

Web Frameworks: Spring

An Introduction

Introduction

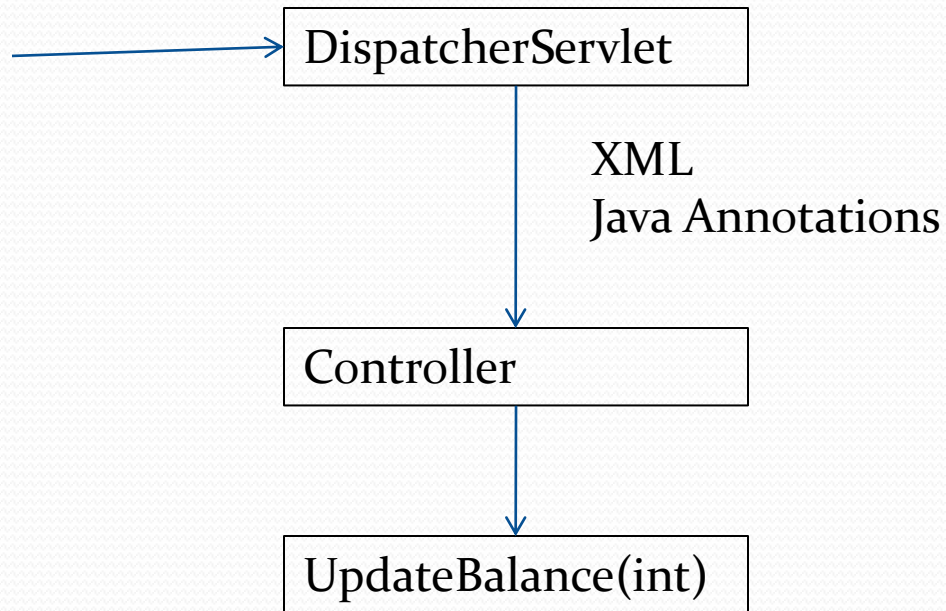
- Web.xml routes requests to the individual servlet's doGet or doPost methods
- doGet(...)
 - //extract parameters from request
 - //validation
 - //Construct objects with parameters
 - //do the processing



Spring

- In Spring
 - A specialized servlet-DispatcherServlet
 - One or more controllers having simple methods to process HTTP requests
 - The DispatcherServlet routes requests to appropriate controller-individual methods of the controllers
 - DispatcherServlet extracts request parameters, performs data validation and marshallng
 - Provides an extra layer of routing over web.xml

Spring



Routing is possible based on

1. Path like servlets
2. Request parameters using annotations
3. Data validation is taken care of

Routing through DispatcherServlet

```
public class ContactController {  
  
    public Contacts getContacts() {  
        //retrieve contacts  
        Contacts c=...  
        ...  
        return c;  
    }  
}
```

Introduction

- In EJB *public class HelloWorldBean implements SessionBean {*
- Spring avoids (as much as possible) littering your application code with its API
- Spring almost never forces you to implement a Spring-specific interface or extend a Spring-specific class
- Instead, the classes in a Spring-based application often have no indication that they're being used by Spring
- Spring has enabled the return of the plain old Java object (POJO) to enterprise development

Mapping Request parameters to method parameters

```
public class ContactController {
```

```
    public Contacts searchContacts(  
        searchstr String SearchStr) {
```

```
        //retrieve contacts
```


```
        Contacts c=...
```

```
        ...
```

```
        return c;
```

```
    }
```

Retrieves request parameters and performs basic data validation so that value of *searchstr* can be mapped to *SearchStr*

- 
- It doesn't want to be worried with
 - how that request got to the server,
 - what format it got there in,
 - how all the data got extracted from it.
 - It simplifies the methods and write cleaner, simpler methods, by using request parameters in the request mapping to extract that data and pass it into the method


```
public class ContactController {
```

```
    public Contacts searchContacts(  
        Search s) {  
        //retrieve contacts  
        Contacts c=...  
        ...  
        return c;  
    }  
}
```

Path variable provides a nicer way of parsing the request parameters rather than
?<key>=value

```
public class ContactController {
```

```
    public Contacts searchContacts(  
        Search s) {  
        //retrieve contacts  
        Contacts c=...  
        ...  
        return c;  
    }
```

```
public class Search {
```

```
    private string fname;  
    Private string lname;
```

```
    public String  
    getFname() {..  
    public setFname(String  
        name) {..  
    ...}
```

Automatic data marshalling
through HTTP message
converters

Key Features of Spring

- Inversion of Control
- Dependency Injection
- Aspect Oriented Programming

Inversion of Control

- Inversion of Control (or IoC) covers a broad range of techniques that allow an object to become a passive participant in the system

```
public class BankAccount {  
    public void transfer(BigDecimal amount, BankAccount recipient) {  
  
        recipient.deposit(this.withdraw(amount));  
    }  
  
    public void closeOut() {  
  
        this.open = false;  
    }  
  
    public void changeRates(BigDecimal newRate) {  
  
        this.rate = newRate;  
    }  
}
```

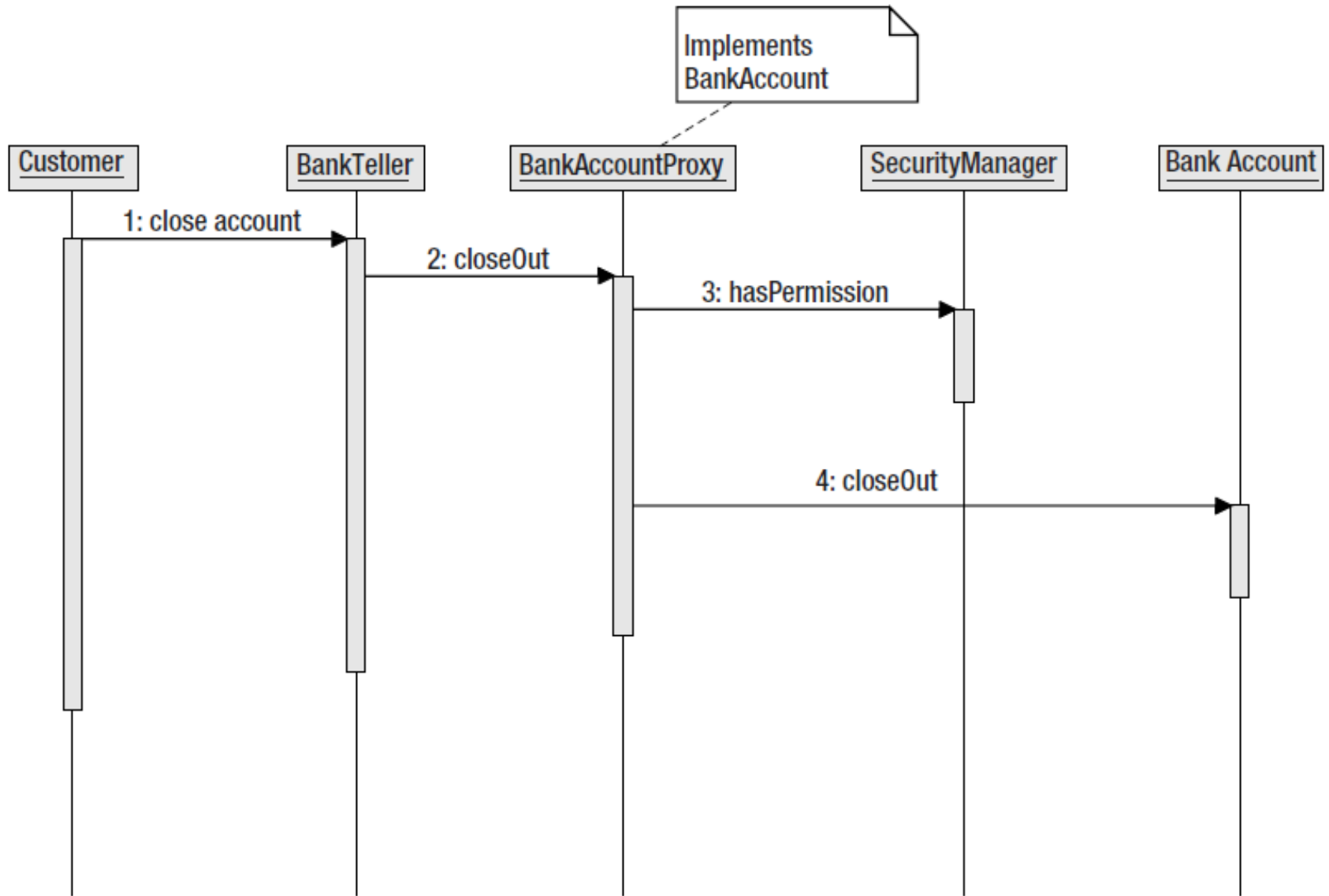
IoC Enabled code

```
public class BankAccount {  
    public void transfer(BigDecimal amount, BankAccount recipient) {  
        recipient.deposit(this.withdraw(amount));  
    }  
  
    public void closeOut() {  
        this.open = false;  
    }  
  
    public void changeRates(BigDecimal newRate) {  
        this.rate = newRate;  
    }  
}
```

- You can add the authorization mechanism into the execution path with a type of IoC implementation called *aspect-oriented programming (AOP)*

AOP

- Aspects are concerns of the application that apply themselves across the entire system
- The SecurityManager is one example of a system-wide aspect, as its hasPermission methods are used by many methods
- Other typical aspects include logging, and auditing of events
- While we may have removed calls to the SecurityManager from the BankAccount, the deleted code will still be executed in the AOP framework either at compile time or at runtime
- Runtime-proxy based AOP (simple)
- Compile time- weaving (stronger than proxies)



Inversion of Control is the broad concept of giving control back to the framework. This control can be control over creating new objects, control over transactions, or control over the security implementation.

```
public interface PriceMatrix {  
    public BigDecimal lookupPrice(Item item);  
}
```

```
public class CashRegisterImpl implements CashRegister {  
    private PriceMatrix priceMatrix = new PriceMatrixImpl();  
  
    public BigDecimal calculateTotalPrice(ShoppingCart cart) {  
        BigDecimal total = new BigDecimal("0.0");  
        for (Item item : cart.getItems()) {  
            total.add(priceMatrix.lookupPrice(item));  
        }  
        return total;  
    }  
}
```


Problems

```
public class CashRegisterImpl implements CashRegister {  
    private PriceMatrix priceMatrix = new PriceMatrixImpl();  
  
    public BigDecimal calculateTotalPrice(ShoppingCart cart) {  
        BigDecimal total = new BigDecimal("0.0");  
        for (Item item : cart.getItems()) {  
            total.add(priceMatrix.lookupPrice(item));  
        }  
        return total;  
    }  
}
```

- Every instance of CashRegisterImpl has a separate instance of PriceMatrixImpl
 - With heavy services (those that are remote or those that require connections to external resources such as databases) it is preferable to share a single instance across multiple clients
- The CashRegisterImpl now has concrete knowledge of the implementation of PriceMatrix
 - CashRegisterImpl has tightly coupled itself to the concrete implementation class
- One of the most important tenets of writing unit tests is to divorce them from any environment requirements
- The unit test itself should run without connecting to outside resources

Don't ask for
the resource;
I'll give it to
you

```
public class CashRegisterImpl implements CashRegister {
```

```
    public BigDecimal calculateTotalPrice(ShoppingCart cart) {  
        BigDecimal total = new BigDecimal("0.0");  
        for (Item item : cart.getItems()) {  
            total.add(priceMatrix.lookupPrice(item));  
        }  
        return total;  
    }  
}
```

```
public class CashRegisterImpl implements CashRegister {  
    private PriceMatrix priceMatrix;
```

```
    public BigDecimal calculateTotalPrice(ShoppingCart cart) {  
        BigDecimal total = new BigDecimal("0.0");  
        for (Item item : cart.getItems()) {  
            total.add(priceMatrix.lookupPrice(item));  
        }  
        return total;  
    }  
}
```

Dependency Injection

- Dependency Injection is a technique to wire an application together without any participation by the code that requires the dependency.
- The client usually exposes setter methods so that the framework may inject any needed dependencies.
- By moving the dependency out of the client object, it is no longer solely owned by CashRegisterImpl, and can now easily be shared among all classes.
- The client also becomes much more testable.
- The client has no environment-specific code to tie it to a particular framework.
- DispatcherServlets route to Controllers and controllers depend on certain objects

DI Example

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.ApplicationContext;
```

```
@SpringBootApplication
```

```
public class SpringDIApplication {
```

```
    public static void main(String[] args) {
```

```
        ApplicationContext applicationContext =
            SpringApplication.run(SpringDIApplication.class,
                args);
```

```
        BinarySearchImpl binarySearch =
            applicationContext.getBean(BinarySearchImpl.class);
```

```
        int result = binarySearch.binarySearch(new int[] {
            12, 4, 6 }, 3);
```

```
        System.out.println(result);
```

```
    }
```

DI Example

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;
@Component
public class BinarySearchImpl {

    private SortAlgorithm sortAlgorithm;

    public int binarySearch(int[] numbers, int
        numberToSearchFor) {
        //sortAlgorithm =new SortAlgorithmImpl();
        int[] sortedNumbers = sortAlgorithm.sort(numbers);
        System.out.println(sortAlgorithm);
        // Search the array
        return 3;
    }
}
```

DI Example

SortAlgorithm interface:

```
public interface SortAlgorithm {  
    public int[] sort(int[] numbers);  
}
```

BubbleSortAlgorithm Class:

```
import org.springframework.context.annotation.Primary;  
import org.springframework.stereotype.Component;
```

@Component

```
public class BubbleSortAlgorithm implements  
    SortAlgorithm {  
    public int[] sort(int[] numbers) {  
        // Logic for Bubble Sort  
        return numbers;  
    }  
}
```

What is Dependency Injection?

- The ability to supply (inject) an external dependency into a software component.
- Types of Dependency Injection:
 - Constructor (Most popular)
 - Setter
 - Method
 - `setSortAlgorithm(SortAlgorithm sa) {`
 - `this.sa=sa;`
 - `}`

What is a “Dependency”?

- Some common dependencies include:
 - Application Layers
 - Data Access Layer & Databases
 - Business Layer
 - External services & Components
 - Web Services
 - Third Party Components

Dependencies at a Very High Level

User Interface

Depends on



Business Logic Layer

Which Depends On

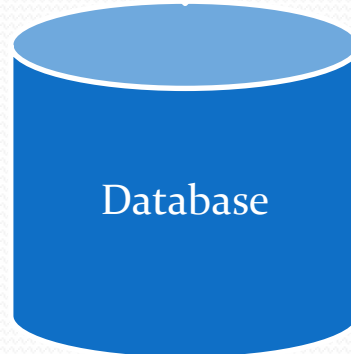


Data Access Layer

Which Depends On



Database



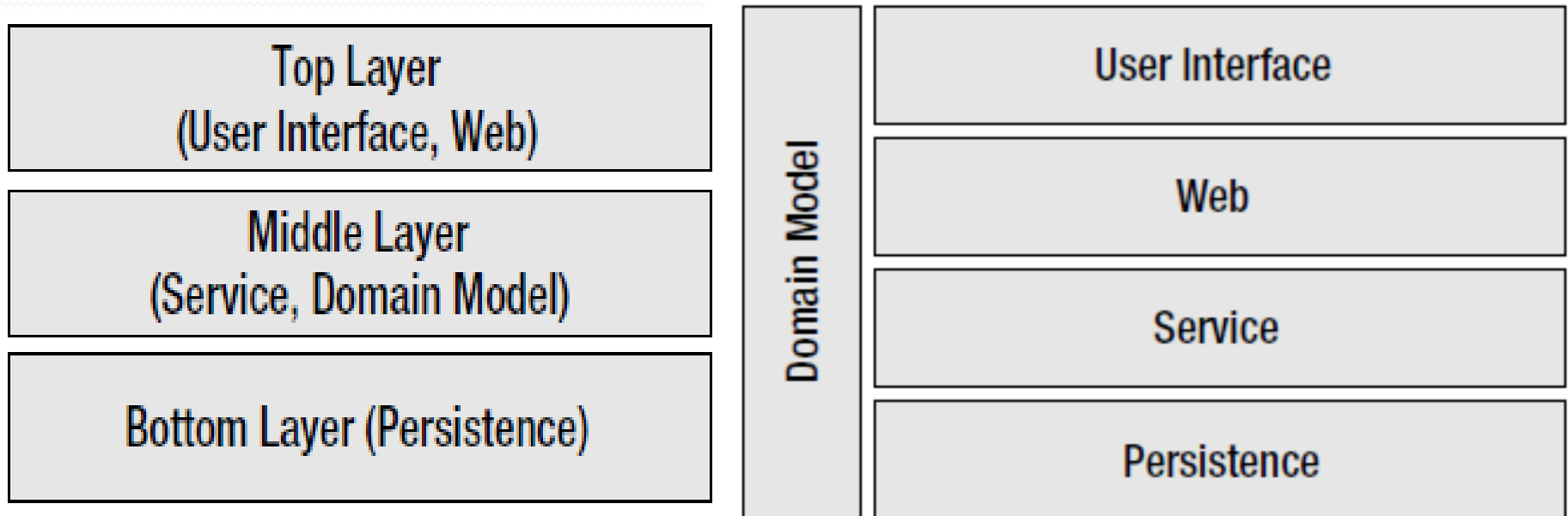
Dependency Injection Pros & Cons

- Pros
 - Loosely Coupled
 - Increases Testability
 - Separates components cleanly
 - Allows for use of Inversion of Control Container
- Cons
 - Increases code complexity
 - Developers learning time increases
 - Can Complicate Debugging at First
 - Complicates following Code Flow

ApplicationContext

- If Dependency Injection is the core concept of Spring, then the ApplicationContext is its core object.
- The ApplicationContext is a specialization of a BeanFactory, which is the registry of all the objects managed by Spring.
- Under normal circumstances, the BeanFactory is responsible for
 - creating the beans,
 - wiring them with any dependencies, and
 - providing a convenient lookup facility for the beans.
- Other interfaces are there to help to define the life cycle of beans managed by the BeanFactory
- Applications typically interact with an ApplicationContext instead of a BeanFactory
- Additionally ApplicationContext provides internationalization (i18n) facilities for resolving messages

Architecture of Spring MVC



Isolating problem domains, such as persistence, web navigation, and user interface, into separate layers creates a flexible and testable application

If we find that a layer has permeated throughout many layers, consider if that layer is itself an aspect of the system.

The interface is a contract for a layer, making it easy to keep implementations and their details hidden while enforcing correct layer usage.

Layers in Spring MVC

- User Interface Layer
 - This layer renders the response generated by the web layer into the form requested by the client
 - The act of rendering a response for a client is separate from the act of gathering the response
 - Other requests can be processed in the underlying layers while packets are sent and gathered to render the user interface
 - There are many toolkits for rendering the user interface
 - Some examples include JSP, Velocity, FreeMarker, and XSLT (all of which are well supported by Spring)
- UI designers typically work with a different toolset and are focused on a different set of concerns than the developers
 - Providing them with a layer dedicated to their needs shields them from the internal details of much of the system

User Interface Layer

- The `org.springframework.web.servlet.View` interface represents a view, or page, of the web application
- Each view technology will provide an implementation of this interface
 - Spring MVC natively supports JSP, FreeMarker, Velocity, XSLT, JasperReports, Excel, and PDF
- The `org.springframework.web.servlet.ViewResolver` provides a helpful layer of indirection.
- The `ViewResolver` provides a map between view instances and their logical names
- The view layer typically has a dependency on the domain model
 - Much of the convenience of using Spring MVC for form processing comes from the fact that the view is working directly with a domain object
 - Often the system isn't that decoupled because the view-specific classes are nearly one-to-one reflections on the domain classes

Web Layer

- Navigation logic is one of two important functions handled by the web layer
 - Managing the user experience and travels through the site is a unique responsibility of the web layer
 - Many of the layers assume much more of a stateless role.
 - The web layer, however, typically does contain some state to help guide the user through the correct path.
 - There typically isn't any navigation logic in the domain model or service layer; it is the sole domain of the web layer---flexible design customized for users
- The web layer's second main function is to provide the glue between the service layer and the world of HTTP.
- It becomes a thin layer, delegating to the service layer for all coordination of the business logic.
- The web layer is concerned with request parameters, HTTP session handling, HTTP response codes

Web Layer

- The web layer is dependent on the service layer and the domain model.
- The web layer will delegate its processing to the service layer, and it is responsible for converting information sent in from the web to domain objects sufficient for calls into the service layer
- Spring MVC provides an `org.springframework.web.servlet.mvc.Controller` interface for implementing web layer
- The Controller is responsible for accepting the `HttpServletRequest` and the `HttpServletResponse`, performing some unit of work, and passing off control to a View.

Service Layer

- This layer exposes and encapsulates coarse-grained system functionality (use cases) for easy client usage
- Client is shielded from all the POJO interactions that implement the use case by using the service layer
- No single method call on a service object should assume any previous method calls to itself.
- Any state across method calls is kept in the domain model
- In a typical Spring MVC application, a single service layer object will handle many concurrent threads of execution
- This layer attempts to provide encapsulations of all the use cases of the system.
- A single use case is often one transactional unit of work
- It also makes it easy to refer to one layer for all the high-level system functionality

Service layer

- The service layer is dependent upon the domain model and the persistence layer
- It combines and coordinates calls to both the data access objects and the domain model objects.
- The service layer should never have a dependency on the view or web layers.
- Instead of defining your business interfaces, Spring helps with the programming model.
 - Typically, the Spring Framework's `ApplicationContext` will inject instances of the service into the web Controllers.
 - Spring will also enhance our service layer with services such as transaction management and performance monitoring

Domain Model Layer

- This layer contains the business logic of the system, and thus, the true implementation of the use cases.
- The domain model is the collection of nouns in the system, implemented as POJOs.
- These nouns, such as User, Address, and ShoppingCart, contain both state (user's first name, user's last name) and behavior (shoppingCart.purchase())
- Centralizing the business logic inside POJOs makes it possible to take advantage of core object-oriented principles and practices, such as polymorphism and inheritance
- Any logic the system performs to satisfy some rule or constraint dictated by the customer is considered business logic.
- This can include anything from complex state verification to simple validation rules
 - there are some business rules that live in the database, in the form of constraints such as UNIQUE or NOT NULL
- The domain model should contain most of the business logic, but place the logic outside the model when there is a good reason

Domain model layer

- Many other layers have dependencies on the domain model.
- It is important to note, however, that **the object has no dependencies on any other layer**
- The business logic can be tested outside of the container and independently of the framework
- This speeds up development tremendously, as no deployments are required for testing
- Each layer is responsible for their problem domains, but they all live to service the domain model
 - The service layer typically combines multiple methods from the domain model together to run under one transaction.
 - The user interface layer might serialize the domain model for a client into XML or XHTML.
 - The data access layer is responsible for persisting and retrieving instances of the objects from the model.
- Spring can provide services like dependency injection in this layer

Domain model layer

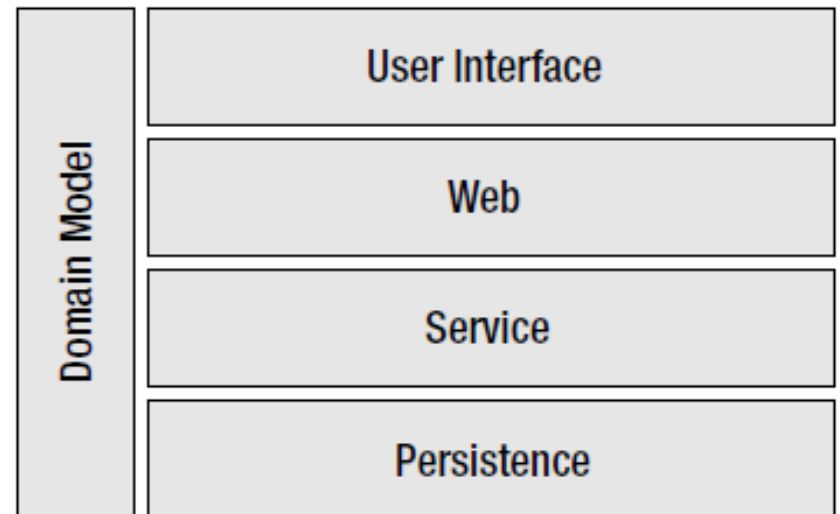
- Spring does an excellent job of creating POJOs and wiring them together.
- This works well when the object is actually created and initialized by Spring, but this isn't always the case with objects in the domain model.
- These instances can come from outside the `ApplicationContext`, for instance loaded directly by the database.

Data Access Layer

- The data access layer is responsible for interfacing with the persistence mechanism to store and retrieve instances of the object model
- Typically, only the service layer has a dependency on the data access layer
- Spring does provide common patterns for interacting with the data access layer
- For example, the template pattern is often used by data access operations to shield the implementer from common initialization and cleanup code

Use Case

- A list of current special deals must appear on the home page.
- Each special deal must display the departure city, the arrival city, and the cost.
- These special deals are set up by the marketing department and change during the day, so it can't be static.
- Special deals are only good for a limited amount of time.



Service Interface

- Creating the interface first also allows development work to begin on the web layer, before the actual interface implementation is complete
- The use case doesn't specify any type of uniqueness to the special deals.
 - Every user will see the same special deals when they view the home page
- Airports will be instances of the Airport class so that we can encapsulate both the name and airport code
- This interface provides easy access to the use cases through the façade pattern
 - These methods are coarse grained and stateless
 - Multiple calls into the methods may happen concurrently without side effects
- coarse grained to indicate that a single method call will accomplish the use case, instead of many small calls.

FlightService interface

```
public interface FlightService {  
  
    List<SpecialDeal> getSpecialDeals();  
  
    List<Flight> findFlights(SearchFlights search);  
  
}
```

```
public class SpecialDeal {
```

```
    private Airport departFrom;  
    private Airport arriveAt;  
    private BigDecimal cost;  
    private Date beginOn;  
    private Date endOn;
```

```
    public SpecialDeal(Airport arriveAt, Airport departFrom, BigDecimal co  
        Date beginOn, Date endOn) {  
        this.arriveAt = arriveAt;  
        this.departFrom = departFrom;  
        this.cost = cost;  
        this.beginOn = new Date(beginOn.getTime());  
        this.endOn = new Date(endOn.getTime());  
    }
```

```
    public BigDecimal getCost() {  
        return cost;  
    }
```

```
    public Airport getDepartFrom() {  
        return departFrom;  
    }
```

```
    public Airport getArriveAt() {  
        return arriveAt;  
    }
```

```
    public boolean isValidNow() {  
        return isValidOn(new Date());  
    }
```

```
    public boolean isValidOn(Date date) {  
        Assert.notNull(date, "Date must not be null");  
        Date dateCopy = new Date(date.getTime());  
        return ((dateCopy.equals(beginOn) || dateCopy.after(beginOn)) &&  
            (dateCopy.equals(endOn) || dateCopy.before(endOn)));  
    }
```

```
}
```

Service Interface

- For applications where rounding imprecisely can cause problems, such as interest calculations, use `BigDecimal`
- For objects that are not immutable, like `Date`, it's a best practice to make your own copies of the arguments before storing in the class or using with some business logic
 - Defensive copies
- Limited time special deals
- To encapsulate this logic, `beginOn` and `endOn` properties are added along with `isValidOn()` and `isValidNow()` methods
- The class is immutable because it models more correctly an immutable concept (a special deal can't change, but it can be deleted and a new one can take its place)

Design decisions

- First we developed both a domain object model and a service layer
- FlightService implementation is defined as a Spring bean inside the applicationContext.xml so that it may be easily injected into our web components

```
<?xml version="1.0"?>
<!DOCTYPE beans PUBLIC
    "-//SPRING//DTD BEAN//EN"
    "http://www.springframework.org/dtd/spring-beans.dtd">
<beans>

    <bean id="flightService"
        class="com.apress.expertspringmvc.flight.service.DummyFlightService" />

</beans>
```

- 
- Router → servlet
 - Router → method of a Controller

Web layer

- Install the jars
- In pom.xml
 - `<dependency>`
 - `<groupId>org.springframework</groupId>`
 - `<artifactId>spring-webmvc</artifactId>`
 - `<version>4.2.2.RELEASE</version>`
 - `</dependency>`
- Spring MVC delegates the responsibility for handling HTTP requests to Controllers
- Controllers are responsible for processing HTTP requests, performing whatever work necessary, composing the response objects, and passing control back to the main request handling work flow
- The Controller does not handle view rendering, focusing instead on handling the request and response objects and delegating to the service layer
- Views can render more than just text output. Other bundled view rendering toolkits include
 - PDF
 - Excel
 - JasperReports (an open-source reporting tool)

Controller

- When processing is complete, the Controller is responsible for building up the collection of objects that make up the response (the Model) as well as choosing what page (or View) the user sees next
- This combination of Model and View is encapsulated in a class named ModelAndView
- the Model is a Map of arbitrary objects, and the View is typically specified with a logical name.
- The view's name is later resolved to an actual View instance, later in the processing pipeline.
- It is the View's responsibility to intelligently display and render the objects in the Model.

Simple Controller

- `@Controller`
- `public class LoginController {`
 `@RequestMapping(value = "/login")`
- `@ResponseBody`
- `public String sayHello() {`
- `return "Hello World dummy";`
- `}}`

Controller for the Use case

- The most basic Controller implementation is an `org.springframework.web.servlet.mvc`.
- `AbstractController`, perfect for read-only pages and simple requests.
- Well suited for web resources that return dynamic information but don't respond to input
- It provides facilities such as controlling caching via HTTP headers and enforcing certain HTTP methods

```
public class HomeController extends AbstractController {

    private static final int FIVE_MINUTES = 5*60;
    private FlightService flights;

    public HomeController() {
        setSupportedMethods(new String[]{METHOD_GET});
        setCacheSeconds(FIVE_MINUTES);
    }

    public void setFlightService(FlightService flightService) {
        this.flights = flightService;
    }

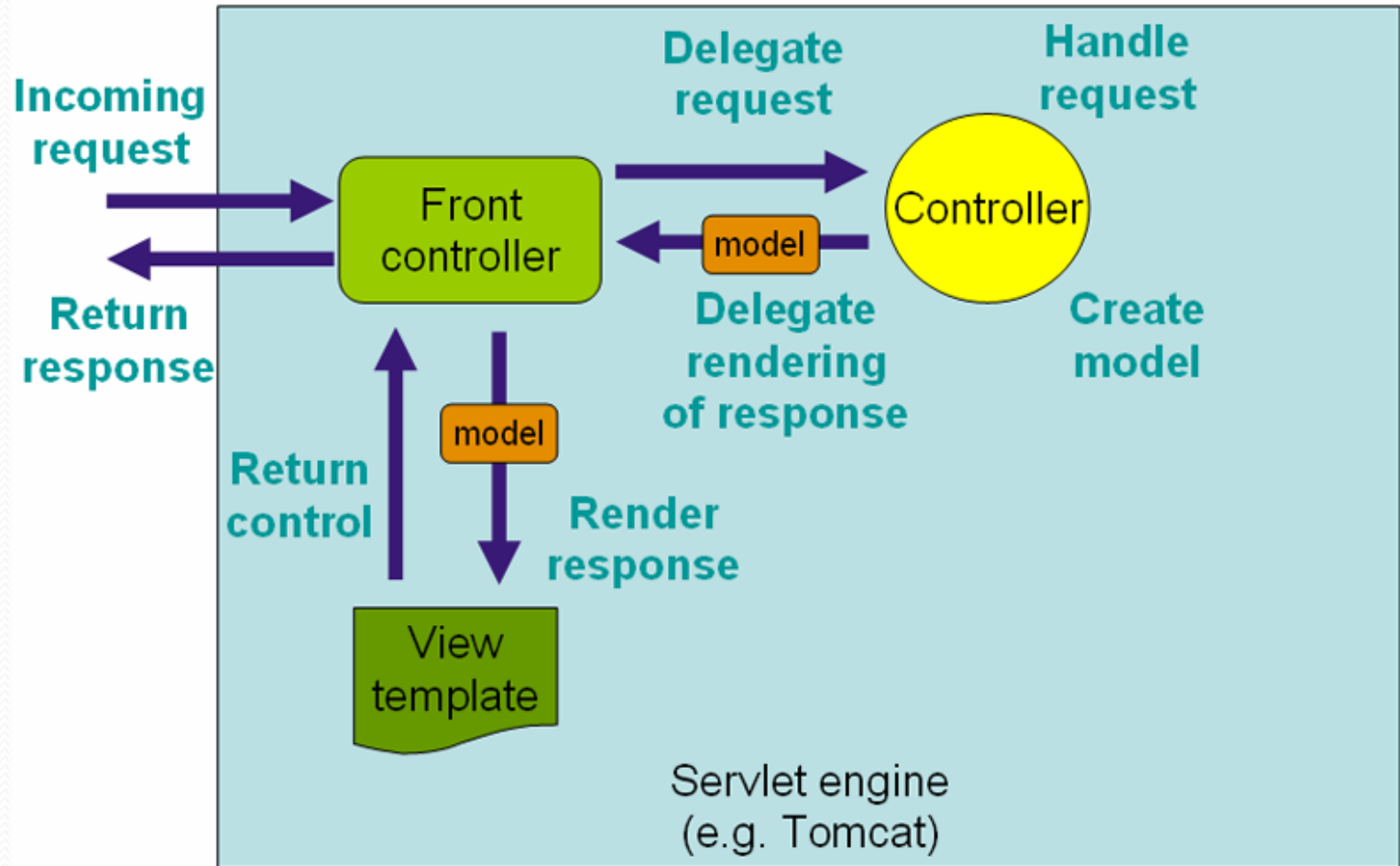
    @Override
    protected ModelAndView handleRequestInternal(HttpServletRequest req,
        HttpServletResponse res) throws Exception {
        ModelAndView mav = new ModelAndView("home");
        mav.addObject("specials", flights.getSpecialDeals());
        return mav;
    }

}
```

Controller

- The constructor configures this Controller to respond only to HTTP GET requests, because GET has semantics that most closely match that of “read.”
- The constructor instructs the superclass to send the appropriate HTTP headers to enable caching
- By default, AbstractController will send the appropriate headers to disable caching
- setCacheSeconds() method replaced by a `<property name="cacheSeconds" value="300" />` XML snippet
- XML for external issues such as wiring the classes together, or for configuration elements that have a reasonable chance of changing
- The setFlightService() method is a clear sign that the HomeController will require Dependency Injection to obtain an instance of FlightService
- The special deals are pulled from the FlightService and placed directly into the model with the name specials
- The view layer will use this name to locate the list of special deals.
- The ModelAndView instance is constructed with the view name home, used to identify which view to render for this Controller.
- These logical view names are used to keep the Controller decoupled from the view technology.

MVC Workflow



View layer

- For our JSP files, the simple `org.springframework.web.servlet.view.InternalResourceViewResolver` will provide the perfect view resolution

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

View Layer for the Use case

```
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1" />
<title>Flight Booking Service</title>
</head>
<body>
<h1>Welcome to the Flight Booking Service</h1>
<p>We have the following specials now:</p>

<ul>
  <c:forEach items="${specials}" var="special">
    <li>${special.departFrom.name} - ${special.arriveAt.name} from
    ${special.cost}</li>
  </c:forEach>
</ul>

<p><a href="search">Search for a flight.</a></p>

</body>
</html>
```

Summary of Web Layer Implementation

- The HomeController delegates to the FlightService to find any special deals and then places those deals inside a ModelAndView.
- This ModelAndView object is returned from the Controller, bringing along with it the name of the view to render.
- The view name is resolved to a JSP file by an InternalResourceViewResolver, which creates a full resource path by combining a prefix, the view name, and a suffix.
- The InternalResourceViewResolver and HomeController are defined and configured in the spring-servlet.xml file, which defines all the beans that make up the web components application context

Spring-servlet.xml

```
<?xml version="1.0"?>
<!DOCTYPE beans PUBLIC
    "-//SPRING//DTD BEAN//EN"
    "http://www.springframework.org/dtd/spring-beans.dtd">
<beans>

    <bean name="/home"
        class="com.apress.expertspringmvc.flight.web.HomeController">
        <property name="flightService" ref="flightService" />
    </bean>

</beans>
```

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

Application Context configuration

- Spring MVC applications usually have two `ApplicationContexts` configured in a parent-child relationship
- The parent context, or root `ApplicationContext`, contains all of the non-web specific beans such as the services, DAOs, and supporting POJOs.
- Root `ApplicationContext` contains the `FlightService` implementation, and is named `applicationContext.xml`.
 - This `ApplicationContext` must be initialized before the web-specific resources are started
- The `ContextLoaderListener` is used to create root `ApplicationContext`, responding to the `contextInitialized()` callback of a `ServletContextListener`
- Once the `ApplicationContext` is built it will be placed into the application scope of the web application so that the entire application may access it

Application-context.xml

```
<?xml version="1.0"?>
<!DOCTYPE beans PUBLIC
    "-//SPRING//DTD BEAN//EN"
    "http://www.springframework.org/dtd/spring-beans.dtd">
<beans>

    <bean id="flightService"
        class="com.apress.expertspringmvc.flight.service.DummyFlightService" />

</beans>
```

Root application context

- applicationContext.xml is the root context configuration for every web application.
- Spring loads applicationContext.xml file and creates the ApplicationContext for the whole application.
- If you are not explicitly declaring the context configuration file name in web.xml using the contextConfigLocation param, Spring will search for the applicationContext.xml under WEB-INF folder and throw FileNotFoundException if it could not find this file.

Web application context

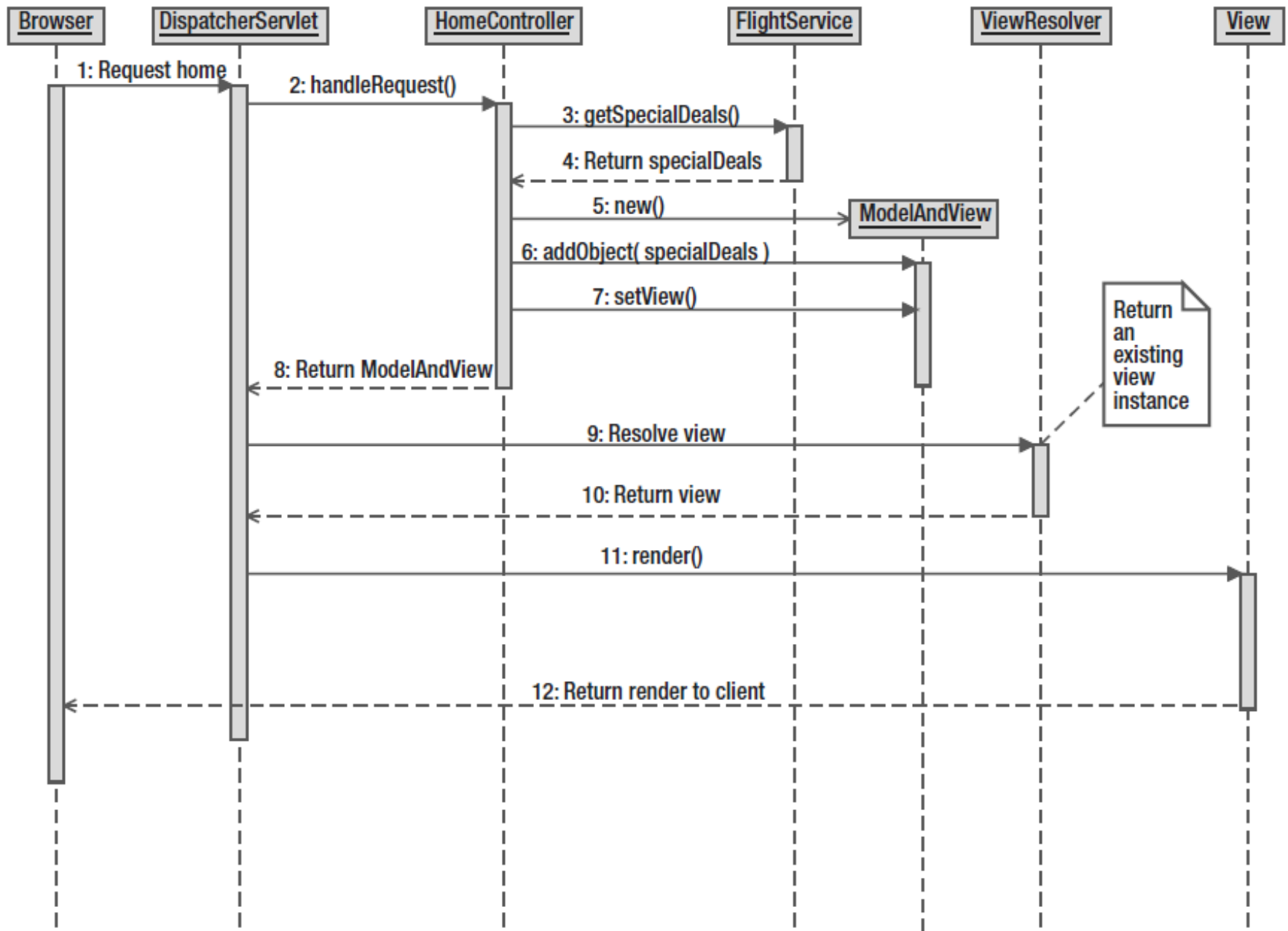
- There can be multiple **WebApplicationContext** in a single web application.
- Each **DispatcherServlet** associated with single **WebApplicationContext**.
- **xxx-servlet.xml** file is specific to the **DispatcherServlet** and a web application can have more than one **DispatcherServlet** configured to handle the requests
 - In such scenarios, each **DispatcherServlet** would have a separate **xxx-servlet.xml** configured.
 - But, **applicationContext.xml** will be common for all the servlet configuration files
- Spring will by default load file named “**xxx-servlet.xml**” from your webapps **WEB-INF** folder where **xxx** is the servlet name in **web.xml**
- If you want to change the name of that file name or change the location, add **init-param** with **contextConfigLocation** as param name.

DispatcherServlet

- The Spring (MVC) framework is designed around a DispatcherServlet that dispatches requests to handlers, with configurable handler mappings, view resolution
- A Front Controller as “a controller that handles all requests for a Web site.”
- DispatcherServlet serves the role of a Front Controller
- The response from a Controller is rendered for output by an instance of the View class

DispatcherServlet

- When the DispatcherServlet initializes, it will search the WebApplicationContext for one or more instances of the elements that make up the processing pipeline (such as ViewResolvers)
- WebApplicationContext is a special ApplicationContext implementation of the servlet environment and the ServletConfig object
- To locate the XML for the WebApplicationContext, the DispatcherServlet will by default take the name of its servlet definition from web.xml, append -servlet.xml, and look for that file in /WEB-INF
 - if the servlet is named spring, it will look for a file named /WEB-INF/spring-servlet.xml



DispatcherServlet

- It gets its name from the fact that it dispatches the request to many different components, each an abstraction of the processing pipeline
 1. Discover the request's Locale; expose for later usage.
 2. Locate which request handler is responsible for this request (e.g., a Controller).
 3. Locate any request interceptors for this request. Interceptors are like filters, but customized for Spring MVC.
 4. Invoke the Controller.
 5. Call `postHandle()` methods on any interceptors.
 6. If there is any exception, handle it with a `HandlerExceptionResolver`.
 7. If no exceptions were thrown, and the Controller returned a `ModelAndView`, then render the view. When rendering the view, first resolve the view name to a View instance.

DispatcherServlet

- The `WebApplicationContext` is a special `ApplicationContext` implementation that is aware of the servlet environment and the `ServletConfig` object
- For interfaces like `ViewResolvers`, the `DispatcherServlet` can be configured to locate all instances of the same type
- The `DispatcherServlet` uses the `Ordered` interface to sort many of its collections of delegates.
 - This is done through a property named *order*
- Usually, the first element to respond with a non-null value wins
- During initialization, the `DispatcherServlet` will look for all implementations by type of `HandlerAdapters`, `HandlerMappings`, `HandlerExceptionResolvers`, and `ViewResolvers`
- The `DispatcherServlet` is configured with default implementations for most of these interfaces.
 - This means that if no implementations are found in the `ApplicationContext` (either by name or by type), the `DispatcherServlet` will create and use them