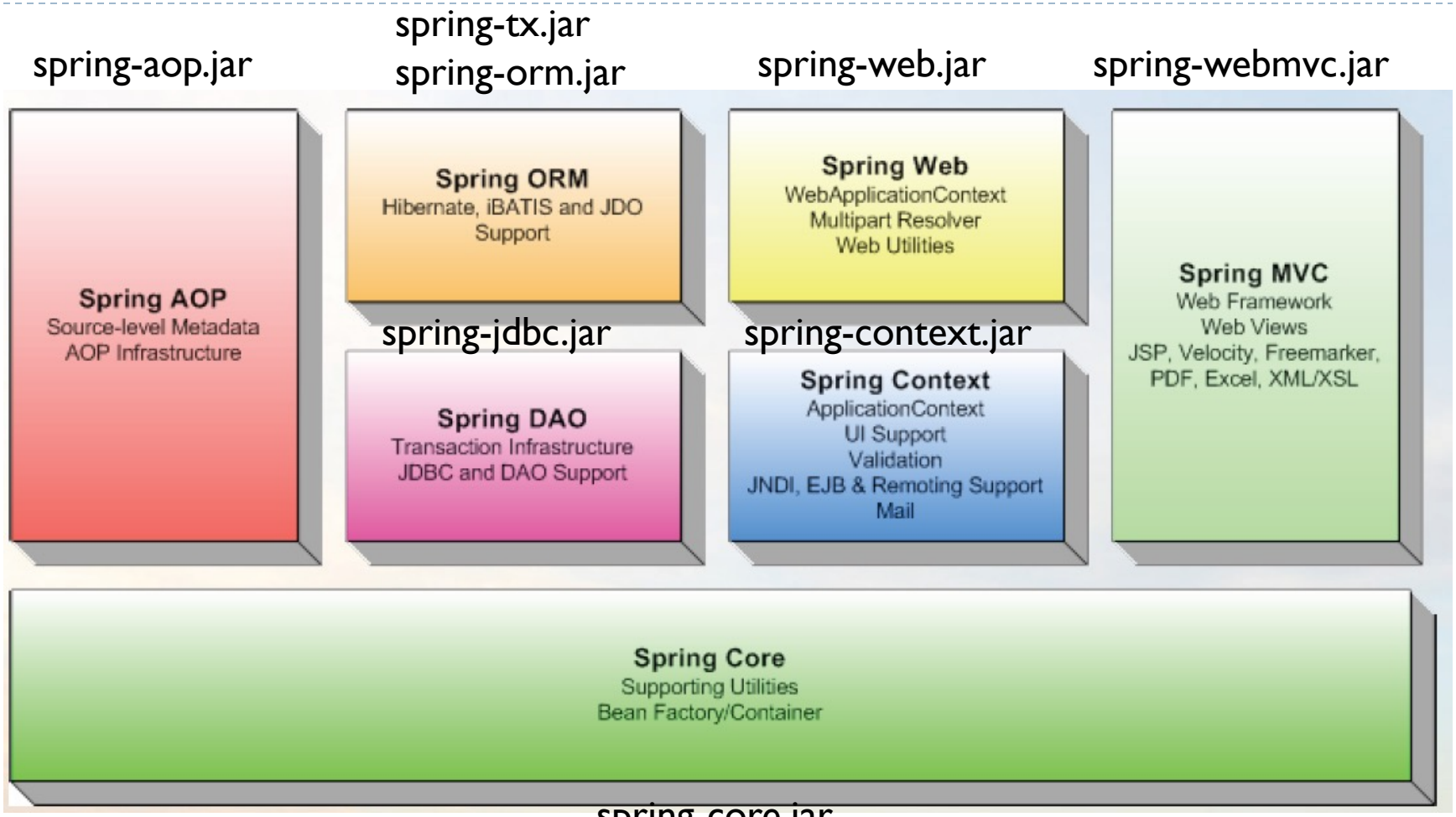
Spring & Spring MVC Framework

Infinite Diversity Arising from Unity

Spring Architecture



https://docs.spring.io/spring/docs/current/javadoc-api/overview-summary.html

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]***

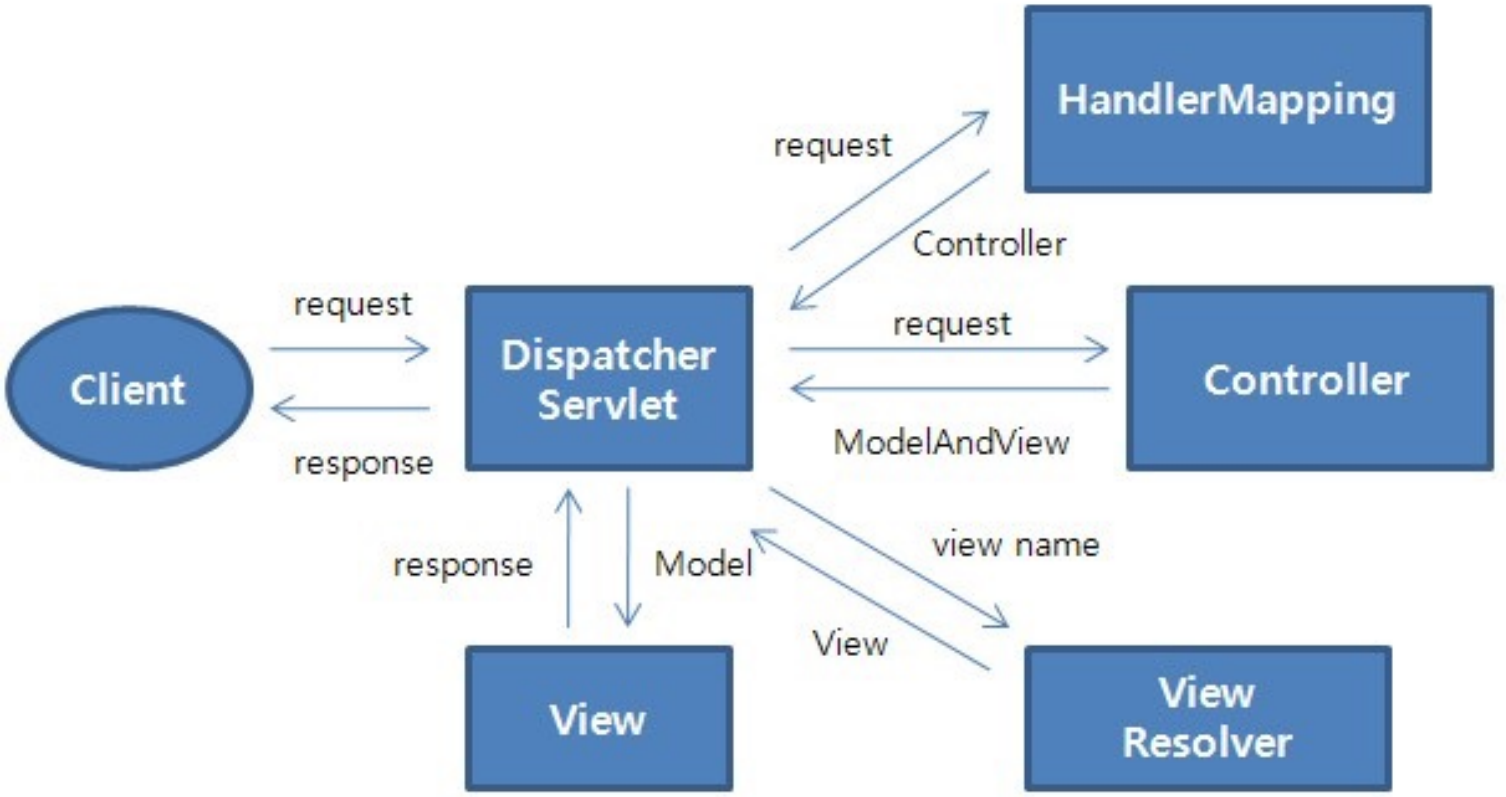
JavaBean vs POJO vs Spring Bean

- JavaBean
  - Adhere to Sun's JavaBeans specification
  - Implements Serializable interface
  - Must have default constructor, setters & getters
  - Reusable Java classes for visual application composition
- POJO
  - 'Fancy' way to describe ordinary Java Objects
  - Doesn't require a framework
  - Doesn't require an application server environment
  - Simpler, lightweight compared to 'heavyweight' EJBs
- Spring Bean
  - Spring managed - configured, instantiated and injected
- A Java object can be a JavaBean, a POJO and a Spring bean all at the same time.

Spring MVC

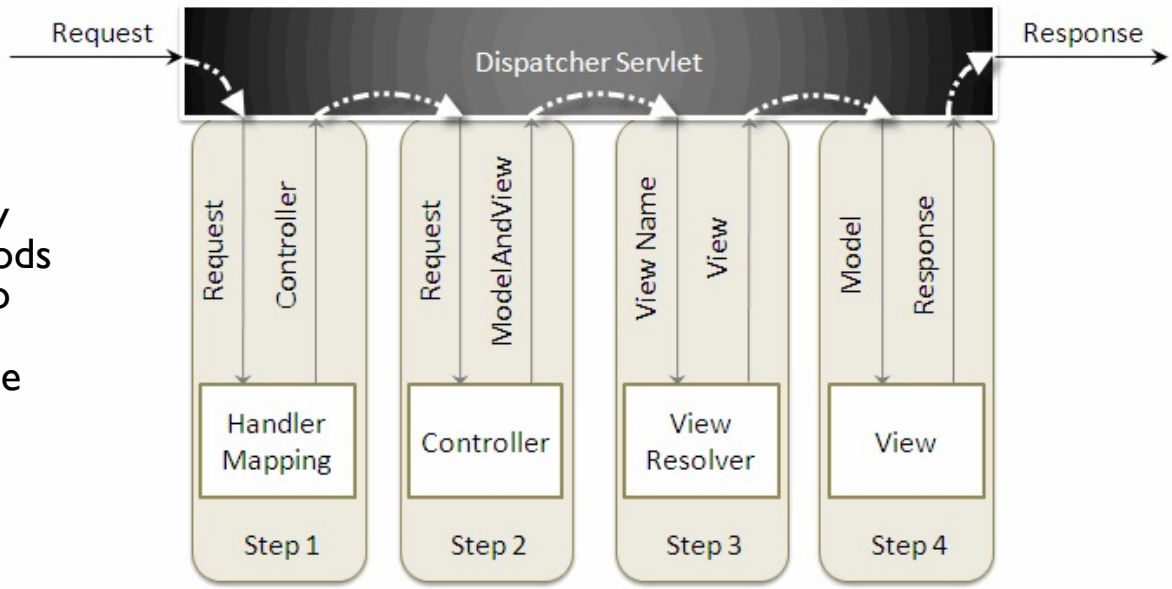
- Distinct Separation of Concerns
- Cleary defined interfaces for role/responsibilities "beyond" Model-View-Controller
- Single Central Servlet
  - Manages HTTP level request/response
  - delegates to defined interfaces
- Models integrate/communicate with views
  - No need for separate form objects
- Views are plug and play
- Controllers allowed to be HTTP agnostic

Spring MVC Major Interfaces

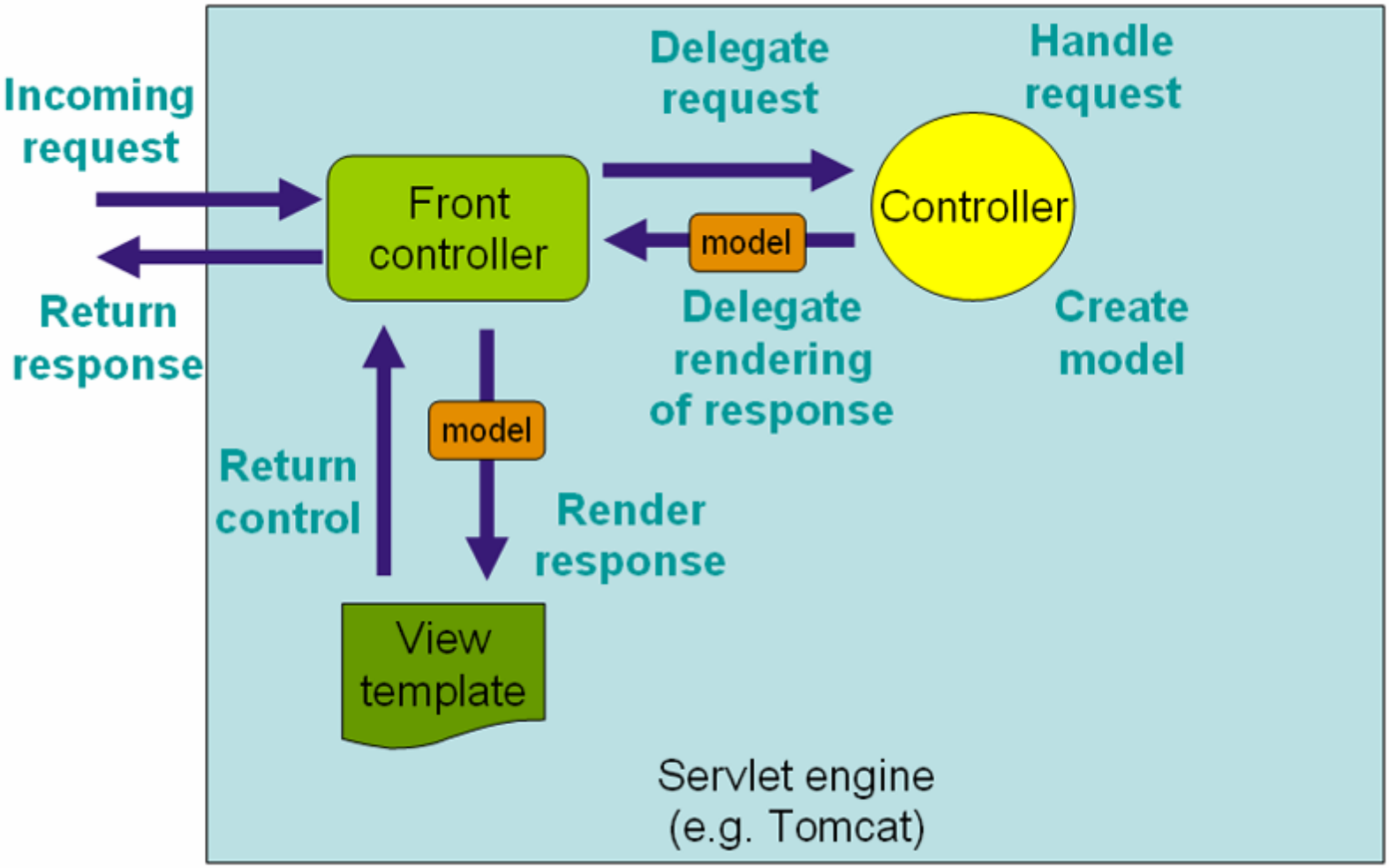


Spring MVC Flow

- The *DispatcherServlet* first receives the request.
- The *DispatcherServlet* consults the *HandlerMapping* and invokes the *Controller* associated with the request.
- The *Controller* process the request by calling the appropriate service methods and returns a *ModeAndView* object to the *DispatcherServlet*.
  The *ModeAndView* object contains the model data and the view name.
- The *DispatcherServlet* sends the view name to a *ViewResolver* to find the actual *View* to invoke.
- Now the *DispatcherServlet* will pass the model object to the *View* to render the result.
- The *View* with the help of the model data will render the result back to the user.



Spring MVC Front Controller



JavaConfig Version

```
public class DispatcherServletInitializer extends
    AbstractAnnotationConfigDispatcherServletInitializer {

      @Override
      protected Class<?>[] getRootConfigClasses() {
      return new Class[] { RootApplicationContextConfig.class };
     }

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

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

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 ApplicationContext.
- DispatcherServlet - "Front Controller" design pattern
  - Common pattern used by MVC frameworks

Spring Configuration Metadata

- **XML based**
  - Wire components without touching their source code or recompiling them.
  - 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

Configuration annotations

- **@Configuration**
- is a class-level annotation indicating that an object is a source of bean definitions. @Configuration classes declare beans via public @Bean annotated methods.
- **@EnableWebMVC**
- To enable MVC Java config add the annotation @EnableWebMvc to one of your @Configuration classes
- **@ComponentScan**
- To autodetect these classes and register the corresponding beans, you need to add @ComponentScan to your @Configuration class, where the basePackages attribute is a common parent package for the two classes.

JavaConfig Version

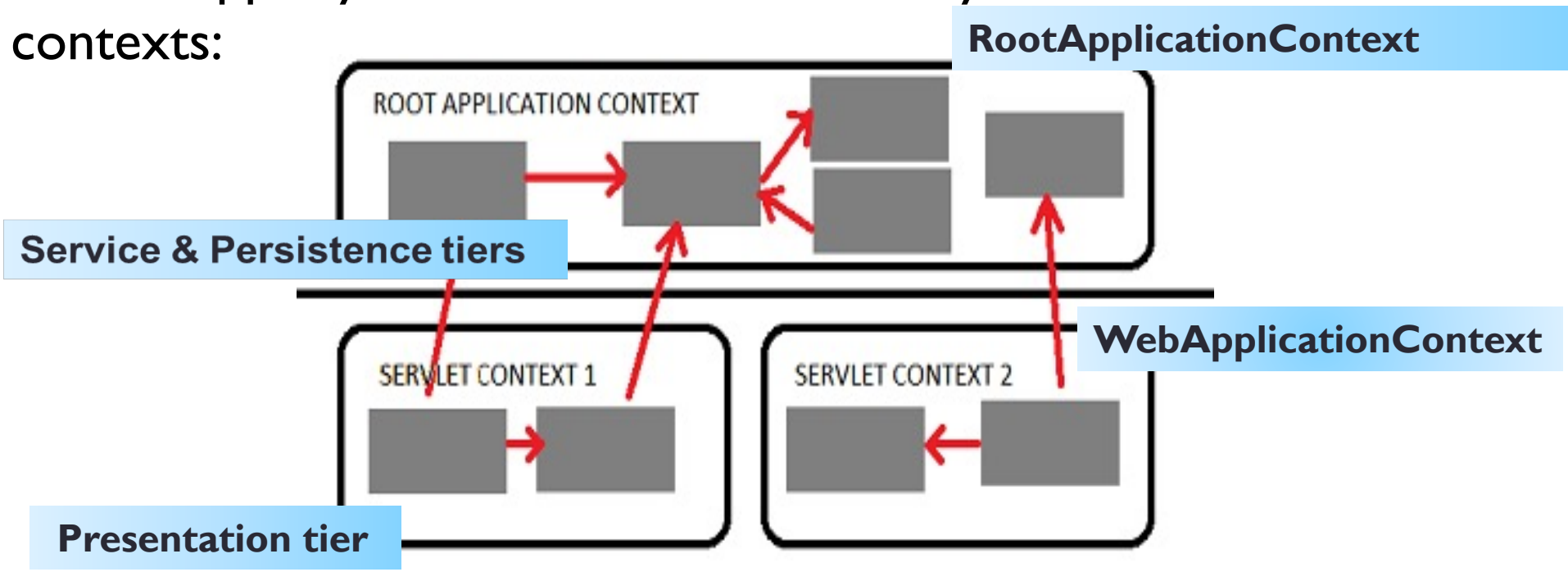
```
@Configuration
@EnableWebMvc
@ComponentScan("com.packt.webstore")
public class WebApplicationContextConfig implements WebMvcConfigurerAdapter {
    @Override
    public void configureDefaultServletHandling(DefaultServletHandlerConfigurer configurer) {
        configurer.enable();
    }

    @Bean
    public InternalResourceViewResolver getInternalResourceViewResolver() {
        InternalResourceViewResolver resolver = new InternalResourceViewResolver();
        resolver.setViewClass(JstlView.class);
        resolver.setPrefix("/WEB-INF/views/");
        resolver.setSuffix(".jsp");
        return resolver;
    }
}
```

- `@Configuration` This indicates that a class declares one or more `@Bean` methods.
- `@EnableWebMvc` imports some special Spring MVC configuration
- `@ComponentScan("com.packt.webstore")` This specifies the base packages to scan for annotated components(beans)

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 ApplicationContext [Services, Persistence]

Controller return “view”

- View Resolver[s] can simplify view declaration
- For example with the view resolver:

```
@Bean
public InternalResourceViewResolver getInternalResourceViewResolver() {
    InternalResourceViewResolver resolver = new InternalResourceViewResolver();
    resolver.setViewClass(JstlView.class);
    resolver.setPrefix("/WEB-INF/views/");
    resolver.setSuffix(".jsp");

    return resolver;
}
```

- WelcomeController.java

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

resolves to: /WEB-INF/jsp/welcome.jsp

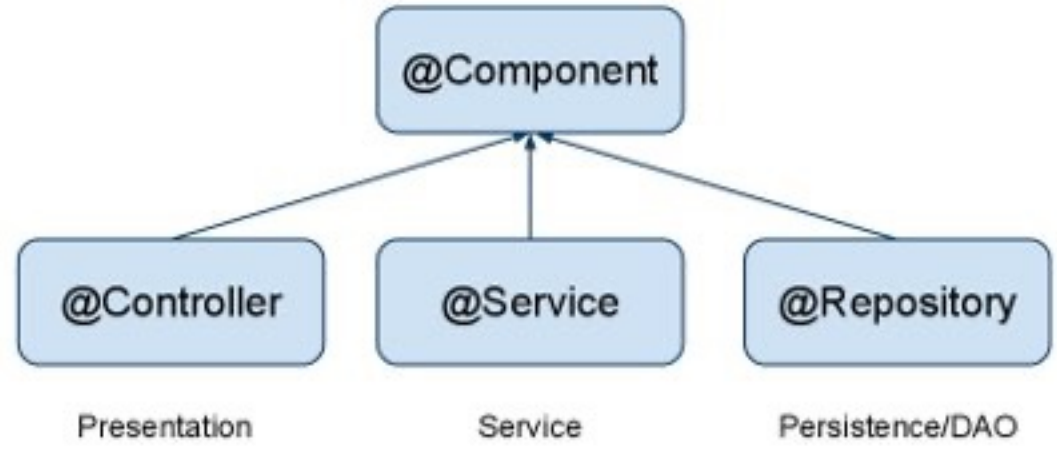
The subsequent RequestDispatcher forward is done by the framework

Main Point

- The basic ingredients of a Spring MVC application include web pages for the view (the known), the back end domain logic and data (knower or underlying intelligence), and the Spring framework and Dispatcher Servlet and managed beans as the controller to connect the view and model.
- *Knowledge is the wholeness of knower, known, and process of knowing.*

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 `@Transactional`
  - More properly suited for associating with aspects
  - May carry additional semantics in future releases of the Spring Framework.



Spring MVC Annotations

- Handler Mapping
- Spring Annotations
  - @Controller
  - @RequestMapping (@GetMapping @PostMapping)
  - @RequestParam
  - @PathVariable
  - … others that will come up in next chapters
- 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 Annotation Example

```
@RequestMapping("/product")
@Controller
public class ProductController {
    @RequestMapping
    public String getAddProductForm(Model model) {
        model.addAttribute("categories", categoryService.getAll());
        return "ProductForm";
    }
    @RequestMapping(value="/add", method = RequestMethod.GET)
    public String getProductForm(Model model) {
        model.addAttribute("categories", categoryService.getAll());
        return "ProductForm";
    }
    @RequestMapping(value="/add", method = RequestMethod.POST)
    public String saveProduct(Product product) {
        Category category = categoryService.getCategory(product.getCategory
        product.setCategory(category);
        productService.save(product);
        return "ProductDetails";
    }
    @RequestMapping(value= {"/listproducts", "productlist"})
    public String listProducts(Model model) {
        model.addAttribute("products", productService.getAll());
        return "ListProducts";
    }
}
```

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

Null String URL - default

Re-use URL based on Method

The `@Controller` annotation in conjunction with the `@RequestMapping` allows for multiple controller handling methods

Multiple URLs can be assigned

See demo: Product_SpringMVC

@RequestParam

- Placed on Method argument

http://localhost:8080/webstore_SpringMVC/market/product?id=P1234

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

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

http://localhost:8080/webstore_SpringMVC/sizechoices?sizes=Small&sizes=Large&sizes=Medium

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

@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("/products/{category}")
public String getProductsByCategory(Model model, @PathVariable("category") String
   productCategory) {
   model.addAttribute("products",
      productService.getProductsByCategory(productCategory));
   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

See demo: webstore_SpringMVC

Data Binding

- Automatically maps request parameters to 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

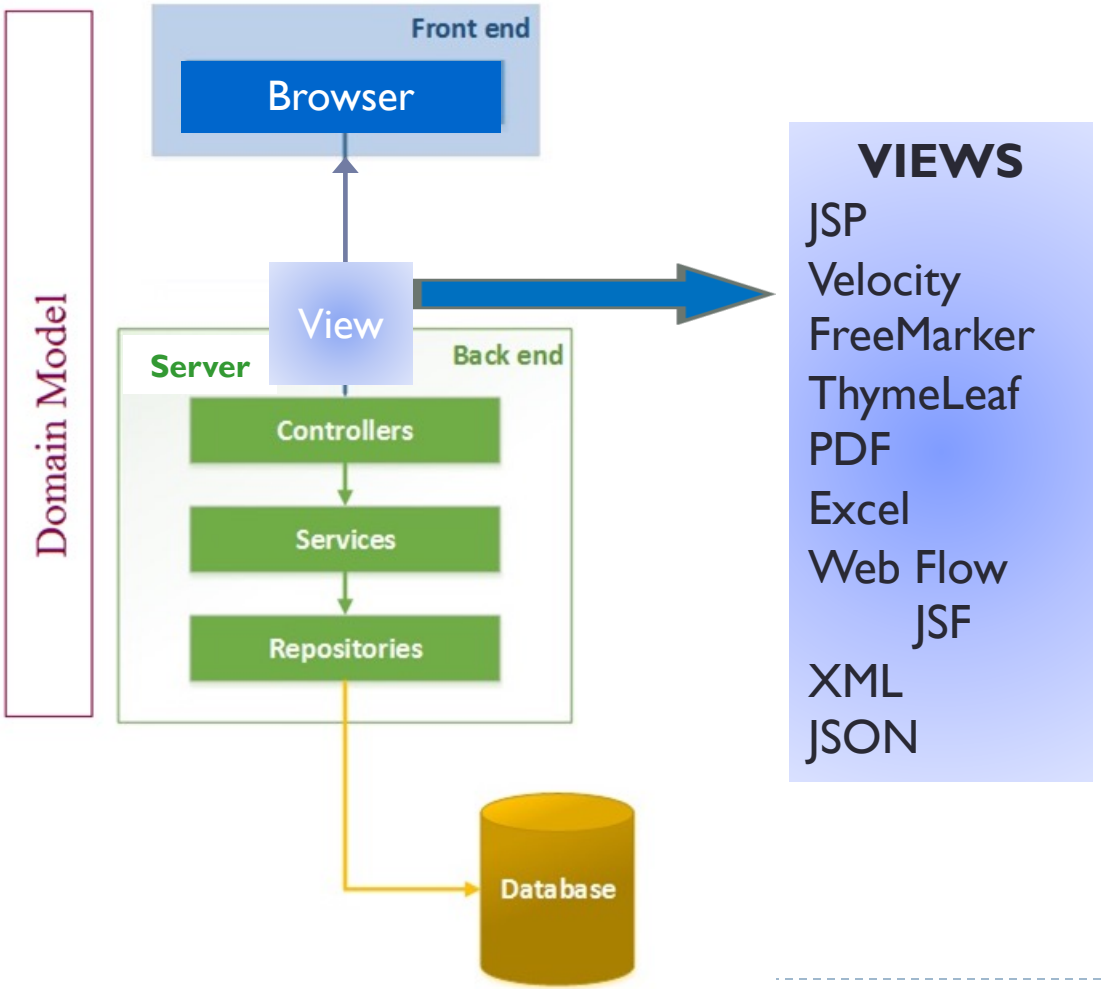
Data Binding example

```
@RequestMapping("/product")
@Controller
public class ProductController {
    @Autowired
    private CategoryService categoryService;
    @Autowired
    private ProductService productService;

    @RequestMapping
    public String getProductForm(Model model) {
        model.addAttribute("categories", categoryService.getAll());
        return "ProductForm";
    }

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

Spring Layers – With Spring MVC Layer



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.

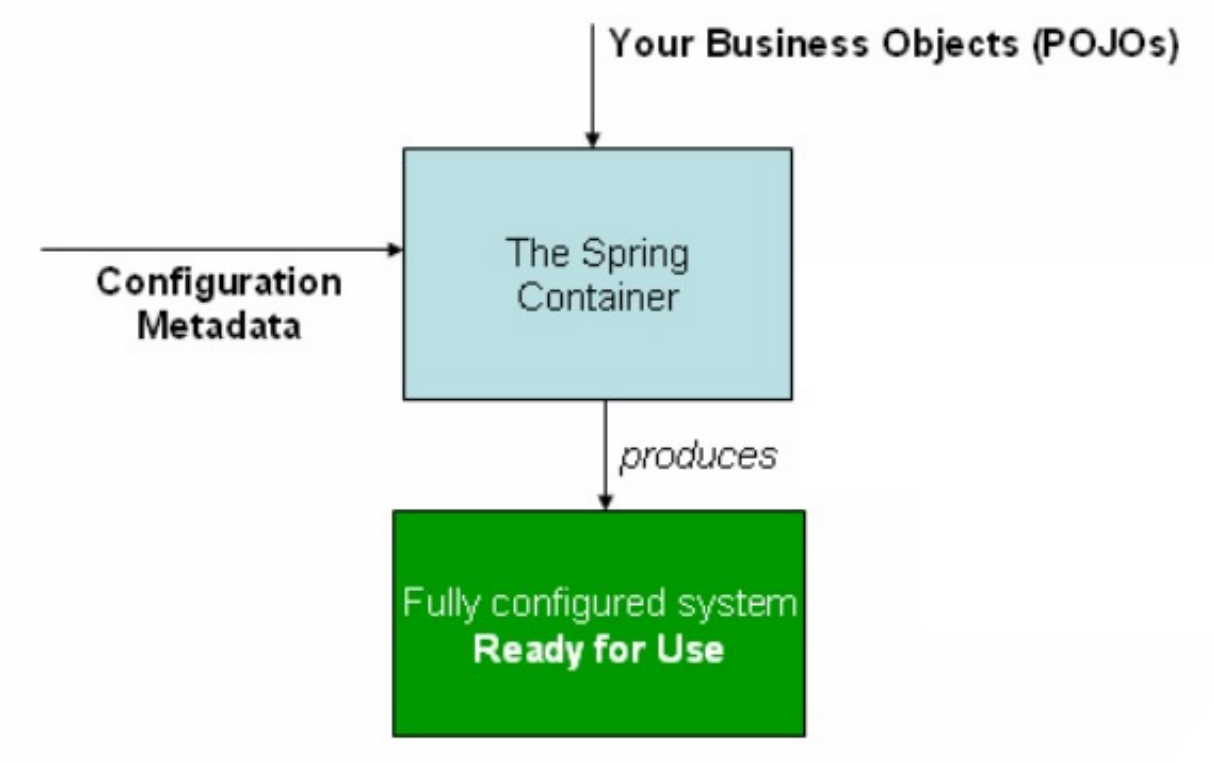
Inversion of Control [IOC]

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).

The Spring container



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

- Annotations are metadata that allow the knowledge about the creation of an object to reside within the object itself
- *The self-referral nature of Transcendental Consciousness makes all our actions clearer and more powerful*