ĐẠI HỌC TÔN ĐỨC THẮNG
TON DUC THANG UNIVERSITY

503111

Java Technology

SPRING MVC

Introduction to Spring Framework

- The Spring Framework is a Java platform that provides comprehensive infrastructure support for developing Java applications.
- Spring is the most popular application development framework for enterprise Java.
- The core features of the Spring Framework can be used in developing any Java application, but there are extensions for building web applications.

Spring Framework
Topic

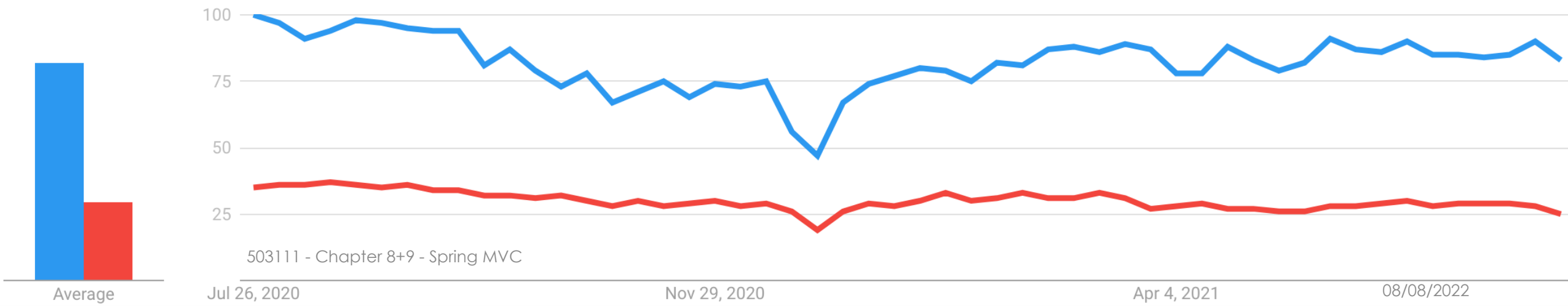
ASP.NET
Search term

+ Add comparison

Worldwide | Past 12 months | All categories | Web Search

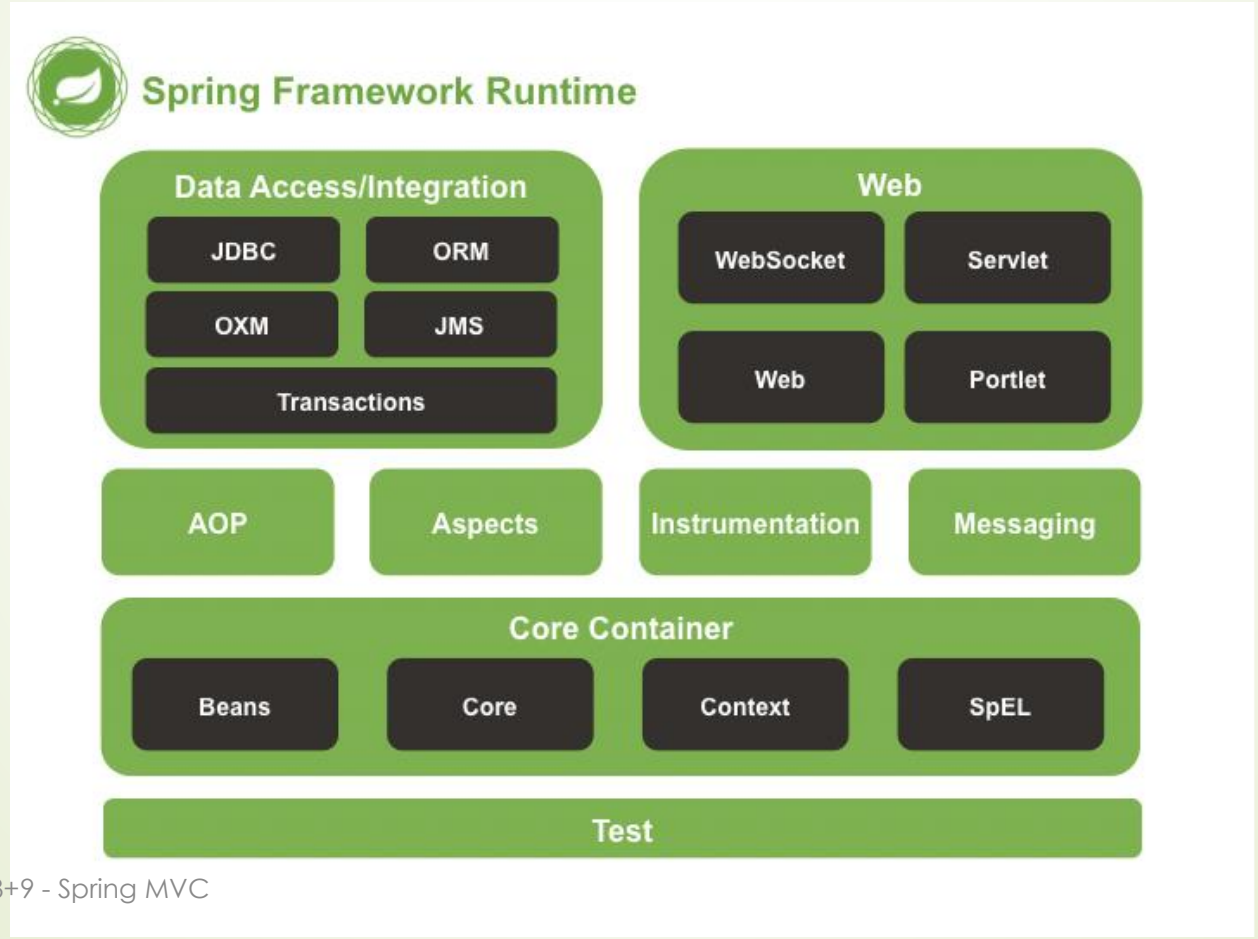
Note: This comparison contains both Search terms and Topics, which are measured differently. LEARN MOR

Interest over time



Spring Modules

- The Spring Framework consists of features organized into about 20 modules.



Spring Web Layer

- The *Web* layer consists of the spring-web, spring-webmvc and spring-websocket modules.
  - 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.

Spring Framework

- Spring Framework is based on two design principles:
  - Inversion of Control & Dependency Injection
  - Aspect Oriented Programming

Inversion of Control

- IoC is a design principle which recommends the inversion of different kinds of controls in object-oriented design to achieve loose coupling between application classes.
- In this case, control refers to any additional responsibilities a class has, other than its main responsibility, such as control over the flow of an application, or control over the dependent object creation and binding.
- If you want to do TDD (Test Driven Development), then you must use the IoC principle, without which TDD is not possible

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.
- 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).

Dependency Injection

- Dependency Injection (DI) is a design pattern which implements the IoC principle to invert the creation of dependent objects.
- 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 ProductService {

    private ProductRepository repo = new ProductRepository();

    public void addProduct(Product p) {
        repo.add(p);
    }
}
```

Dependency Injection

- By using DI, we can rewrite the example without specifying the implementation of the ProductRepository that we want.

```
public class ProductService {

    private ProductRepository repo;

    public ProductService(ProductRepository repo) {
        this.repo = repo;
    }
}
```

- This example is called Constructor-Based Dependency Injection.

Dependency Injection

- Setter-Based Dependency Injection.

```
public class ProductService {

    private ProductRepository repo;

    public ProductService() {
    }

    public void setProductRepository(ProductRepository repo) {
        this.repo = repo;
    }
}
```

Spring Core

Spring Core Terminologies

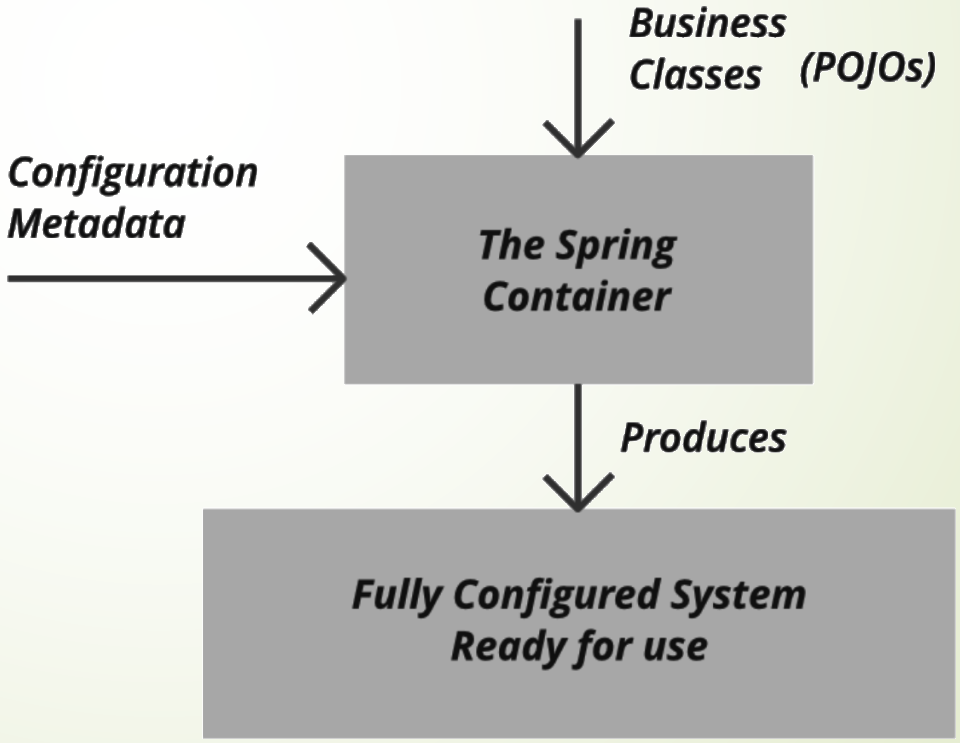
- IoC container (Application Context)
- Bean
- Configuration Metadata

IoC Container

- The IoC container is a framework used to manage automatic dependency injection throughout the application.
- 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.

IoC Container

- The container gets its instructions on what objects to instantiate, configure, and assemble by reading the configuration metadata provided.
- The configuration metadata can be represented either by XML, Java annotations, or Java code.



Spring Configuration Metadata

- The Spring IoC container consumes a form of configuration metadata.
- This configuration metadata represents how you, as an application developer, tell the Spring container to instantiate, configure, and assemble the objects in your application.
- Following are the three important methods to provide configuration metadata to the Spring Container:
  - XML based configuration file
  - Annotation-based configuration
  - Java-based configuration

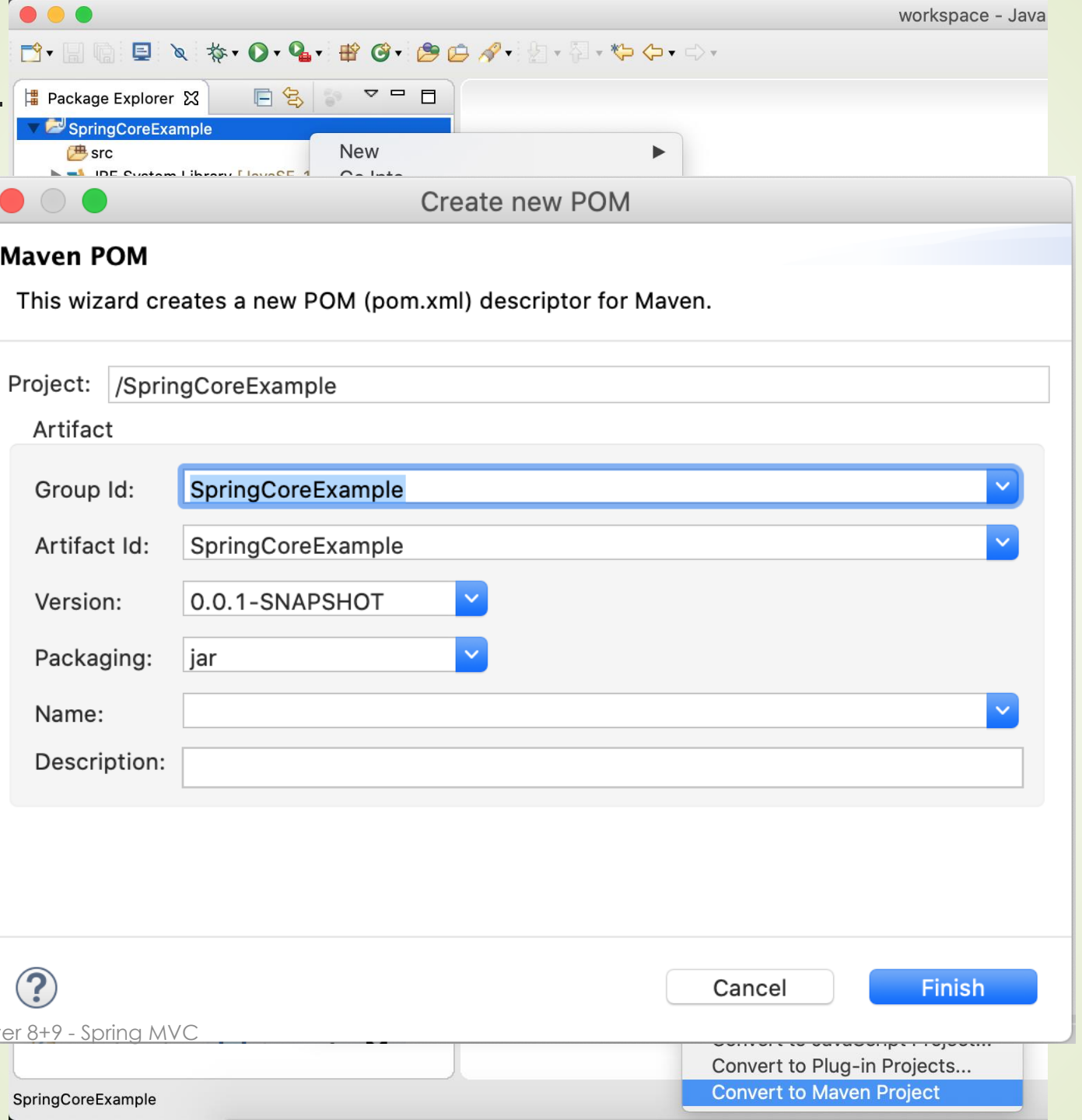
Create a XML-based Spring Core Project

XML-based Spring Core project

1. Creae a Standard Java Project from the Eclipse IDE
2. Convert the project to a Maven Project
3. Add the spring-context dependency (pom.xml)
4. Create two Java classes: Product.java and ProductService.java
5. Create the Spring Configuration Metadata file: appContext.xml
6. Create Program.java which contain the main method.

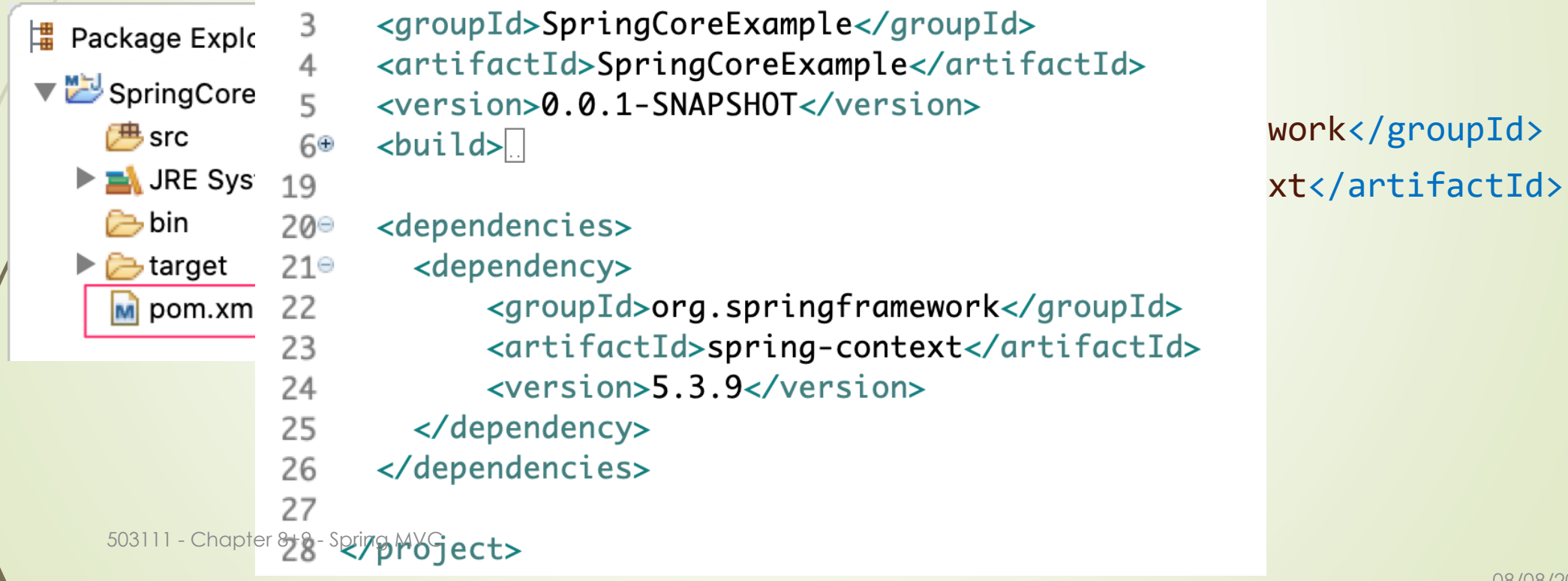
XML-

1. Create
2. Conver



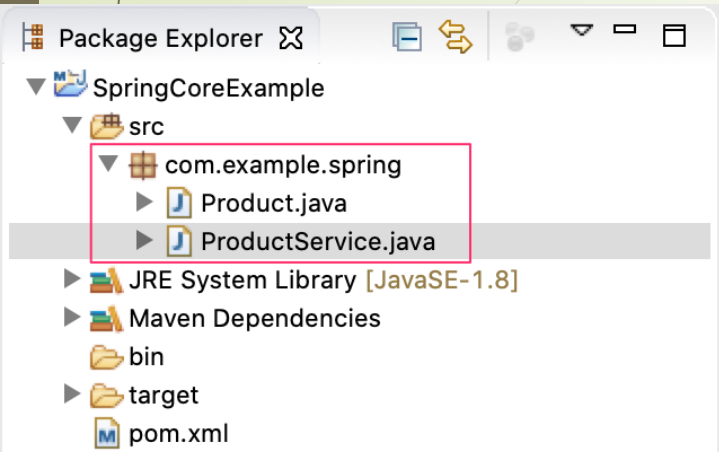
XML-based Spring Core project

3. Add the spring-context dependency



XML-based Spring Core project

4. Create two classes



Product.java | ProductService.java

```
4  import java.util.List;
5
6  public class ProductService {
7
8      private List<Product> list = new ArrayList<>();
9
10     public List<Product> findAll() {
11         return list;
12     }
13
14     public Product find(int id) {
15         return list.stream().filter(p -> p.getId() == id).findFirst().get();
16     }
17
18     public void add(Product p){
19         list.add(p);
20     }
21
22 }
23
```

XML-based Spring Core project

- Under the src folder, create a file called appContext.xml with the following content:

```
<?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">

   // content here

</beans>
```

appContext.xml

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


</beans>
```

XML-based Spring Core project

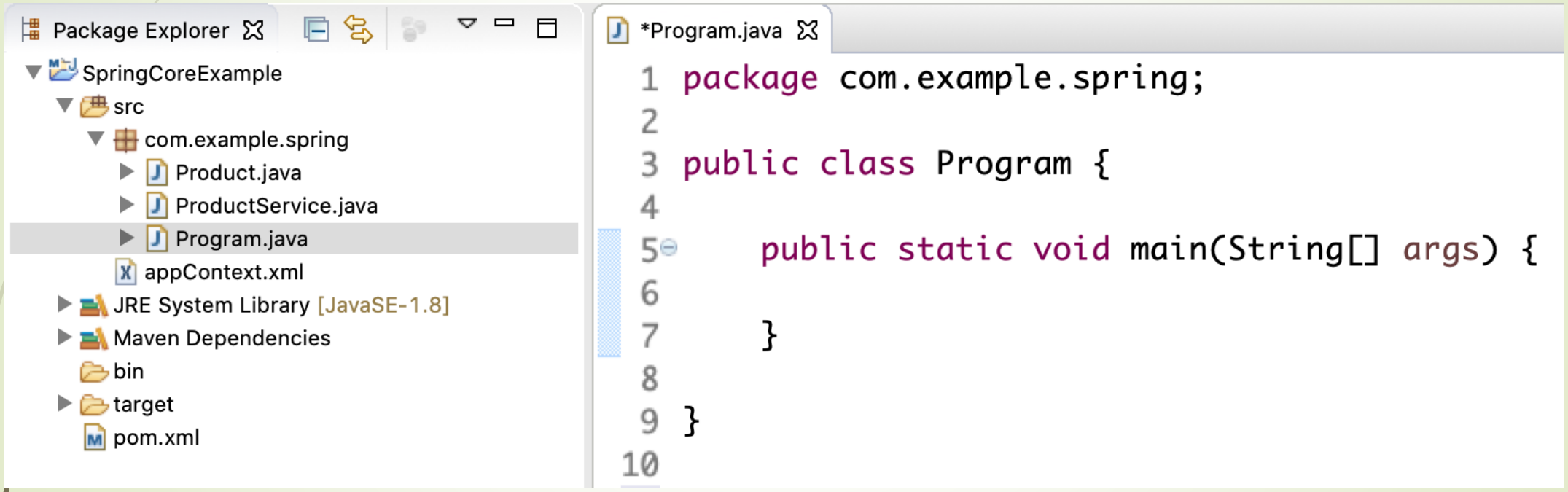
- Declare bean definations for the Product and ProductService classes.

appContext.xml

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

    <bean id="myProduct"
        class="com.example.spring.Product" scope="prototype"/>

    <bean id="productService"
        class="com.example.spring.ProductService" scope="singleton"/>

</beans>
```

XML-based Spring Core project

- Create a Program.java where we put the main method:



XML-based Spring Core project

Program.java

```
6  public class Program {
7
8      public static void main(String[] args) {
9
10         ApplicationContext ct = new ClassPathXmlApplicationContext("appContext.xml");
11
12         Product p1 = ct.getBean(Product.class);
13         Product p2 = ct.getBean(Product.class);
14
15         ProductService service1 = ct.getBean(ProductService.class);
16         ProductService service2 = ct.getBean(ProductService.class);
17
18         System.out.println(p1 == p2); // fase
19         System.out.println(service1 == service2); // true
20
21     }
22 }
```

Bean

- 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.
- Bean Properties & Description:
  - class: This attribute specifies the class to be used to create the bean.
  - name: This attribute specifies the bean identifier uniquely.
  - scope: This attribute specifies the scope of the objects created from a particular bean.
  - constructor-arg: This is used to inject the dependencies
  - properties: This is used to inject the dependencies

appContext.xml

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

    <bean id="product1" class="com.example.spring.Product" />

    <bean id="product2" class="com.example.spring.Product">
        <property name="id" value="139"/>
        <property name="name" value="iPhone X"/>
        <property name="price" value="1299"/>
    </bean>

</beans>
```

```
public class
    public st
        Appli
        Produ
        Produ
        System.out
        System.out
    }
}
```

ontext.xml");

15 </be

21
22
23);

Problems @ Javadoc Declaration Console

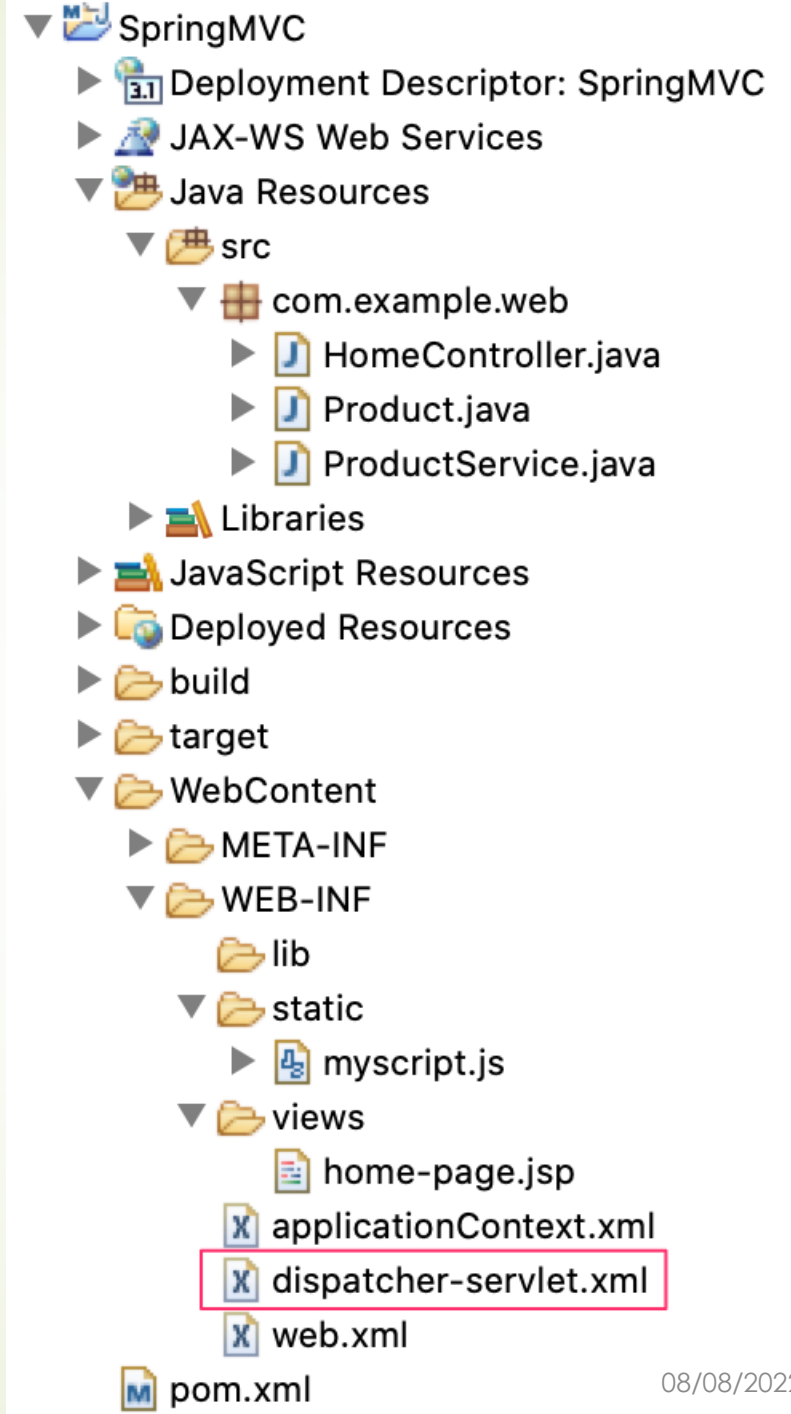
<terminated> Program (3) [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_2

```
setId() is called
setName() is called
setPrice() is called
Product (id=0, name=null, price=0)
Product (id=139, name=iPhone X, price=1299)
```

Spring MVC

Sample Project Structure

- src: contains all java source codes.
- WEB-INF: contains all web related stuffs and spring configuration files.
- static: contains all static resources like javascripts, stylesheets, images…
- views: contains all view template files.
- pom.xml: contains maven configrations and dependencies.
- web.xml: contains settings for the entire web app
- applicationContext.xml: Spring application context
- dispatcher-servlet.xml: Spring WebApplicationContext



The pom.xml file

```
1  <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-
   4.0.0.xsd">
2
3    <dependencies>
4      <dependency>
5          <groupId>org.springframework</groupId>
6          <artifactId>spring-webmvc</artifactId>
7          <version>5.3.9</version>
8      </dependency>
9
10     <dependency>
11         <groupId>javax.servlet</groupId>
12         <artifactId>jstl</artifactId>
13         <version>1.2</version>
14     </dependency>
15   </dependencies>
16
17 </project>
```

Home » org.springframework » spring-webmvc » 5.3.9

Spring Web MVC » 5.3.9

Spring Web MVC

License	Apache 2.0
Categories	Web Frameworks
Organization	Spring IO
HomePage	https://github.com/spring-projects/spring-framework
Date	(Jul 14, 2021)
Files	pom (2 KB) jar (983 KB) View All
Repositories	Central
Used By	**4,328 artifacts**

Maven | Gradle | Gradle (Short) | Gradle (Kotlin) | SBT | Ivy | Grape | Leiningen | Buildr

```
<!-- https://mvnrepository.com/artifact/org.springframework/spring-webmvc -->
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-webmvc</artifactId>
    <version>5.3.9</version>
</dependency>
```


☑ Include comment with link to declaration

08/08/2022

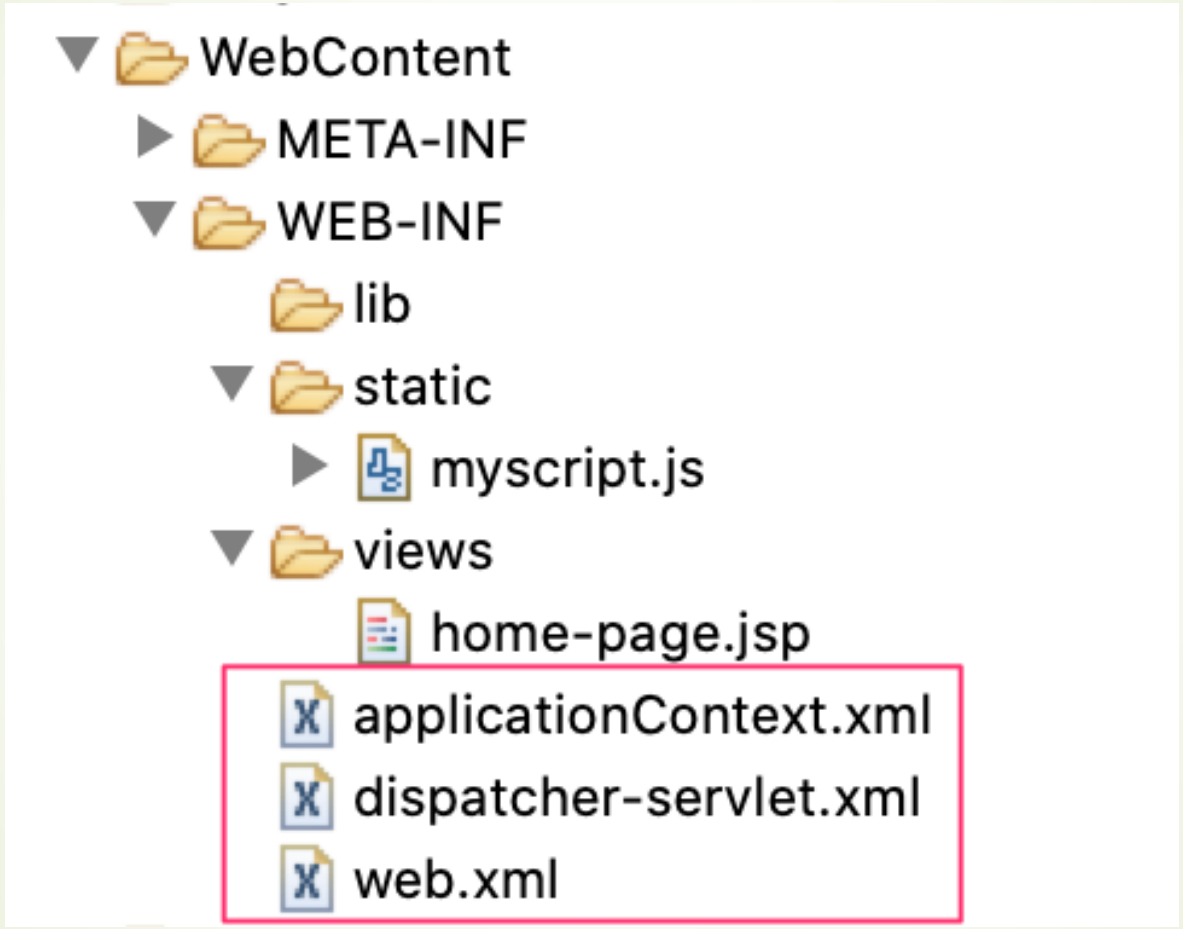
The web.xml file

- The web.xml file is the standard deployment descriptor for the Web application that the Web service is a part of.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" ...>
3    <display-name>SpringMVC</display-name>
4
5    <listener>
6      <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
7    </listener>
8
9
10
11
12
13   ...
14 </web-app>
```

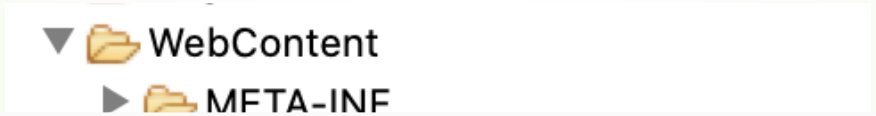
```xml
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" ...>

  ...

  <servlet>
        <servlet-name>dispatcher</servlet-name>
        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>




        <load-on-startup>1</load-on-startup>
    </servlet>

    <servlet-mapping>
        <servlet-name>dispatcher</servlet-name>
        <url-pattern>/</url-pattern>
    </servlet-mapping>

</web-app>
```

The web.xml file



The applicationContext.xml file

- It is standard spring context file which contains all beans and the configuration that are common among all the servlets.
- It is optional file in case of web app.



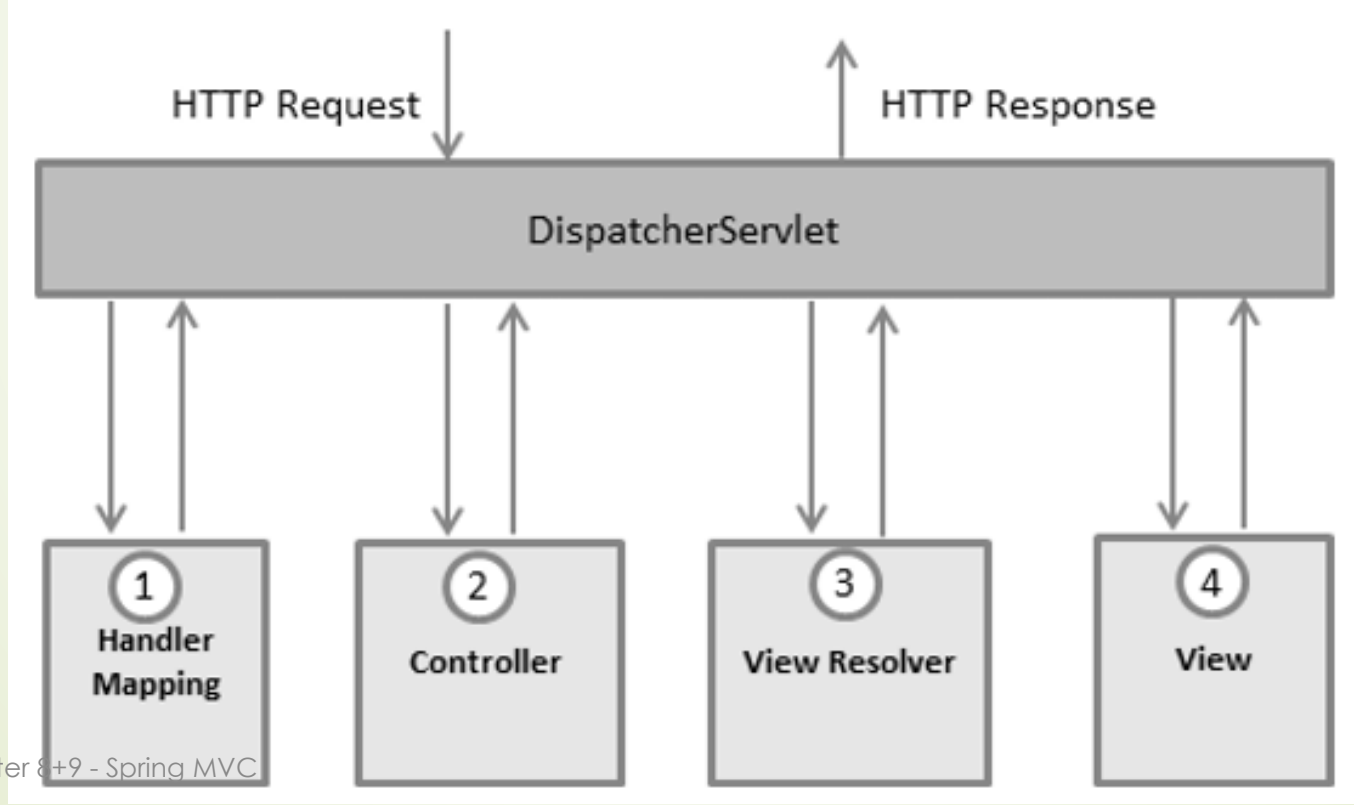

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    ...>

    <bean id="productDAO" class="com.example.web.ProductService" scope="singleton" />

</beans>
```

The dispatcher-servlet.xml file

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    ...>

    <mvc:annotation-driven/>
    <context:component-scan base-package="com.example.web" />
    <mvc:default-servlet-handler />

    <mvc:resources location="/WEB-INF/static/" mapping="/resources/**" />

    <bean id="viewResolver"
        class="org.springframework.web.servlet.view.UrlBasedViewResolver">

        <property name="viewClass" value="org.springframework.web.servlet.view.JstlView"/>
        <property name="prefix" value="/WEB-INF/views/" />
        <property name="suffix" value=".jsp" />
    </bean>

</beans>
```

The DispatcherServlet

- The Spring Web model-view-controller (MVC) framework is designed around a *DispatcherServlet* that handles all the HTTP requests and responses.



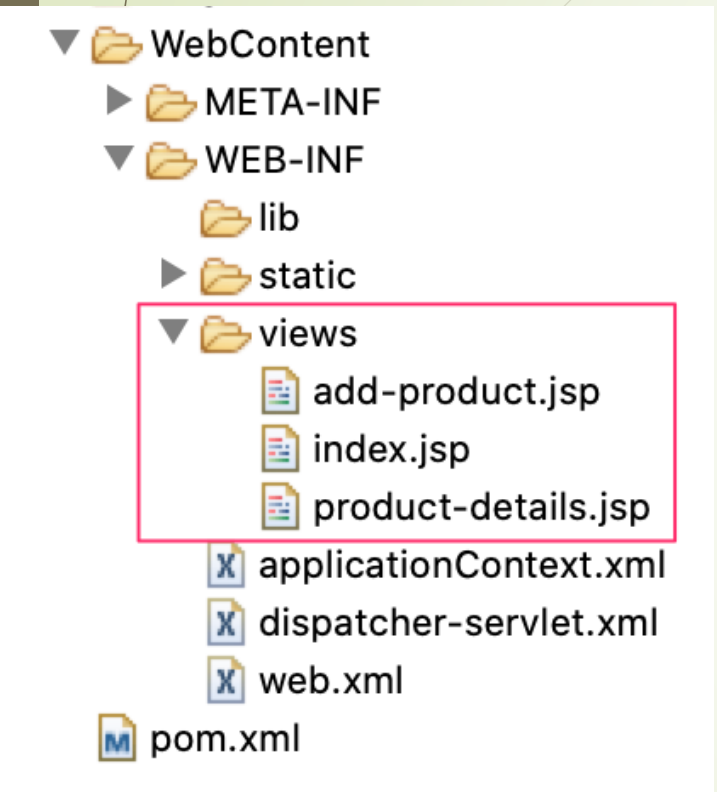
The Controller

- To create the controller class, we are using two annotations @Controller and @RequestMapping:
  - The @Controller annotation indicates that a particular class serves the role of a controller.
  - The @Requestmapping annotation is used to map the class with the specified URL name.

```
@Controller
@RequestMapping("/products")
public class ProductController {


}
```

Action Methods (Handler Methods)

- The Controller class contains handler methods called action methods.
- They following some restrictions:
  - Action method must be public. It cannot be private or protected
  - Action method cannot be a static method.
  - Action method must be annotated with @RequestMapping or its child annotations.
  - By default, action methods return a String value indicating the name of the view to be rendered.




```java
@Controller
@RequestMapping("/products")
public class ProductController {

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

    @RequestMapping("/{id}")
    public String details(@PathParam("id") int id) {
        return "product-details";
    }

    @RequestMapping("/add")
    public String add() {
        return "add-product";
    }
}
```

http://localhost/products/

http://localhost/products/212

http://localhost/products/add

@RequestMapping Variants

- Spring Framework 4.3 introduces the following method-level composed variants of the @RequestMapping annotation that help to simplify mappings for common HTTP methods and better express the semantics of the annotated handler method:
  - @GetMapping
  - @PostMapping
  - @PutMapping
  - @DeleteMapping
  - @PatchMapping

```
@Controller
@RequestMapping("/products")
public class ProductController {

    @RequestMapping(value = "/{id}", method = RequestMethod.POST)
    public String details(@PathParam("id") int id) {
        return "product-details";
    }

    @PostMapping
    public String add(Product product) {
        return "add-product";
    }


}
```

Path Variables and Request Parameters

- *URI templates* can be used for convenient access to selected parts of a URL.
- A URI Template is a URI-like string, containing one or more variable names. When you substitute values for these variables, the template becomes a URI.

/products/{id}

/report/{month}/{year}

- You can aslo narrow request matching through request parameter conditions:

/all-products?page=3&sortOrder=price

Path Variables and Request Parameters

```
@Controller
public class ProductController {

    @RequestMapping(value = "/products/{id}")
    public String details( @PathVariable("id") Integer id) {
        return "product-details";
    }


    @RequestMapping(value = "/all-products")
    public String list( @RequestParam(value = "page", defaultValue = "1")  int page,
       @RequestParam(value = "sortBy") String sortBy) String sortBy) {
        return "product-details";
    }
}
```

http://localhost/products/212

http://localhost/all-products?page=2&sortOrder=price

Model

- In Spring MVC, the model works as a container that contains the data of the application.
- Here, a data can be in any form such as objects, strings, information from the database, etc.
- In order to use the model, It is required to place the **Model** object in the action method of a controller.

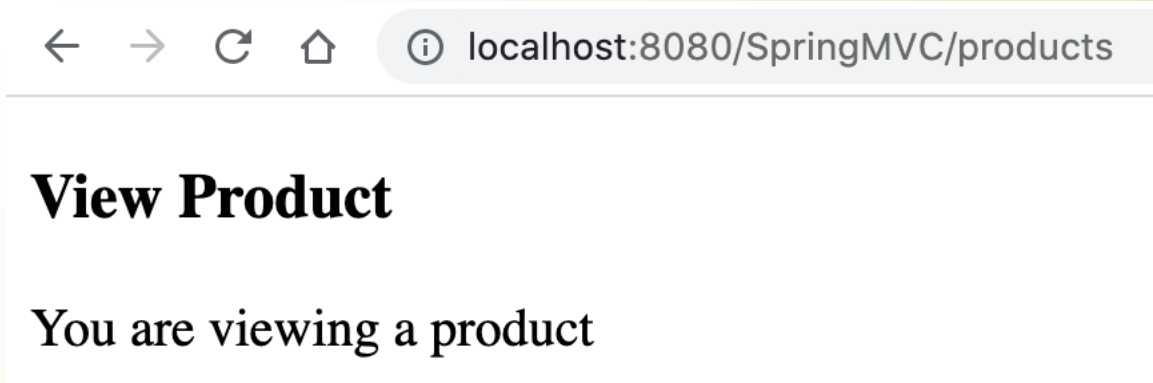
```
@RequestMapping(value = "/products")
public String index(Model model) {

    model.addAttribute("title", "View Product");
    model.addAttribute("message", "You are viewing a product");

    return "product-details";
}
```

Accessing Model from JSP

- Inside a JSP file, we use the ${name} syntax to access data from the model, where name is the name of the attribute that we have previously added to the model.

product-details.jsp

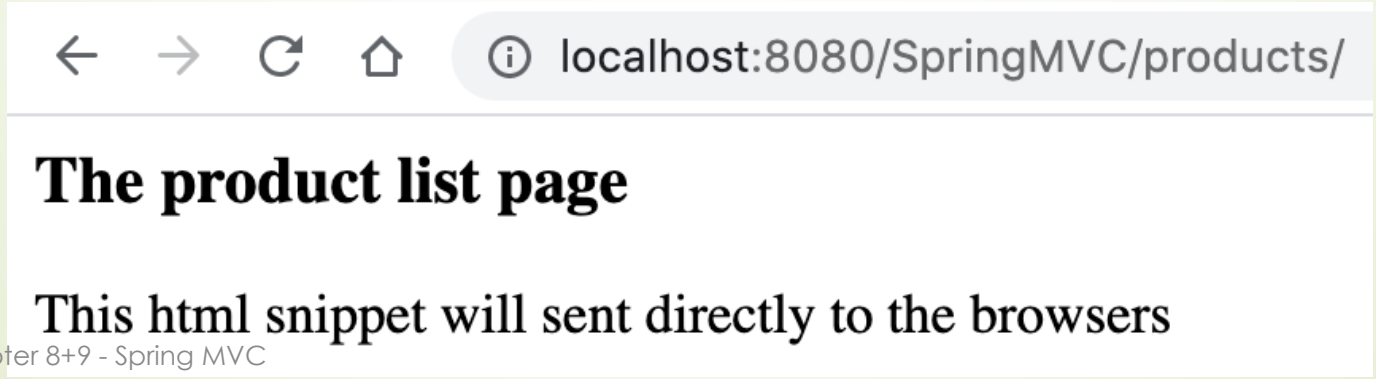
```
1  <!DOCTYPE html>
2  <html>
3  <head>
4  <title>Insert title here</title>
5  </head>
6  <body>
7      <h3>${title}</h3>
8      <p>${message}</p>
9  </body>
10 </html>
```



Action Method without view

- In situations where we just want to send a response directly from the action method without using a view, we use the @ResponseBody annotation.

```
@GetMapping(value = "/", produces = "text/html")
@ResponseBody
public String index() {
    return "<h3>The product list page</h3>" +
            "<p>This html snippet will sent directly to the browsers<p>";
}
```



Action Method Arguments

- The following are the supported method arguments:
  - Model
  - Primitive types (int, String, boolean)
  - User defined types (Product, Employee)
  - HttpSession
  - HttpServletRequest
  - @PathVariable annotated parameters
  - @RequestParam annotated parameters
  - @RequestHeader annotated parameters
  - @RequestBody annotated parameters
  - @RequestPart annotated parameters
  - Errors/BindingResult: validation results

Action Method Arguments: HttpSession

- The HttpSession object allows us to persist data across user requests.

```
@GetMapping()
@ResponseBody
public String index(HttpSession session) {
    if (session.getAttribute("user") == null) {
        return "User exists";
    }
    session.setAttribute("user", "admin");
    return "Session is created";
}
```

Data Binding

- Parameter binding works automatically if the parameter names matches.

http://localhost/products/add?id=1&name=iPhone&price=999

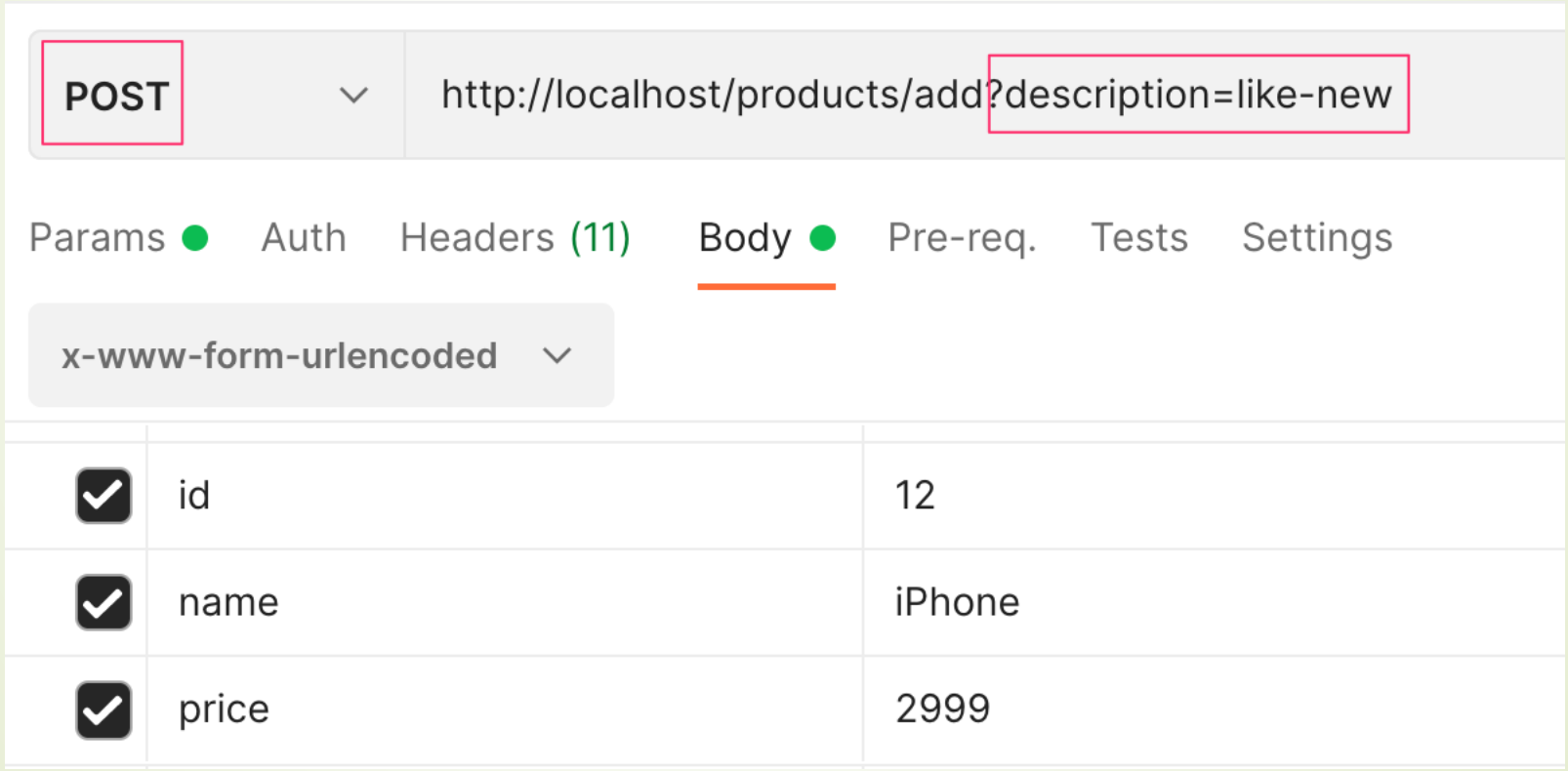
```
public class Product{
  private int id;
  private String name;
  private int price;

  // constructors
  // getters & setters
}
```

```
@GetMapping("/add")
@ResponseBody
public String add(int id, String name, int price) {
    return id + ", " + name + ", " + price;
}
```

```
@GetMapping("/add")
@ResponseBody
public Product add(Product p) {
    return p;
}
```

Data Binding

- Data binding of the primitive types and complex objects can also work together.



Spring Tag Library

Spring Tags Library

- There are two types of tag that Spring support to work with JSP:
  - Standard tags
  - Form tags
- In order to use these tags, we first must include the taglib directive at the beginning of the jsp file

add-product.jsp

```
1 <%@ taglib prefix="spring" uri="http://www.springframework.org/tags" %>
2 <%@ taglib prefix="f" uri="http://www.springframework.org/tags/form" %>
3
4 <!DOCTYPE html>
5 <html>
6 <head>
```

Spring Standard Tags

- Spring Framework provides you with some tags for evaluating errors, setting themes and outputting internationalized messages.
- To enable the support for standard tag library, it is required to include the following tablib directive at the beginning of the jsp file.

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

- List of commond standard tags:
  - <spring:url>
  - <spring:message>
  - <spring:htmlEscape>
  - <spring:theme>

Spring Standard Tags

- The Spring URL tag:
  - Creates URLs with support for URI template variables, HTML/XML escaping, and Javascript escaping.

```
<spring:url value="/products/add" />
```

equivalent to

```
/SpringMVC/products/add
```

Spring Standard Tags

- The Spring URL tag:
  - Creates URLs with support for URI template variables, HTML/XML escaping, and Javascript escaping.

Instead of writing

```
<form method="post" action="/SpringMVC/products/add">
```

We write

```
<form method="post" action="<spring:url value="/products/add"/>">
```

Or

```
<spring:url value="/products/add" var="target" />

<form method="post" action="${target}" >
```

Spring Form Tags

- The Spring MVC form tags can be seen as data binding-aware tags that can automatically set data to Java object/bean and also retrieve from it.
- Here, each tag provides support for the set of attributes of its corresponding HTML tag counterpart, making the tags familiar and easy to use.
- To enable the support for form tag library, it is required to include the following tablib directive at the beginning of the jsp file.

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

Spring Form Tags

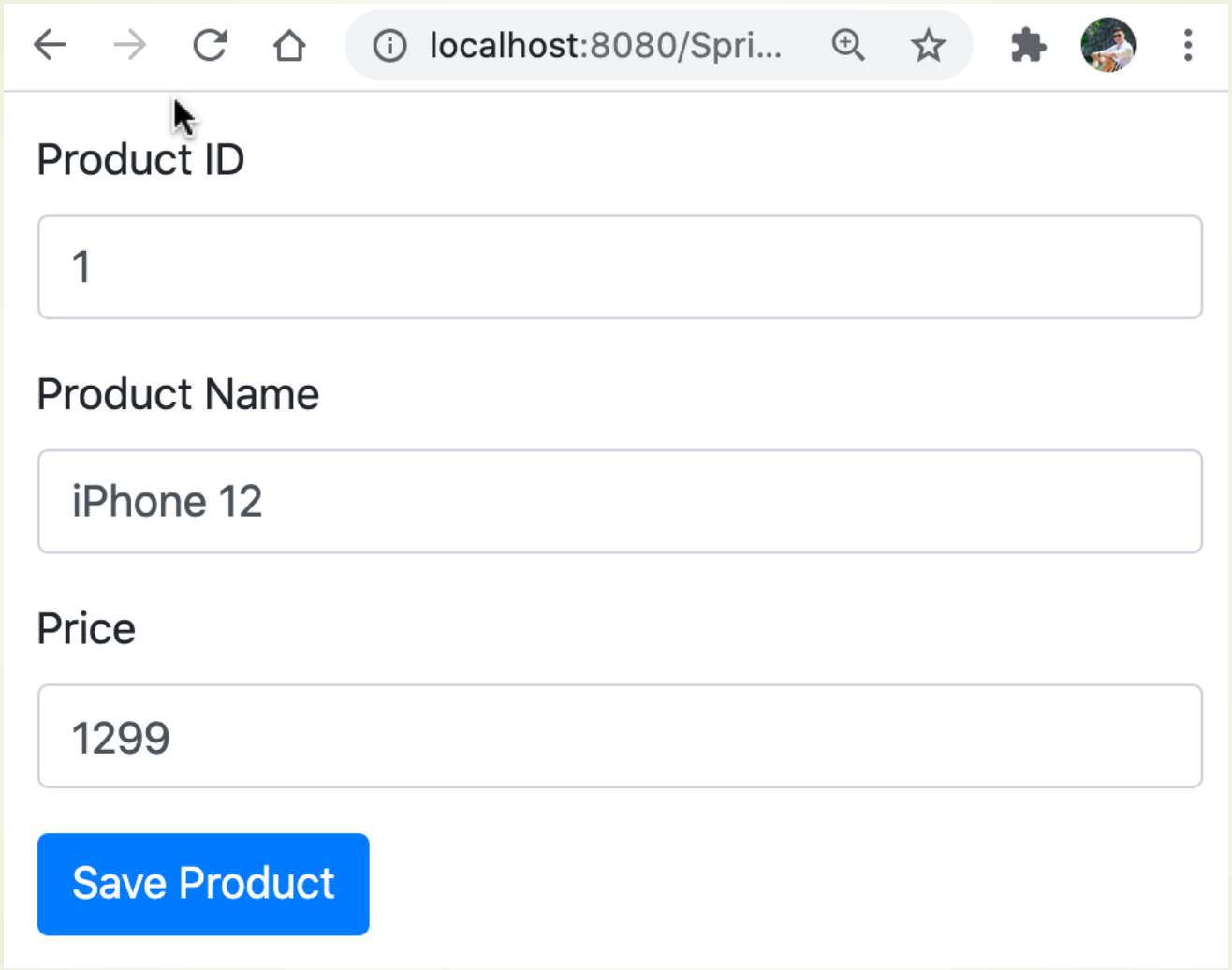
- The <f:form> tag:
  - This tag renders an HTML 'form' tag and exposes a binding path to inner tags for binding.
  - It puts the command object in the PageContext so that the command object can be accessed by inner tags.
  - All the other tags in this library are nested tags of the form tag.
- Children tags:
  - <f:input>, <f:password>
  - <f:label>
  - <f:radio>, <f:checkbox>
  - <f:select>, <f:textarea>

```
<spring:url value="/products/save" var="target"/>

<f:form method="post" action="${target}" modelAttribute="product">
    <div class="form-group">
        <f:label path="id">Product ID</f:label>
        <f:input path="id" cssClass="form-control"/>
    </div>

    <div class="form-group">
        <f:label path="name">Product Name</f:label>
        <f:input path="name" cssClass="form-control"/>
    </div>

    <div class="form-group">
        <f:label path="price">Price</f:label>
        <f:input path="price" cssClass="form-control"/>
    </div>

    <div class="form-group">
        <button class="btn btn-primary">Save Product</button>
    </div>
</f:form>
```

Spring Form Tags

- Back end code
  - Get /edit: return the product form
  - Post /save: handle saving product

```
@GetMapping("/edit")
public String add(Model model) {

    model.addAttribute("product", new Product(1, "iPhone 12", 1299));

    return "edit-product";
}

@PostMapping("/save")
public String save(@ModelAttribute("product") Product product) {
    System.out.println(product);
    // save product
    return "index";
}
```


localhost:8080/Spri...
Product ID
1
Product Name
iPhone 12
Price
1299
Save Product

Spring MVC Examples

- CRUD Example
- Dependency Injection
- Validator
- Exception Handling