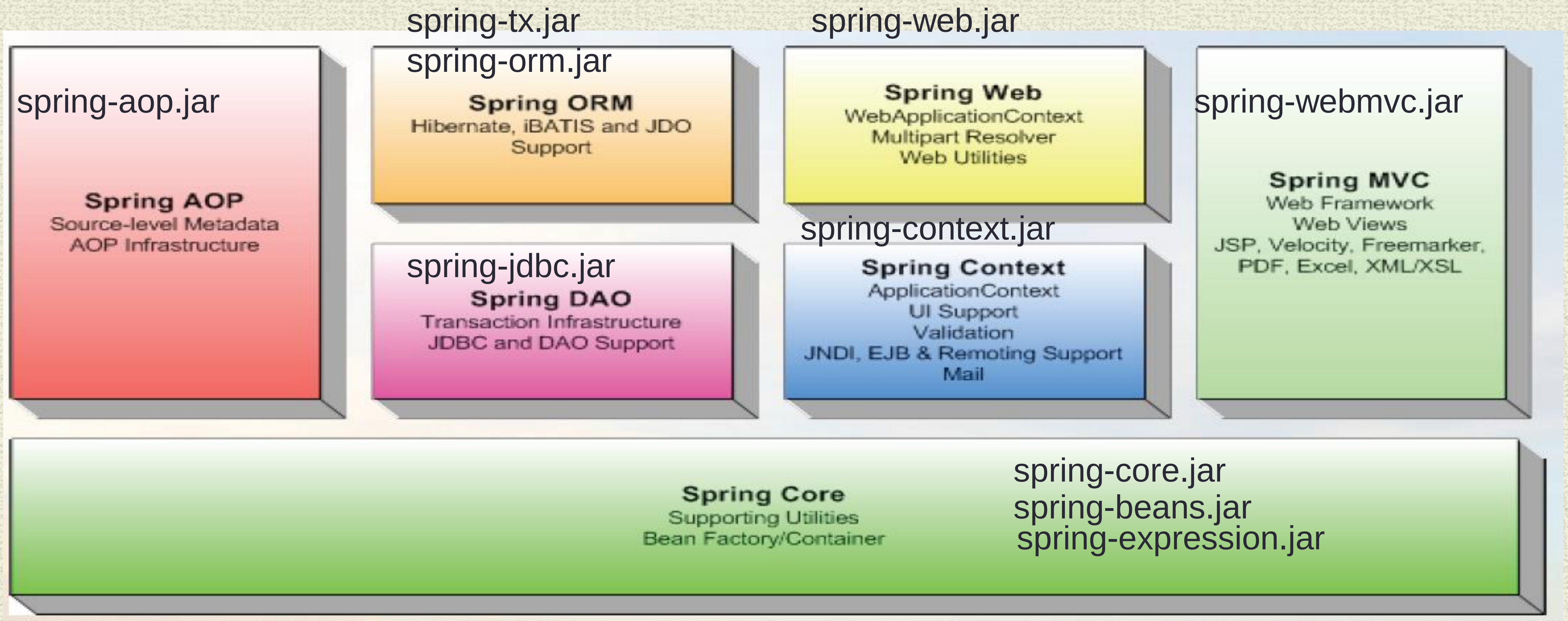
Lesson 3 Spring & Spring MVC Framework

Infinite Diversity Arising from Unity

Spring Architecture



Spring Framework

Infrastructure support for developing Java applications.

Configure disparate components into a fully working application ready for use.

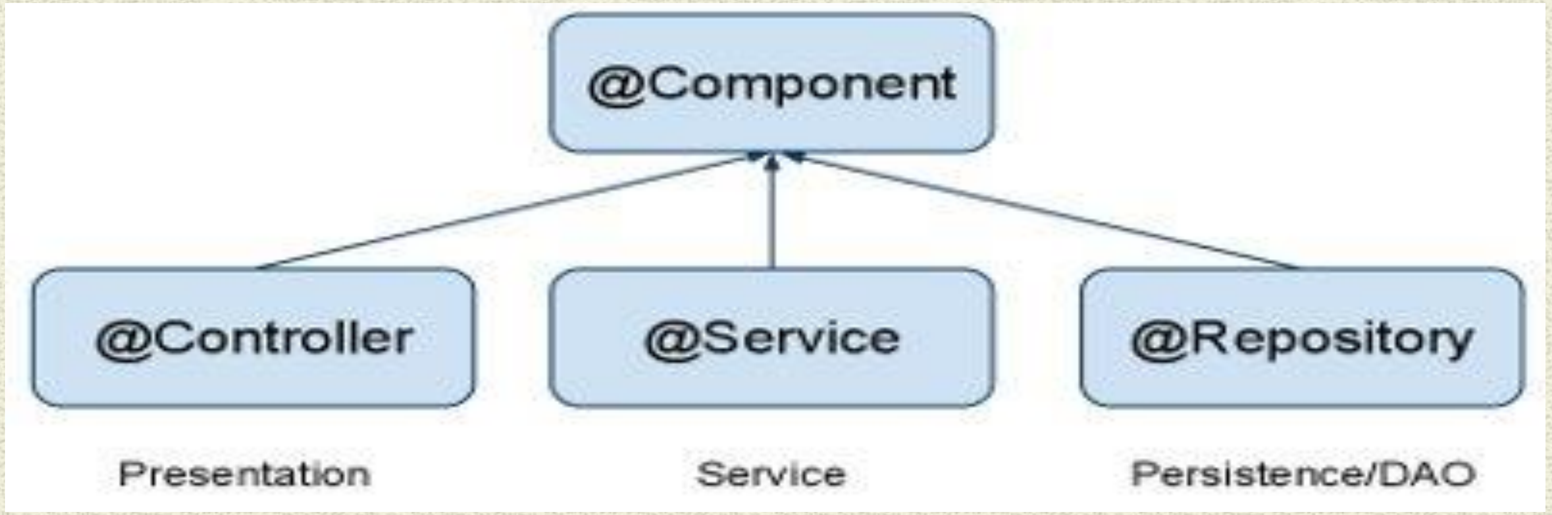
Build applications from “plain old Java objects” (POJOs)

Non-intrusive - domain logic has little or no dependencies on framework

Lightweight application model is that of a layered [N-tier] architecture. Spring 3 Tiers:

1. Presentation objects such as Spring MVC controllers are typically configured in a distinct ***presentation context[tier]***
2. Service objects, business-specific objects, etc. exist in a distinct ***business context[tier]***
3. Data access objects exist in a distinct ***persistence context[tier]***

Backend Components



@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.

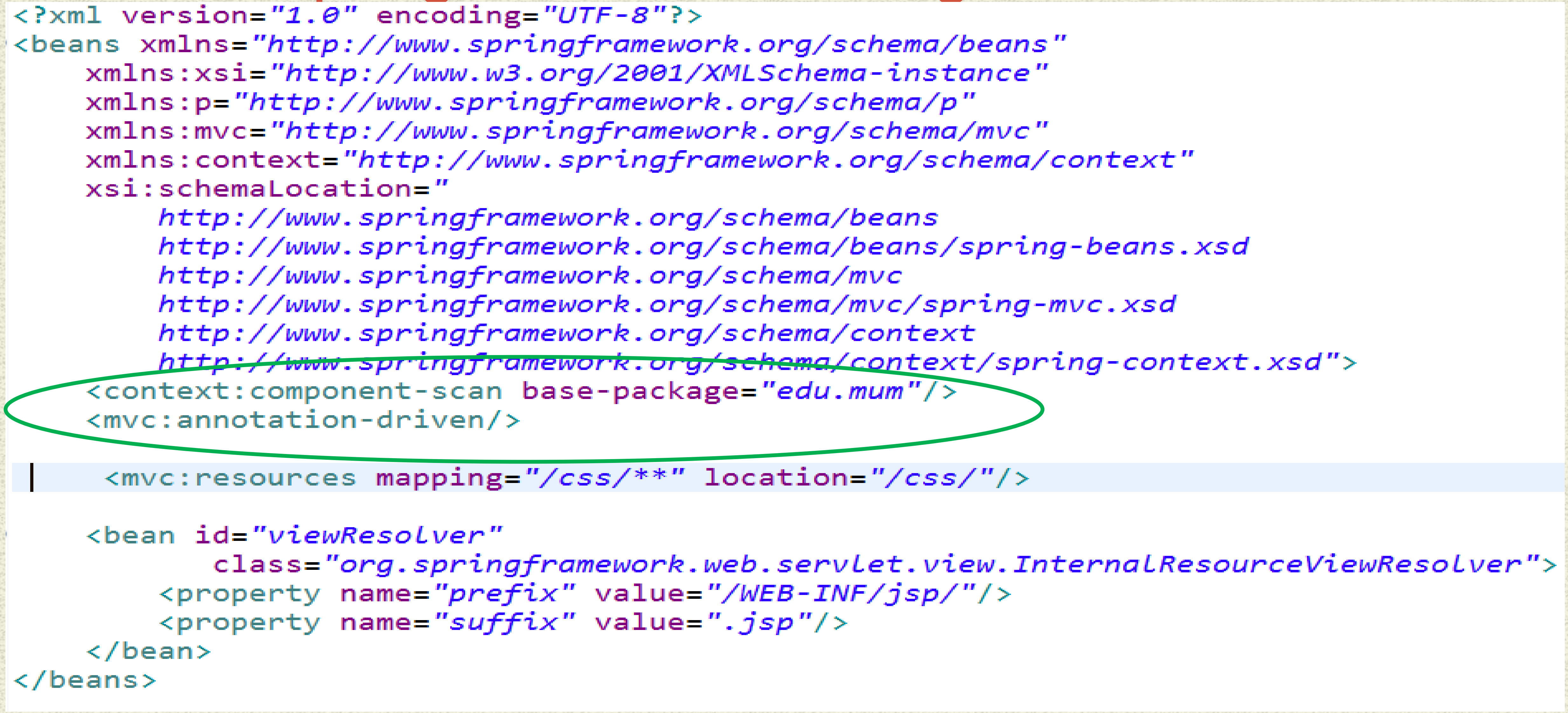
Annotate based on Function

OPTION - annotate all your component classes with @Component

Using @Repository, @Service, and @Controller is:

- Better suited for processing by tools
  - @Repository - automatic translation of exceptions
  - @Controller – rich set of framework functionality
  - @Service – “home” of @Transactions
- More properly suited for associating with aspects
- May carry additional semantics in future releases of the Spring Framework.

Spring MVC XML Configuration File



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

     <mvc:resources mapping="/css/**" location="/css/"/>

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

XML Configuration file – enable annotations

<context:component-scan base-package= “pkg,pkg…" **>**

Scans defined packages to find and register @Component-annotated classes and activate basic annotations[e.g. @Autowired] - within the current application context

<mvc:annotation-driven/>

Enables support for specific annotations [e.g. @RequestMapping, etc.] that are required for Spring MVC to dispatch requests to @Controllers. It is based on MVC XML namespace

<tx:annotation-driven />

Enables support for specific annotations that are required for Spring Transactions @Transaction It is based on transaction XML namespace.

Controller return “view”

View Resolver[s] can simplify view declaration

For example with the view resolver:

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

```
 return "ProductForm";
resolves to:
            /WEB-INF/jsp/ProductForm.jsp
```

```
The subsequent RequestDispatcher forward is done by the framework
```

Spring Configuration Metadata

XML based

- Wire components without touching their source code or recompiling them.
- CLAIM: Annotated classes are no longer POJOs ****
- Configuration centralized and easier to control.

Annotation [Version 2.5]

- Component wiring close to the source
- Shorter and more concise configuration.

JavaConfig [Version 3.0]

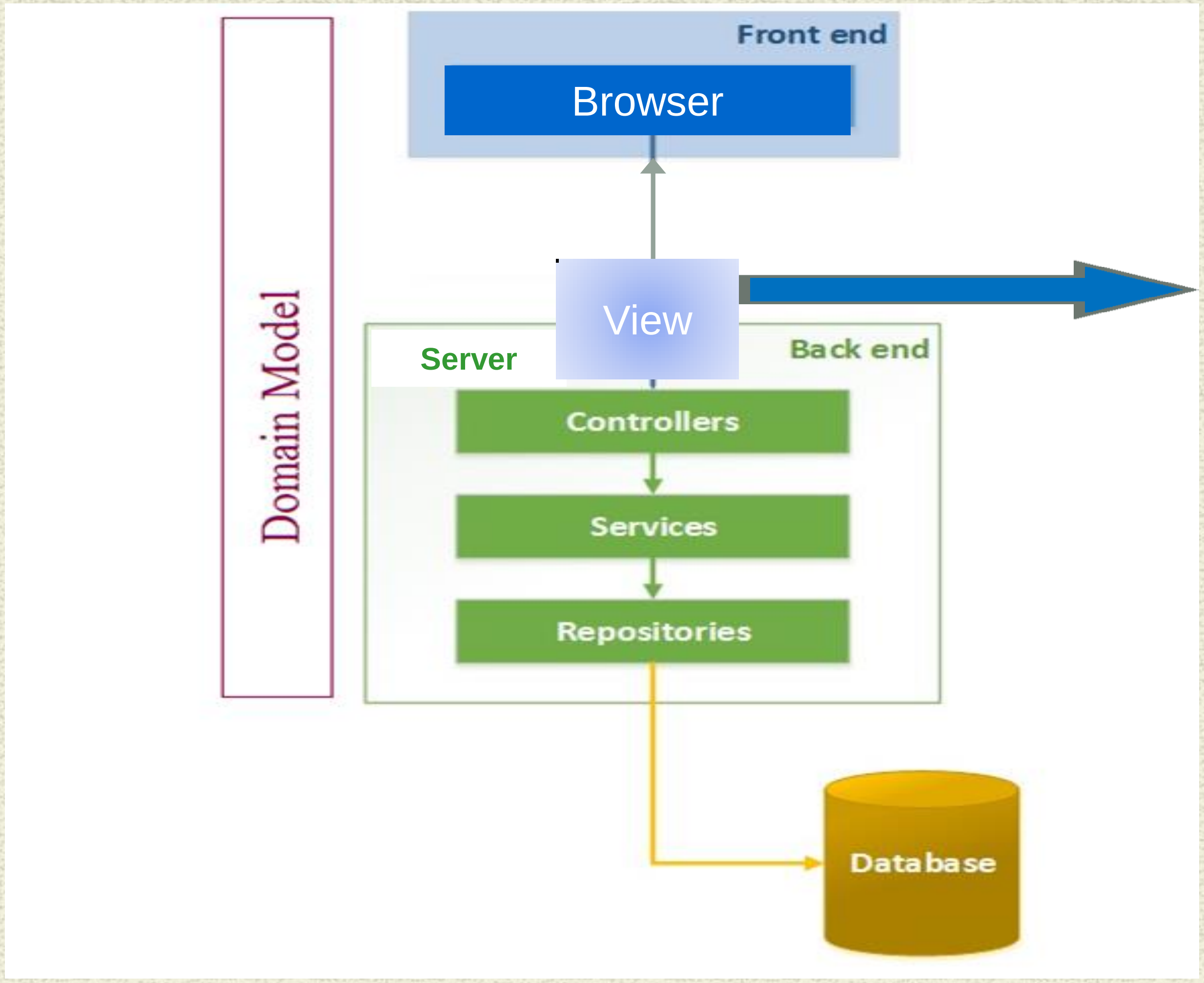
- Define beans external to your application classes by using Java rather than XML files

Annotation injection is performed *before* XML injection. Therefore XML injection takes precedence over Annotation injection. It is the “last word”

JavaConfig Version

```
EnableWebMvc
@Configuration
@ComponentScan(basePackages = { "edu.mum" })
public class Dispatcher extends WebMvcConfigurerAdapter {
public void addResourceHandlers(ResourceHandlerRegistry registry) {
       registry.addResourceHandler("/resources/**")
                    .addResourceLocations("/resources/");
}
@Bean
  public InternalResourceViewResolver jspViewResolver() {
     InternalResourceViewResolver bean =
                     new InternalResourceViewResolver();
        bean.setPrefix("/WEB-INF/jsp/");
        bean.setSuffix(".jsp");
        return bean;
  }
```

Spring Layers – With Spring MVC Layer



VIEWS
JSP
Velocity
FileMarker
ThymeLeaf
PDF
Excel
Web Flow
JSF
XML
JSON

Service Layer

Issue: not whether or not it is needed

BUT

What it contains

Domain Driven Design

- Primary focus - the core domain and domain logic.
- Complex designs based on a model of the domain.
- Collaboration between technical and domain experts to iteratively refine a conceptual model that addresses particular domain problems.

- GOAL: a Rich Domain Model

“Thin” Domain Model

- Contains objects properly named after the nouns in the domain space
- Objects are connected with the rich relationships and structure that true domain models have.

Extreme case: Anemic Domain Model

Little or no behavior – bags of getters and setters.

Service Layer

In a perfect world:

“Thin Layer”

With

“Rich Domain Model”

No business rules or knowledge

Coordinates tasks

Delegates work to domain objects

“The Reality”

Quite often additional **“Domain”** Services exist - populated with “externalized” Business/Logic rules.

Main Point

An N Tier Architecture separates an application into layers thereby supporting a separation of concerns making any application more efficient, flexible and scalable.

Life is structured in layers. It is a structure that is both stable and flexible, consistent yet variable and it encompasses an infinite range of possibilities[scalable]

Spring MVC

Distinct Separation of Concerns

Clearly defined interfaces for role/responsibilities “beyond” Model-View-Controller

Single Central Servlet

- Manages HTTP level request/response
- delegates to defined interfaces

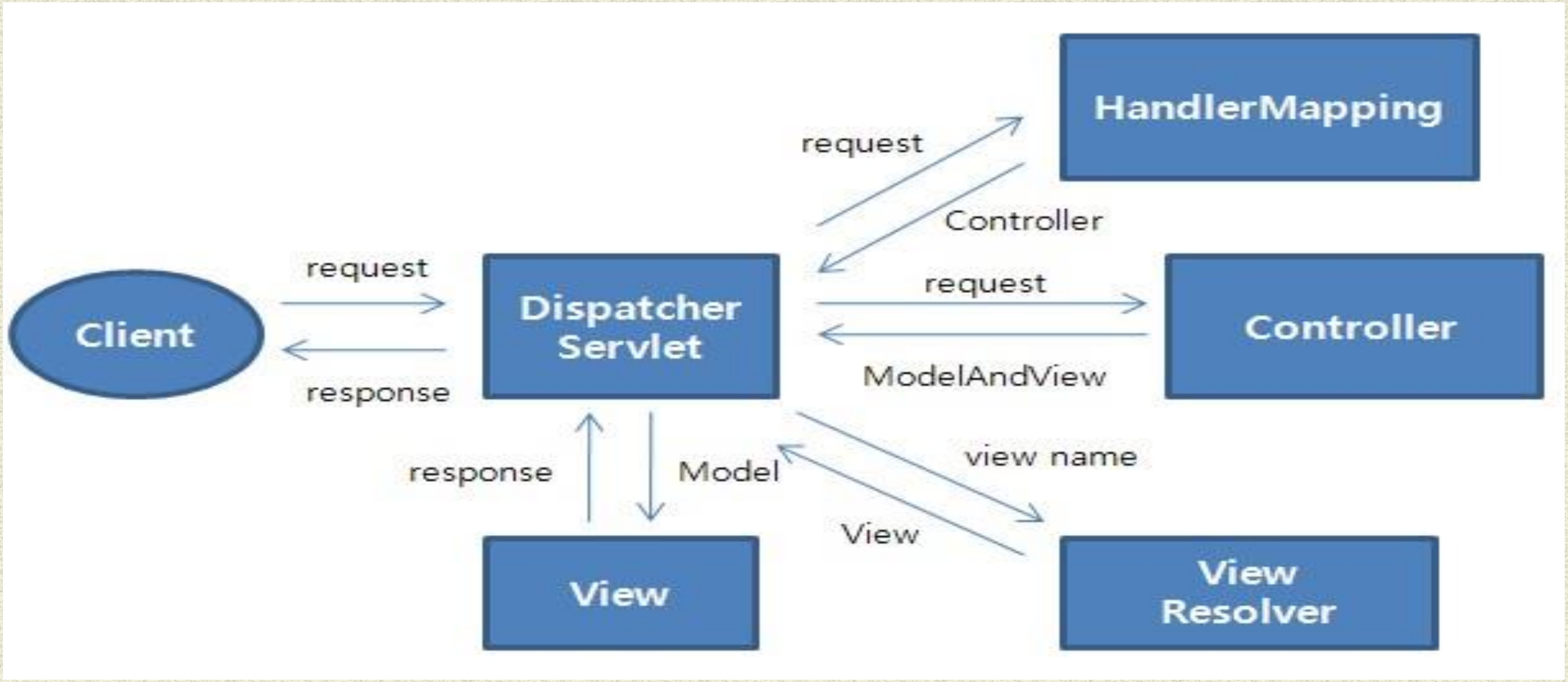
Models integrate/communicate with views

- No need for separate form objects

Views are plug and play

Controllers allowed to be HTTP agnostic

Spring MVC Major Interfaces



Spring MVC Flow

After receiving an HTTP request, *DispatcherServlet* consults the *HandlerMapping* to call the appropriate *Controller*.

The *Controller* takes the request and calls the appropriate service methods based on GET or POST method. The service method will set model data based on defined business logic and return view name to the *DispatcherServlet*.

The *DispatcherServlet* will take help from *ViewResolver* to pickup the defined view for the request.

Once view is finalized, The *DispatcherServlet* passes the model data to the view which is finally rendered on the browser.

Spring MVC DispatcherServlet

Single Central Servlet that dispatches requests to controllers and offers other functionality that facilitates the development of web applications.

Completely integrated with the Spring container

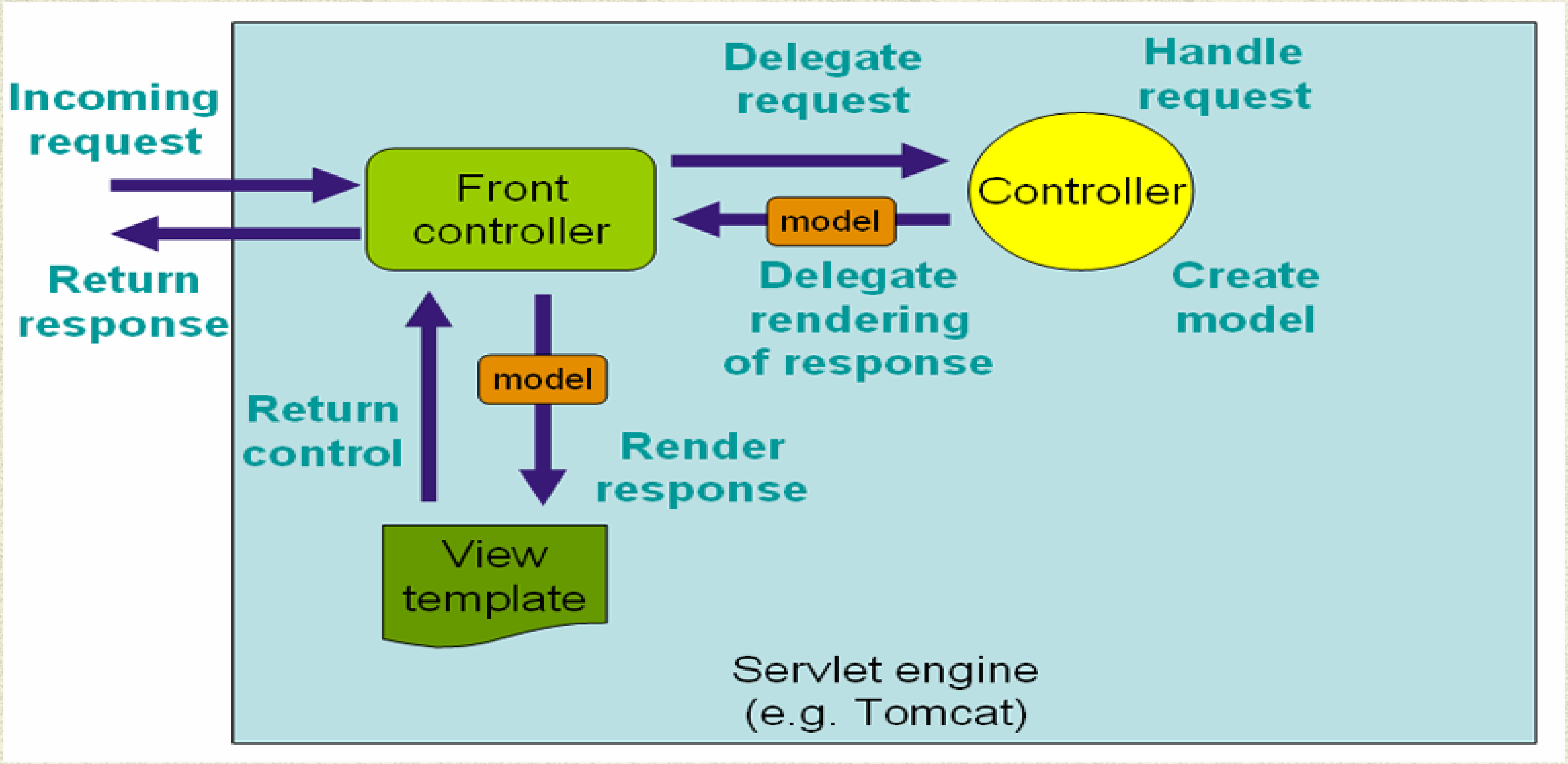
Able to “exploit” Spring framework features

Has a WebApplicationContext, which inherits all the beans already defined in the root WebApplicationContext.

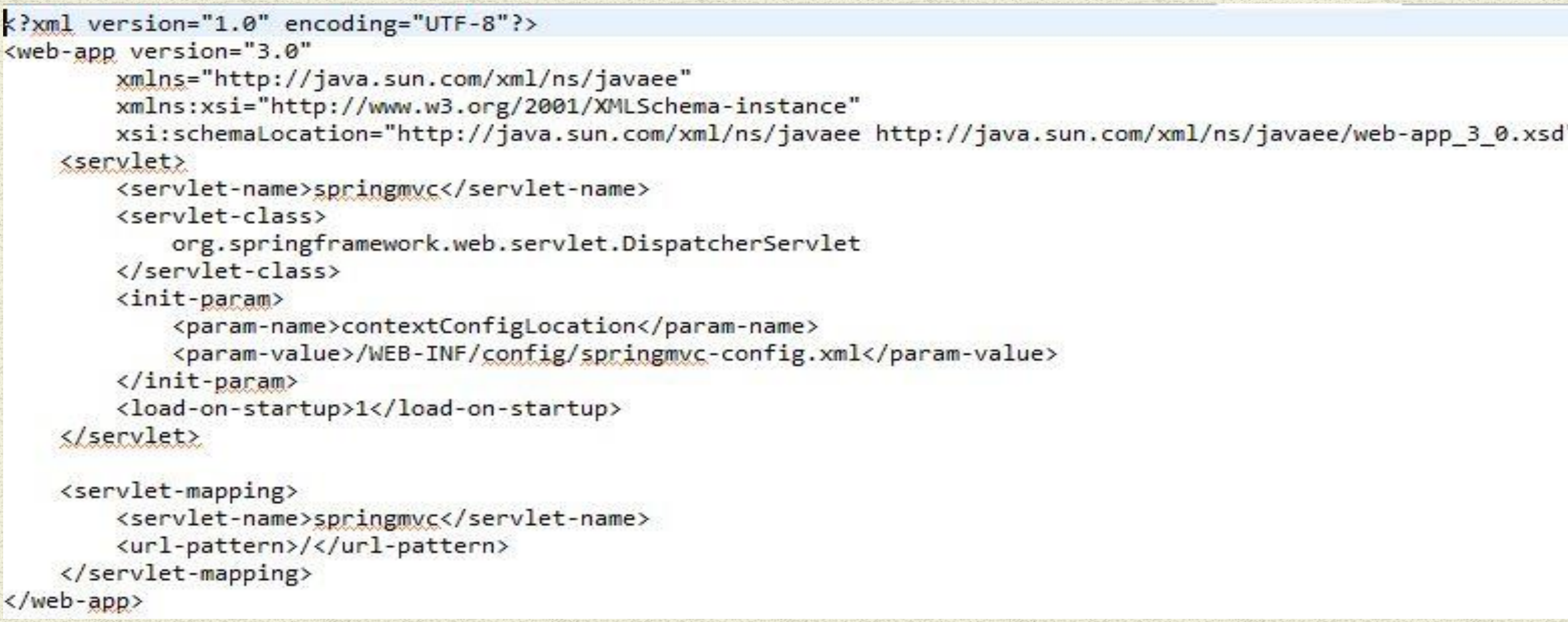
DispatcherServlet - "Front Controller" design pattern

Common pattern used by MVC frameworks

Spring MVC Front Controller



Spring MVC Front Controller Configuration



```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="3.0"
        xmlns="http://java.sun.com/xml/ns/javaee"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd
    <servlet>
        <servlet-name>springmvc</servlet-name>
        <servlet-class>
            org.springframework.web.servlet.DispatcherServlet
        </servlet-class>
        <init-param>
            <param-name>contextConfigLocation</param-name>
            <param-value>/WEB-INF/config/springmvc-config.xml</param-value>
        </init-param>
        <load-on-startup>1</load-on-startup>
    </servlet>

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

Front Controller - JavaConfig Version

```
public class DeploymentDescriptor extends
                        AbstractAnnotationConfigDispatcherServletInitializer {

        protected Class<?>[] getServletConfigClasses()  {
            return new Class[] {Dispatcher.class};
        }

        protected String[] getServletMappings() {
            return new String[] {"/"};
        }

        protected Class<?>[] getRootConfigClasses() {
            return null;
        }
    }
```

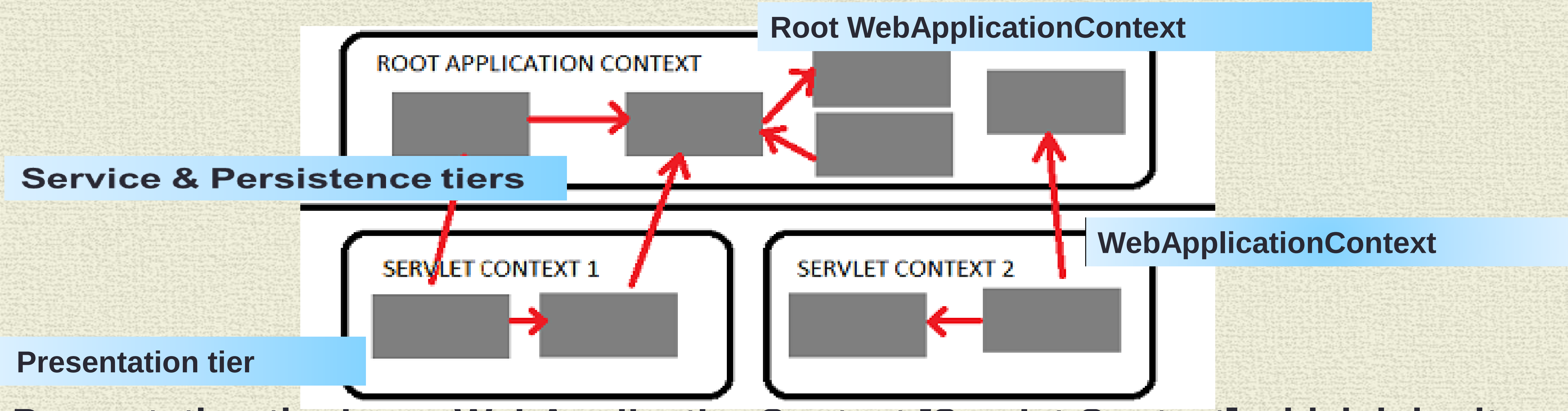
IN pom.xml

```
<plugin>
<groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-war-plugin</artifactId>
    <version>2.1.1</version>
 <configuration>
  <failOnMissingWebXml>false</failOnMissingWebXml>
 </configuration>
</plugin>
```

Web Application Context

Spring has multilevel application context hierarchies.

Web apps by default have two hierarchy levels, root and servlet contexts:



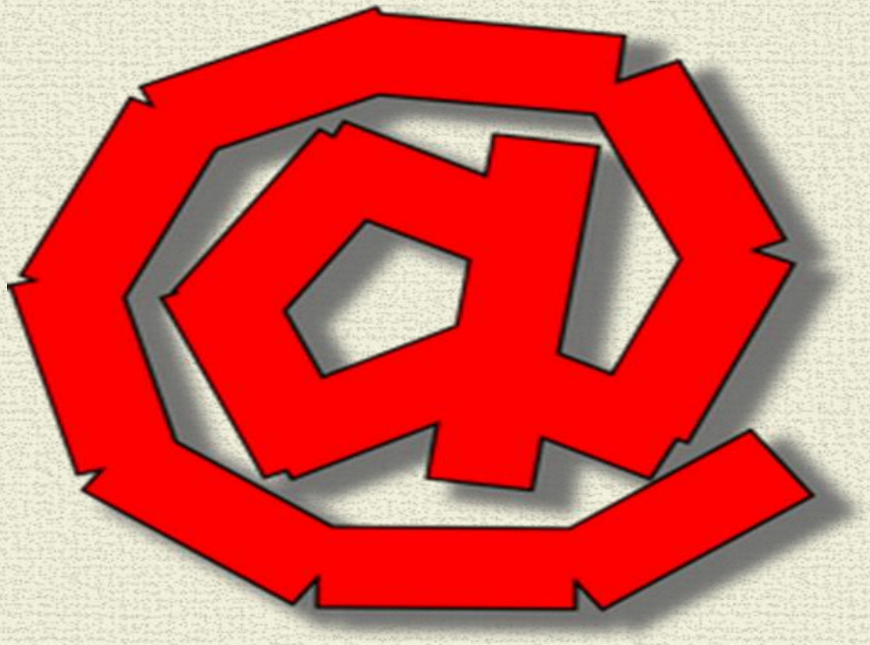
Presentation tier has a WebApplicationContext [Servlet Context] which inherits all the resources already defined in the root WebApplicationContext [Services, Persistence]

Main Point

- The basic ingredients of a Spring MVC application include web pages for the **view** (the known), **domain model** (knower-underlying intelligence), and the Spring Dispatcher Servlet and managed beans as the **controller** to connect the view and model.

Spring MVC Architecture & Annotations

- Handler Mapping
- Spring Annotations
  - @Controller
    - @RequestMapping
    - @ModelAttribute
    - @RequestParam
    - @SessionAttributes



- ViewResolvers
- Views

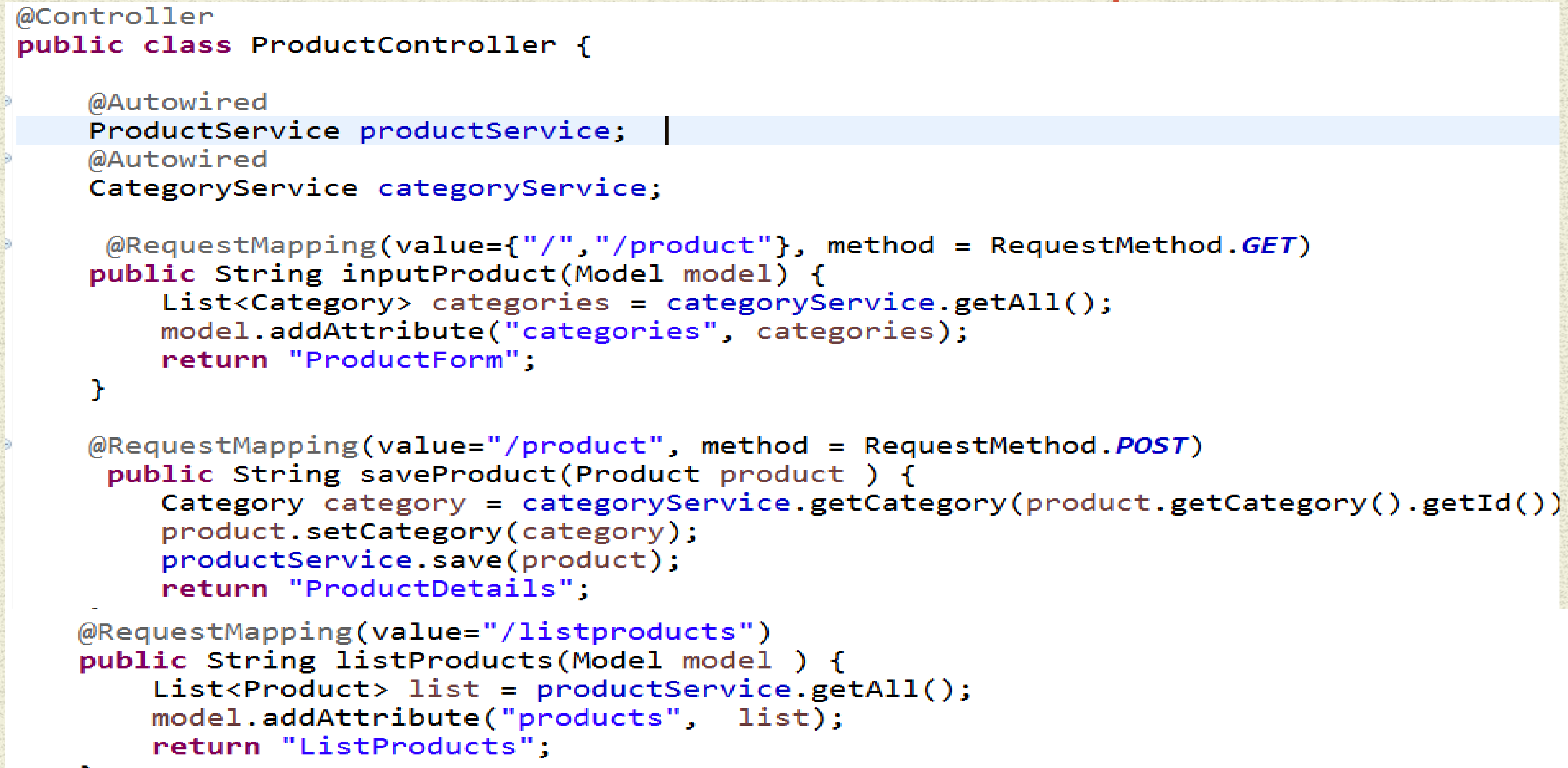
Spring MVC @Controller

Spring implements a controller in a very abstract way, which enables you to create a wide variety of controllers.

Spring Controllers do not extend specific base classes or implement specific interfaces

They do not have direct dependencies on Servlet APIs, although you can easily configure access to Servlet facilities. [actual request, response objects, etc.]

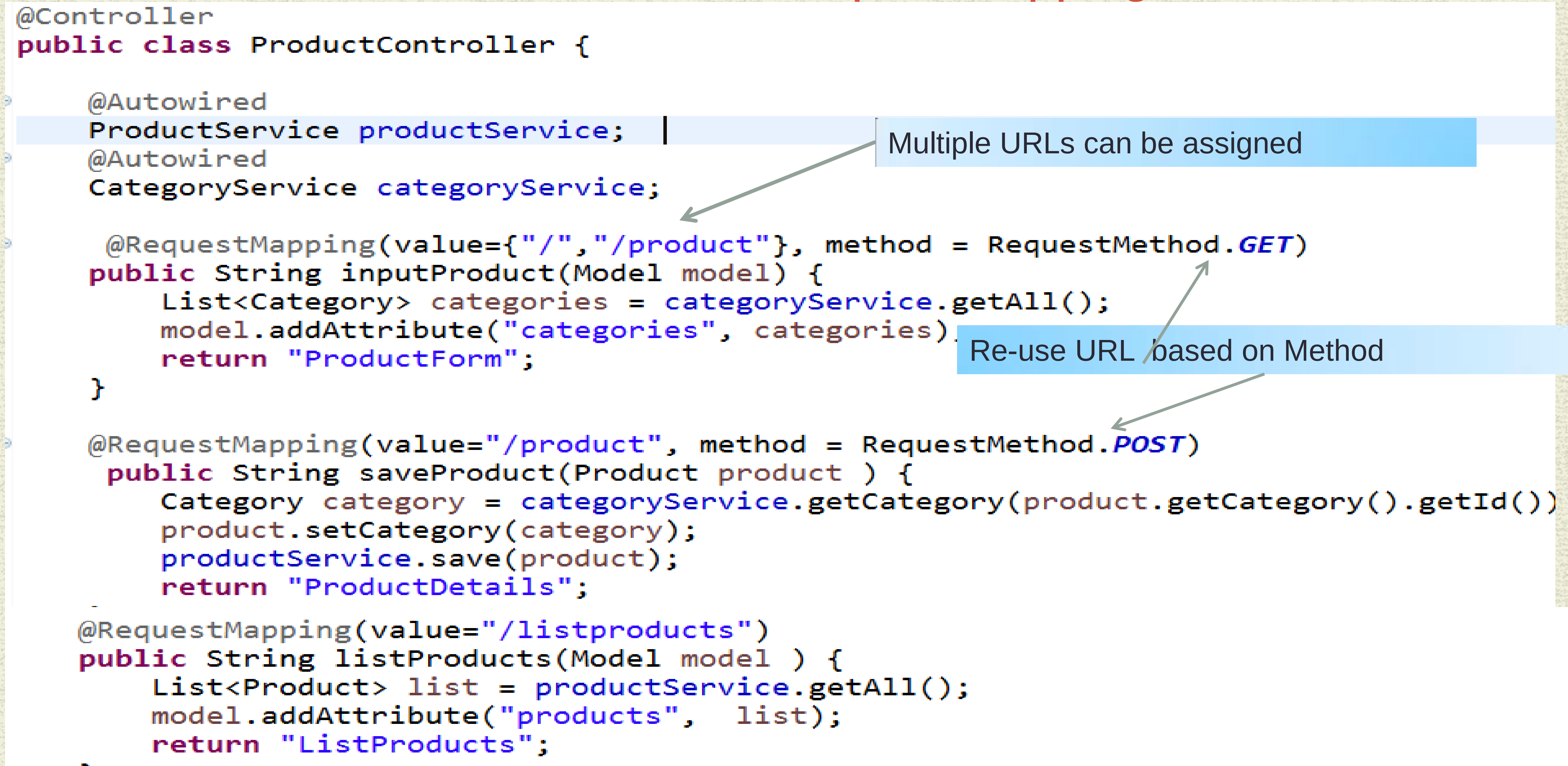
Controller Annotations Example



```
@Controller
public class ProductController {

    @Autowired
    ProductService productService;
    @Autowired
    CategoryService categoryService;

     @RequestMapping(value={"/","/product"}, method = RequestMethod.GET)
    public String inputProduct(Model model) {
        List<Category> categories = categoryService.getAll();
        model.addAttribute("categories", categories);
        return "ProductForm";
    }

    @RequestMapping(value="/product", method = RequestMethod.POST)
     public String saveProduct(Product product ) {
        Category category = categoryService.getCategory(product.getCategory().getId())
        product.setCategory(category);
        productService.save(product);
        return "ProductDetails";
```

```
   @RequestMapping(value="/listproducts")
   public String listProducts(Model model ) {
       List<Product> list = productService.getAll();
       model.addAttribute("products",  list);
       return "ListProducts";
```

Method level @RequestMapping

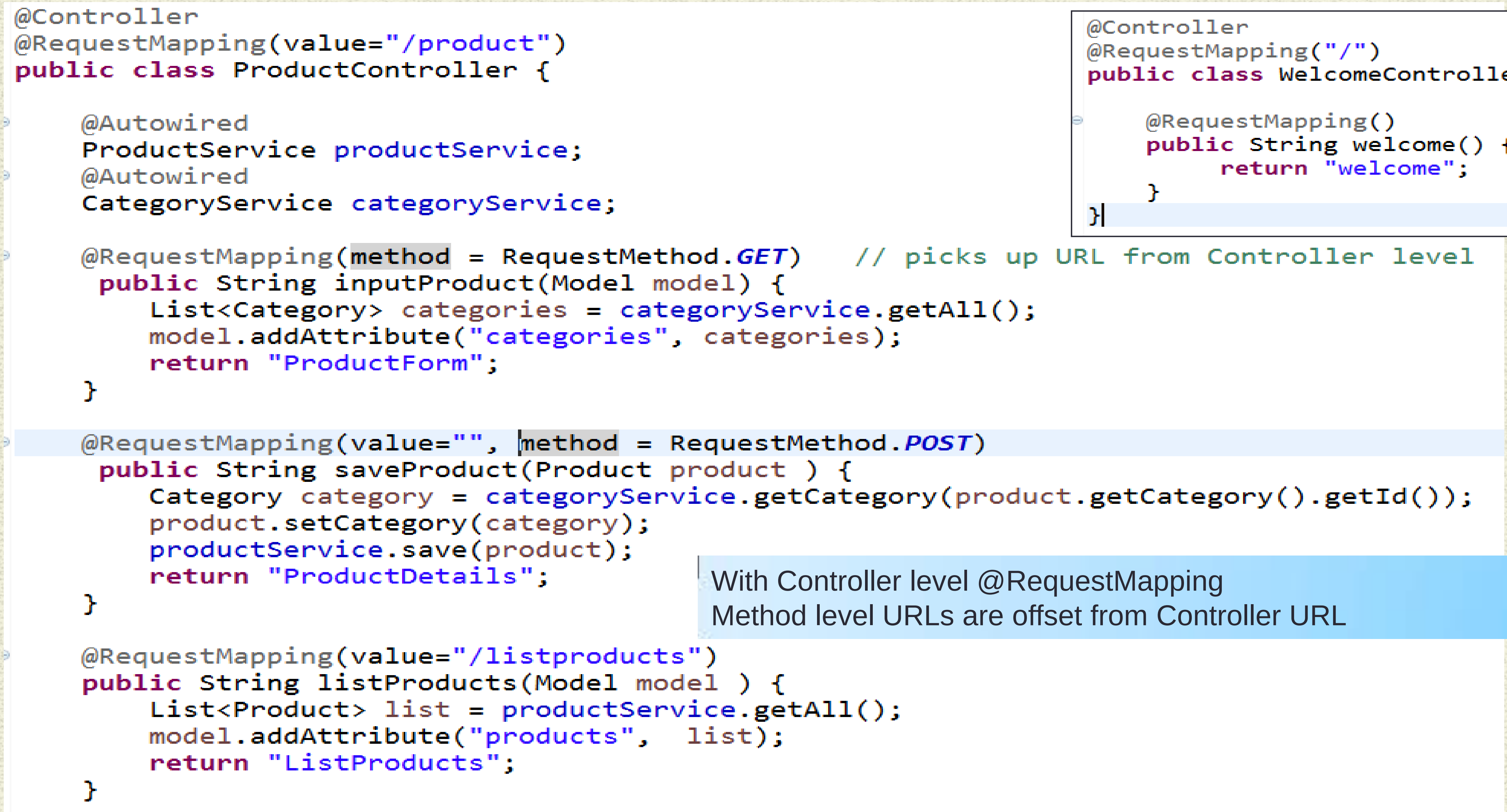


```
@Controller
public class ProductController {

    @Autowired
    ProductService productService;
    @Autowired
    CategoryService categoryService;

     @RequestMapping(value={"/","/product"}, method = RequestMethod.GET)
    public String inputProduct(Model model) {
        List<Category> categories = categoryService.getAll();
        model.addAttribute("categories", categories);
        return "ProductForm";
    }

    @RequestMapping(value="/product", method = RequestMethod.POST)
     public String saveProduct(Product product ) {
        Category category = categoryService.getCategory(product.getCategory().getId())
        product.setCategory(category);
        productService.save(product);
        return "ProductDetails";

   @RequestMapping(value="/listproducts")
   public String listProducts(Model model ) {
       List<Product> list = productService.getAll();
       model.addAttribute("products",  list);
       return "ListProducts";
```

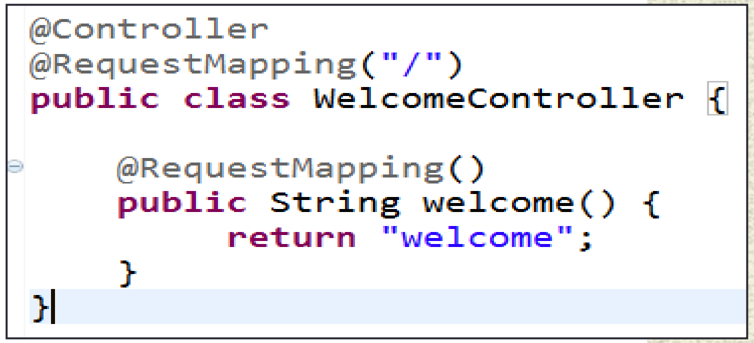
Multiple URLs can be assigned

Re-use URL based on Method

Class level @RequestMapping



```
@Controller
@RequestMapping(value="/product")
public class ProductController {

    @Autowired
    ProductService productService;
    @Autowired
    CategoryService categoryService;

    @RequestMapping(method = RequestMethod.GET)   // picks up URL from Controller level
     public String inputProduct(Model model) {
        List<Category> categories = categoryService.getAll();
        model.addAttribute("categories", categories);
        return "ProductForm";
    }

    @RequestMapping(value="", method = RequestMethod.POST)
     public String saveProduct(Product product ) {
        Category category = categoryService.getCategory(product.getCategory().getId());
        product.setCategory(category);
        productService.save(product);
        return "ProductDetails";
    }

    @RequestMapping(value="/listproducts")
    public String listProducts(Model model ) {
        List<Product> list = productService.getAll();
        model.addAttribute("products",  list);
        return "ListProducts";
    }
```



```
@Controller
@RequestMapping("/")
public class WelcomeController {

    @RequestMapping()
    public String welcome() {
        return "welcome";
    }
}
```

With Controller level @RequestMapping
Method level URLs are offset from Controller URL

@RequestParam

Placed on Method argument

http://localhost:8080/webstore/products/product?id=P1234

```
public String getProductById(@RequestParam("id")String productId,Model model) {
model.addAttribute("product", productService.getProductById(productId));
}
```

Handling multiple values [e.g., multiple selection list]

http://localhost:8091/store/sizechoices?sizes=small&sizes=medium&sizes=large

```
  public String getSizes(@RequestParam(“sizes")String sizeArray[]
```

@RequestMapping Template with @PathVariable

Facility to pass resource request as part of URL INSTEAD of as a @RequestParam

Conforms to RESTful service syntax

http://localhost:8080/webstore/products/Laptop

```
@RequestMapping("/{category}")
public String getProductsByCategory(@PathVariable("category") String category) {
       productService.getProductsByCategory(category));
       return "products";
}
```

@PathVariable is used in conjunction with @RequestMapping URL template.
In this case it is a means to get the category string passed in the method signature.

The @PathVariable param needs to be the same as the param in the @RequestMapping

Data Binding

Automatically maps request parameters domain objects

Simplifies code by removing repetitive tasks

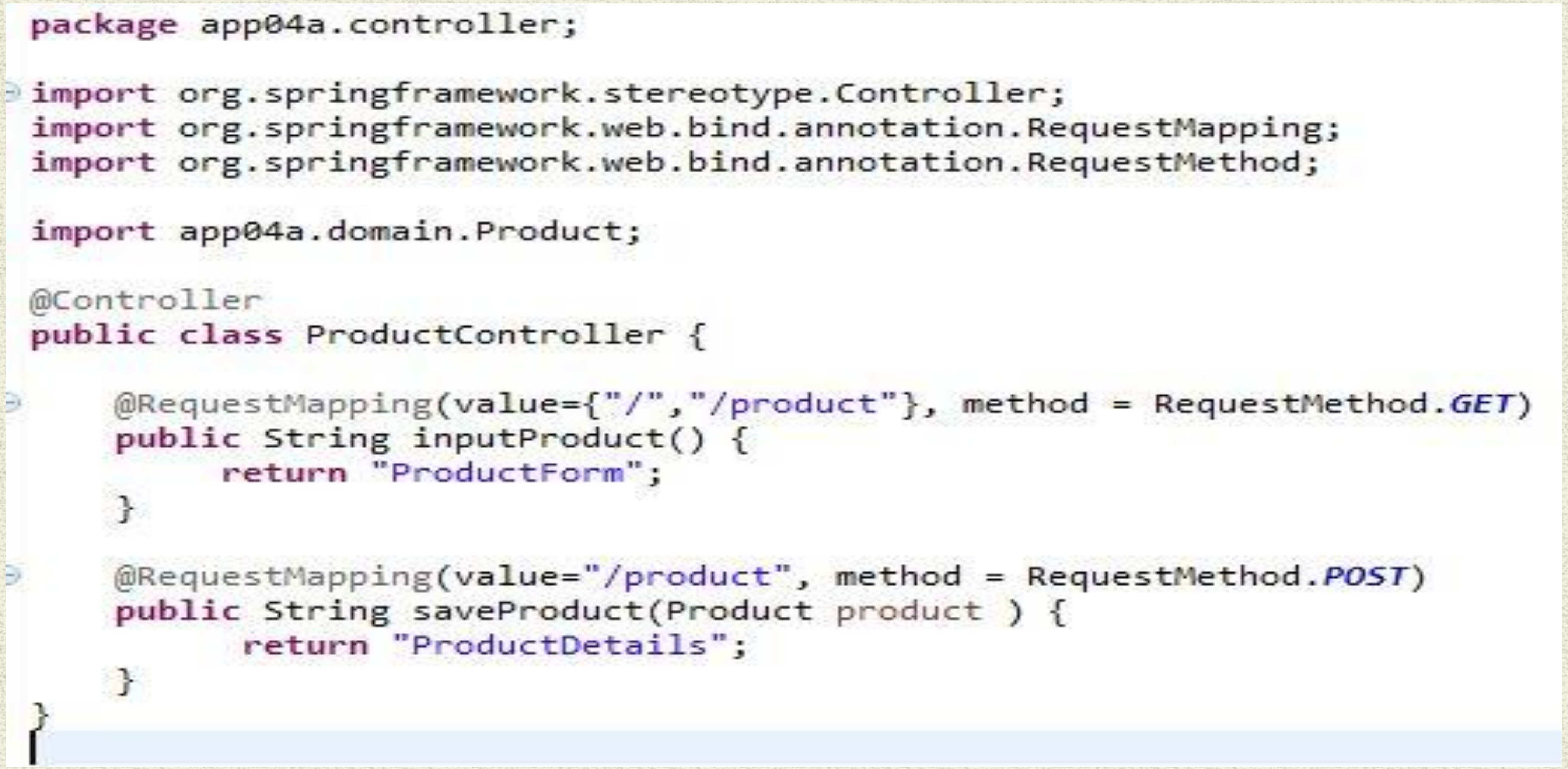
Built-in Data Binding handles simple String to data type conversions

HTTP request parameters [String types] are converted to model object properties of varying data types.

Does NOT handle COMPLEX data types; that requires custom formatters

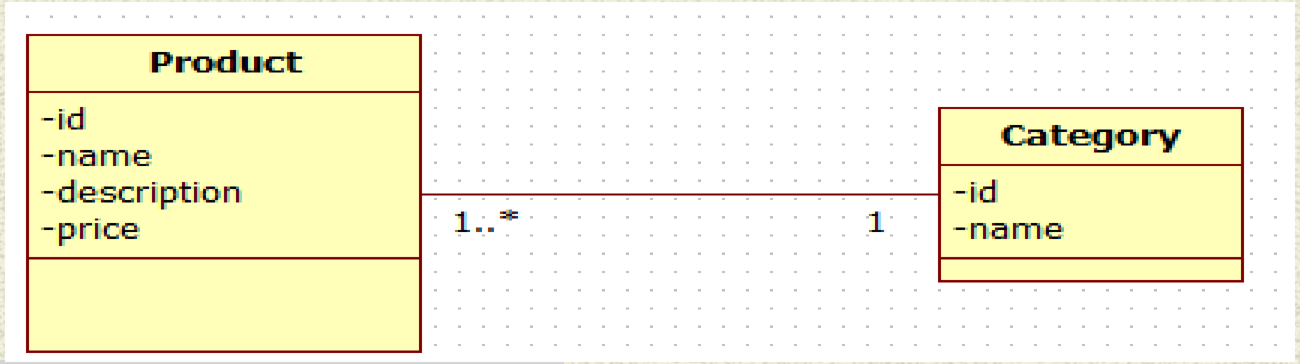
Does handle complex nested relationships

Data Binding example



```java
package app04a.controller;

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

import app04a.domain.Product;

@Controller
public class ProductController {

    @RequestMapping(value={"/","/product"}, method = RequestMethod.GET)
    public String inputProduct() {
        return "ProductForm";
    }

    @RequestMapping(value="/product", method = RequestMethod.POST)
    public String saveProduct(Product product ) {
         return "ProductDetails";
    }
}
```

Data Binding - Relationships



```
<form action="product" method="post">
     <legend>Add a product</legend>
        <p>
     <label for="category">Category </label>
       <select name="category.id">
               <option value="-">--Select Category--</option>
           <c:forEach var="category" items="${categories}">
               <option value="${category.id}" > ${category.name}</option>
           </c:forEach>
        </select>

      <label for="name">Product Name: </label>
         <input type="text" id="name" name="name" >
```

NOTE: Still need to access ENTIRE category object in controller…

Multiple[4] ways to set Request Attributes

(1)

```
public String inputProduct( Product product) {
```

Product created
& added to model

(2)

```
public String inputProduct( Model model) {
        Product product = new Product();
        model.addAttribute(“product", product);
```

Added to `HttpServletRequest` upon method finish

(3)

```
 public String inputProduct( Model model) {
        Product product = new Product();
         model.addAttribute(product);
```

Like “2” except:
defaults key to Class name
with lowercase first letter

(4)

```
public String inputProduct(HttpServletRequest request ) {
        Product product = new Product();
        request.setAttribute(“product", product);
```

Never added to model

Inversion of Control [IOC]

Objects do not create other objects that they depend on.

Promotes loose coupling between classes and subsystems
Adds potential flexibility to a codebase for future changes.
Classes are easier to unit test in isolation.
Enable better code reuse.

IOC is implemented using **Dependency Injection**(DI).

Dependency Injection [DI]

DI exists in three major variants

Dependencies defined through:

- Property-based dependency injection.
- Setter-based dependency injection.
- Constructor-based dependency injection

Container *injects* dependencies when it creates the bean.

Dependency Injection examples

Property based[byType]:

```
   @Autowired
   ProductService productService;
```

Setter based[byName]:

```
   ProductService productService;
   @Autowired
   public void setProductService(ProductService productService){
            this.productService = productService;
```

Constructor based:

```
    ProductService productService;
    @Autowired
 public ProductController(ProductService productService) {
              this.productService = productService;
```

Main Point

The use of Inversion of Control simplifies a business application by delegating the responsibility for managing needed resources

Part of the process of Transcending is letting the physiology ***naturally*** *manage the rest it needs.*

Part of the process of Transcending is letting the physiology to
naturally manage the rest it needs.

DEMO

Product as a JSP

[ProductJSP]

Product as a Controller

[ProductMVCMethod,ProductMVCClass]

@PathVariable;@RequestParam

webstore3