

    @RequestMapping(value = "/ignoustudent", method = RequestMethod.POST)
    public String user(@Validated User user, Model model) {
        System.out.println("IGNOU Student Page Requested");
        model.addAttribute("studentName", user.getStudentName());
        return "ignoustudent";
    }
}
```

4. **Define Model**

```
package com.ignou.model;

public class Ignoustudent {
    private String studentName;

    public String getStudentName() {
        return studentName;
    }

    public void setStudentName(String studentName) {
        this.studentName = studentName;
    }
}
```

5. **Create Views**

```
<%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>
<%@ taglib uri="https://java.sun.com/jsp/jstl/core" prefix="c"%>
<%@ page session="false"%>
<html>
<head>
    <title>IGNOU Landing Page</title>
</head>
<body>
    <h1>Spring MVC Example For IGNOU</h1>
    <p>The time on the server is ${serverTime}.</p>
    <form action="user" method="post">
        <input type="text" name="studentName"><br>
        <input type="submit" value="Login">
    </form>
</body>
</html>
```

```
<%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "https://www.w3.org/TR/html4/loose
<html>
```

```
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
    <title>IGNOU Student Home Page</title>
</head>
<body>
    <h3>Hello ${studentName}</h3>
</body>
</html>
```

- @Controller annotation tells the DispatcherServlet to look for controllers.
- The location of controller classes is specified by context:component-scan.
- The location of view pages is handled by the InternalResourceViewResolver bean.

# Request Parameters and Request Mapping in Spring MVC

- In Spring MVC, request parameters can be handled using the @RequestParam annotation, which binds parameters from a web request to the parameters of a handler method. Request mapping, on the other hand, defines how HTTP requests are mapped to specific handler methods within a controller.

## Handling Request Parameters

Request parameters are typically sent from views or clients and can be accessed and processed in a controller using `@RequestParam` annotation.

Example:

```
@Controller
@RequestMapping("/students")
public class StudentController {

    @GetMapping("/add")
    public String showForm() {
        return "studentForm";
    }

    @PostMapping("/add")
    public String processForm(ModelMap model,
                              @RequestParam("studentId") int studentId,
                              @RequestParam("firstName") String firstName,
                              @RequestParam("lastName") String lastName,
                              @RequestParam("dob") Date dob) {
        // Process the student data, e.g., save to database
        model.addAttribute("studentId", studentId);
        model.addAttribute("firstName", firstName);
        model.addAttribute("lastName", lastName);
        model.addAttribute("dob", dob);

        return "studentConfirmation";
    }
}
```

**Explanation:**

- GetMapping: Handles GET requests to /students/add and returns the studentForm.jsp.
- PostMapping: Handles POST requests to /students/add after form submission. It binds the request parameters (studentId, firstName, lastName, dob) to method parameters using @RequestParam.
- The method processes the form data, possibly saving it to a database, and adds attributes to the ModelMap.
- It then returns a view name (studentConfirmation.jsp) to render the confirmation page.

### Request Mapping Types

- Spring MVC supports different types of request mappings based on path, HTTP method, query parameters, and headers.
- **By Path** `@RequestMapping("/students")`
- **By HTTP Method** `@RequestMapping(value = "/students", method = RequestMethod.GET)`
  - Other methods such as POST, PUT, DELETE, OPTIONS, and TRACE are also supported.
- **By Query Parameter** @RequestMapping(value = "/students", method = RequestMethod.GET, params = "param1")
  - Ensures the handler method is executed only if the request contains a parameter named param1.
- **By Request Header** @RequestMapping(value = "/students", headers = "content-type=text/*")
  - Ensures the handler method is executed only if the request contains a header Content-Type with a value starting with text/.

## Form Tags and Data Binding in Spring MVC

- In Spring MVC, form tags and data binding play a crucial role in handling and processing user input from web forms. These features simplify the development of form-based web applications by providing reusable tags and mechanisms for binding form data to Java objects.

### Form Tags Overview

- Spring MVC provides a set of form tags that mirror HTML form elements but offer enhanced functionality for binding data to Java objects.

### Setting Up Form Tags

- To use Spring MVC form tags in JSP pages, include the following taglib directive at the beginning of your JSP file:
- `<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form" %>`

### Commonly used Form Tags

| Form Tag | Description |
|---|---|
| `form:form` | Container tag for enclosing other form tags. |
| `form:input` | Generates a text input field. |
| `form:radiobutton` | Generates radio buttons. |

| Form Tag | Description |
| --- | --- |
| `form:checkbox` | Generates checkboxes. |
| `form:password` | Generates a password input field. |
| `form:select` | Generates a dropdown list. |
| `form:textarea` | Generates a multi-line text field. |
| `form:hidden` | Generates a hidden input field. |

## Data Binding Mechanism

- Spring MVC's data binding mechanism allows binding user inputs directly to Java objects (often referred to as Beans or POJOs).

1. Model Class (Student.java):

```
package com.ignou.studentdetails;

 public class Student {
     private String fname;
     private String lname;

     // Getters and setters
     // Constructor
 }
```

2. Controller (StudentDetailController.java):

```
package com.ignou.studentdetails;

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

 @Controller
 public class StudentDetailController {

     @RequestMapping(value = "/student_form", method = RequestMethod.GET)
     public String showStudentForm(Model model) {
         Student student = new Student();
         model.addAttribute("student", student);
         return "studentForm";
     }

     @RequestMapping(value = "/studentdetail", method = RequestMethod.POST)
     public String processStudentForm(@ModelAttribute("student") Student student) {
         // Process student data (save to database, etc.)
         return "studentData";
     }
```

```
}
```

3. View (studentForm.jsp):

```
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form" %>
<html>
<head>
    <title>Student Detail Form</title>
</head>
<body>
    <form:form action="studentdetail" modelAttribute="student">
        First Name: <form:input path="fname" /><br>
        Last Name: <form:input path="lname" /><br>
        <input type="submit" value="Submit"/>
    </form:form>
</body>
</html>
```

4. View (studentData.jsp)

```
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<html>
<head>
    <title>Student Detail Data</title>
</head>
<body>
    The student name is ${student.fname} ${student.lname}
</body>
</html>
```

# Form Validation

- Spring validation is crucial for validating inputs in `@Controller`. It ensures server-side validation of user inputs, making it essential for accepting user input in web applications. From Spring Framework version 4 and above, it supports Bean Validation 1.0 (JSR-303) and Bean Validation 1.1 (JSR-349).

## BindingResult

`BindingResult` is an interface representing the binding results. It extends an interface for error registration, allowing a validator to be applied, and adds binding-specific analysis.

## Commonly Used Validation Annotations

In Spring MVC form validation, various annotations are used. These annotations are available in the `javax.validation.constraints` package. Below is a list of commonly used validation annotations:

| Annotation | Description |
|---|---|
| `@Valid` | This annotation is used on a model object that we want to validate. |
| `@Size` | It specifies that the size of the annotated element must be between the specified boundaries. |
| `@Pattern` | It specifies that the annotated `CharSequence` must match the regular expression. |
| `@Past` | It specifies that the annotated element must be a date in the past. |
| `@Null` | It specifies that the annotated element must be null. |
| `@NotNull` | It specifies that the annotated element should not be null. |
| `@Min` | It specifies that the annotated element must be a number whose value must be equal or greater than the specified minimum number. |
| `@Max` | It specifies that the annotated element must be a number whose value must be equal or smaller than the specified maximum. |
| `@Future` | It specifies that the annotated element must be a date in the future. |
| `@Digits` | It specifies that the annotated element must be a digit or number within the specified range. The supported types are short, int, long, byte, etc. |
| `@DecimalMin` | It specifies that the annotated element must be a number whose value must be equal or greater than the specified minimum. It is very similar to `@Min` annotation, but the only difference is it supports `CharSequence` type. |
| `@DecimalMax` | It specifies that the annotated element must be a number whose value should be equal or smaller than the specified maximum. It is very similar to `@Max` annotation, but the only difference is it supports `CharSequence` type. |
| `@AssertTrue` | It determines that the annotated element must be true. |
| `@AssertFalse` | It determines that the annotated element must be false |

```
package com.ignou.studentdetails;
import javax.validation.constraints.Min;
import javax.validation.constraints.NotNull;
import javax.validation.constraints.Size;
public class Student
{
    @NotNull
    @Size(min =1, message ="You can't leave this empty.")
    private String fname;
    @NotNull
    @Size(min =1, message ="You can't leave this empty.")
    private String lname;

    public Student()
    {
    }
    public String getFname()
    {
```

```
        return fname;
    }
    public void setFname(String fname)
    {
        this.fname = fname;
    }
    public String getLname()
    {
        return lname;
    }
    public void setLname(String lname)
    {
        this.lname = lname;
    }
}
```

## Check your progress - 1

1. What do mean by IDE and its benefits?
2. Is Spring Framework open source?
3. What is the use of IoC?
4. What is DI?

## Check your progress - 2

1. What is an Init Binder?

- The Init Binder is an annotation that is used to customize the request being sent to the controller. It is used to to control and format those requests that come to the controller.

2. What is the front controller class of Spring MVC?

- A controller is used to handle all requests for a Web Application is called as Front Controller. DispatcherServlet (actually a servlet) is the front controller in Spring MVC that not only intercepts every request but also dispatches/forwards requests to an appropriate controller.

3. What is the difference between @Component, @Controller, @Repository & @Service annotations?

- One of the major difference between these stereotypes is that these are used for different classification. You may have different layers like presentation, service, business, data access etc. in a multitier application. When you try to annotated a class for auto-detection by Spring, then you should have to use the respective stereotype as below: - @Component – generic and can be used across application. - @Service – annotate classes at service layer level. - @Controller – annotate classes at presentation layers level, mainly used in Spring MVC. - @Repository – annotate classes at persistence layer, which will act as database repository.

4. What is the ViewResolver class?

- ViewResolver is an interface wich is to be implemented by objects. This is used to resolve views by name. There are many other ways also, using which you can resolve view names. These ways are supported by various in-built

implementations of ViewResolve.

# Check your progress - 3

1. What do you understand by Tag libraries?
    - Custom tags are implemented and distributed using taglib. It is a structure which is known as a tag library. A tag library is nothing but a a collection of classes and meta-information that includes:
        - Tag Handlers Java classes. This class implement the functionality of custom tags.
        - Tag Extra Information Classes. These classes are used to supply the JSP container with logic for validating tag attributes and creating scripting variables.
        - A Tag Library Descriptor (TLD). It is an XML document, used to describe the properties of the individual tags and the tag library as a whole.
2. What are the @RequestBody and the @ResponseBody?
    - @RequestBody and @ResponseBody annotations are used to convert the body of the HTTP request and response with java class objects. Both these annotations will use registered HTTP message converters in the process of converting/mapping HTTP request/response body with java objects.
3. State the difference between Request Param & Request Mapping
    - @RequestMapping is a class or method annotation is used to map the request url to the java method.
    - @RequestParam is a (Method) field annotation is used to bind a request parameter to a method parameter
4. What is the Role of the @Autowired Annotation?
    - The @Autowired annotation can be used to autowire (placing an instance of one bean into the desired field of another bean) bean on the setter method just like @Required annotation, constructor, a property or methods with arbitrary names or multiple arguments