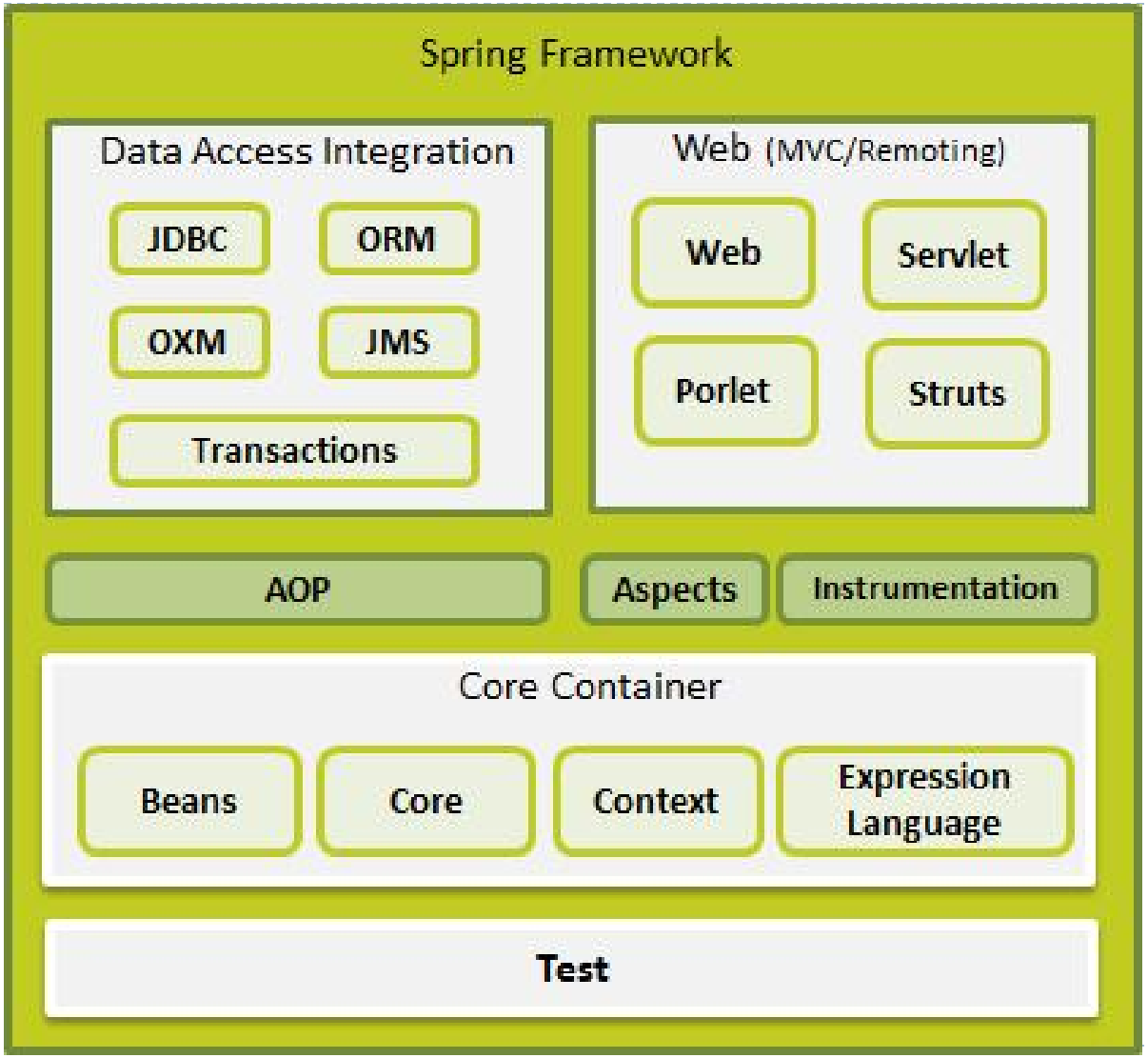
Spring Framework

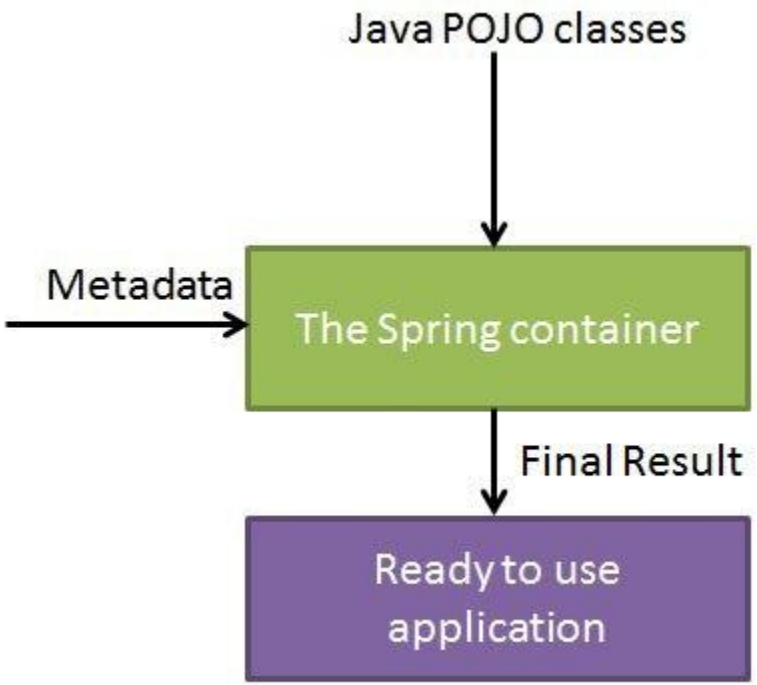
- **Benefits…**
  - Spring enables developers to develop enterprise-class applications using POJOs.
    - You do not need an EJB container such as an application server
    - Use only robust servlet container such as Tomcat
  - Spring is organized in modular fashion
    - Worry only about the ones your need and ignore the rest
  - Spring makes use of the existing technologies easer
    - ORM frameworks (Hibernate/JPA), JEE, and other view technologies
  - Testing with spring is easier (Thanks to POJO's & Spring DI)
    - Support for various test frameworks such as Junit, TestNG
  - Spring MVC – a well-designed web MVC framework
  - Spring provides API to translate technologic-specific exceptions into consistent, unchecked exceptions (Ex: Exceptions thrown by JDBC, Hibernate)
  - Spring IOC Container is lightweight compared to EJB containers
  - Spring has a good transaction management support (Both local [single DB] and global transactions [JTA])

Spring Framework



Spring IOC Container

- The core of the spring framework.
  - Create Objects
  - Wire them together
  - Configure them
  - Manage their complete life cycle from creation till destruction



Spring Bean Life Cycle

- Initialization callbacks

  - When a bean is instantiated (created), it may be required to perform some initialization to get it into a stable state
  - When a bean is no longer required and is removed from the container, some cleanup may be required
  - **init-method / afterPropertiesSet() / @PostConstruct**
    - Method that is called on the bean immediately upon instantiation
    - This method is called after the bean is created and its properties (if any) are set
  - **destroy-method / destroy() / @PreDestroy**
    - Method that is called just before the bean is removed from the container

Spring Bean Life Cycle

Bean Post Processors

- The BeanPostProcessors operate on bean (or object) instances which means that the spring IoC container instantiates the bean instance and then BPP interfaces do their work.

```
public class InitHelloWorld implements BeanPostProcessor {

   public Object postProcessBeforeInitialization(Object bean,
                String beanName) throws BeansException {
      System.out.println("BeforeInitialization : " + beanName);
      return bean;  // you can return any other object as well
   }

   public Object postProcessAfterInitialization(Object bean,
                String beanName) throws BeansException {
      System.out.println("AfterInitialization : " + beanName);
      return bean;  // you can return any other object as well
   }

}
```

- By default, Spring will not be aware of the @PostConstruct and @PreDestroy annotation. To enable it, you have to either register '**CommonAnnotationBeanPostProcessor**' or specify the '**<context:annotation-config />**' in bean configuration file,

Spring Bean Life Cycle

- Bean Life Cycle Sequence
  - Bean is created / instantiated
  - Bean properties are set (Setter Injection)
  - If (BPP)
    - postProcessBeforeInitialization()
  - If (init-method)
    - init-method()
  - If (BPP)
    - postProcessAfterInitialization()
  - Bean is ready
  - Bean is getting destroyed
  - If (destry-metod)
    - destroy()

Spring AOP

- Add the dependencies inside pom.xml
- Inside XML – Enabline @AspectJ Programming Module

```
<aop:aspectj-autoproxy/>
```

- Define an Aspect Java Class

```
@Aspect
public class AspectModule {
        …
}
```

- Define the Aspect as a bean in XML

```
<bean id="myAspect" class="org.xyz.AspectModule">
<!-- configure properties of aspect here as normal -->
</bean>
```

- Declare a PONITCUT in the Aspect

```
@Pointcut("execution(* com.xyz.myapp.service.*.*(..))") // expression
private void businessService() {} // signature
```

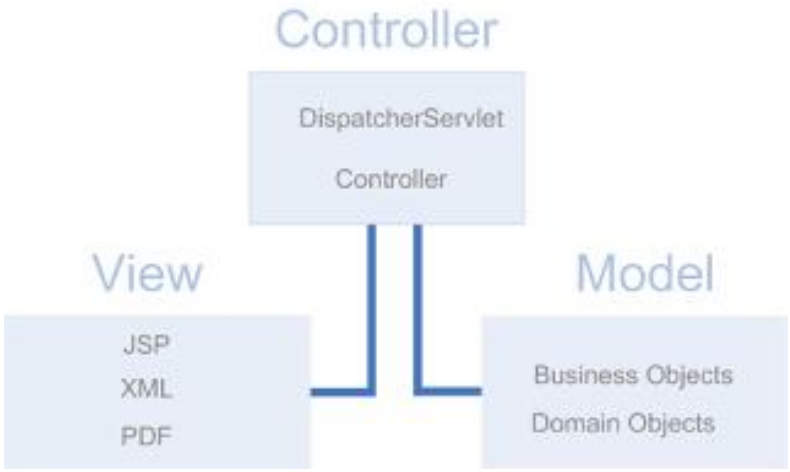
- Declare an ADVICE in the Aspect

```
@Around("businessService()")
public void doAroundTask() {
        ...
}
```

@Before
@After
@AfterReturning
@AfterThrowing
@Around

Spring MVC

- Spring MVC is part of the Spring Framework as a MVC implementation
- The Spring Web model-view-controller (MVC) framework is designed around a **DispatcherServlet** that dispatches requests to handlers

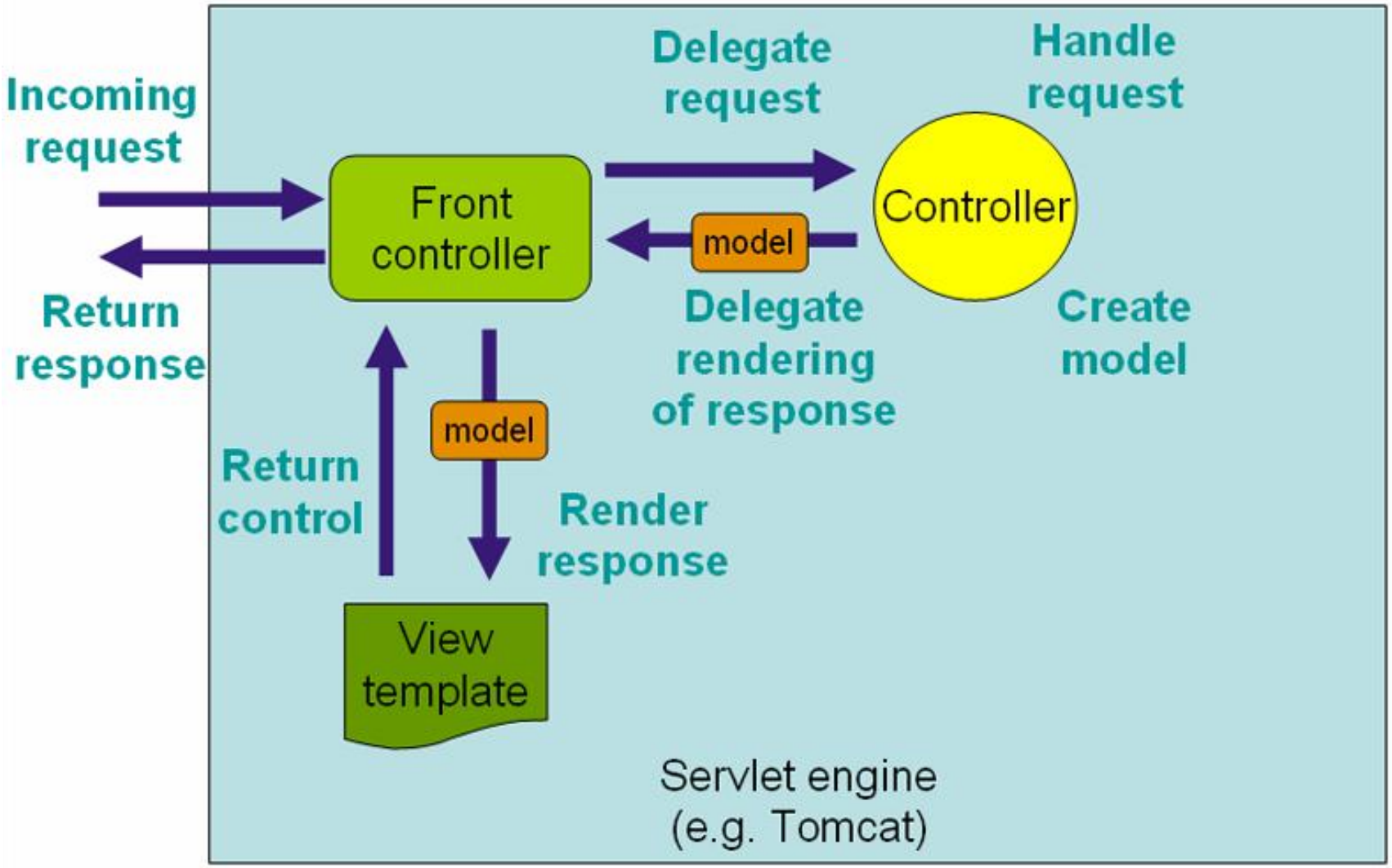


  - Open for extension…. Closed for modification

The Dispatcher Servlet

- Spring MVC framework is request-driven, designed around a central Servlet that dispatches requests to controllers and offers other functionalities that facilitates the development of web applications
- Spring's DispatcherServlet is completely integrated with the Spring IoC container and as such allows you to use every other feature that Spring has
- DispatcherServlet is an expression of the "Front Controller" design pattern

The Dispatcher Servlet



src/main/webapp/WEB-INF/web.xml

- All incoming requests flow through a **DispatcherServlet**
- **DispatcherServlet** is an actual Servlet (it inherits from HttpServlet base class) and as such is declared in the web.xml file.
- You need to map requests that you want the DispatcherServlet to handle, by using a URL mapping in the same web.xml file

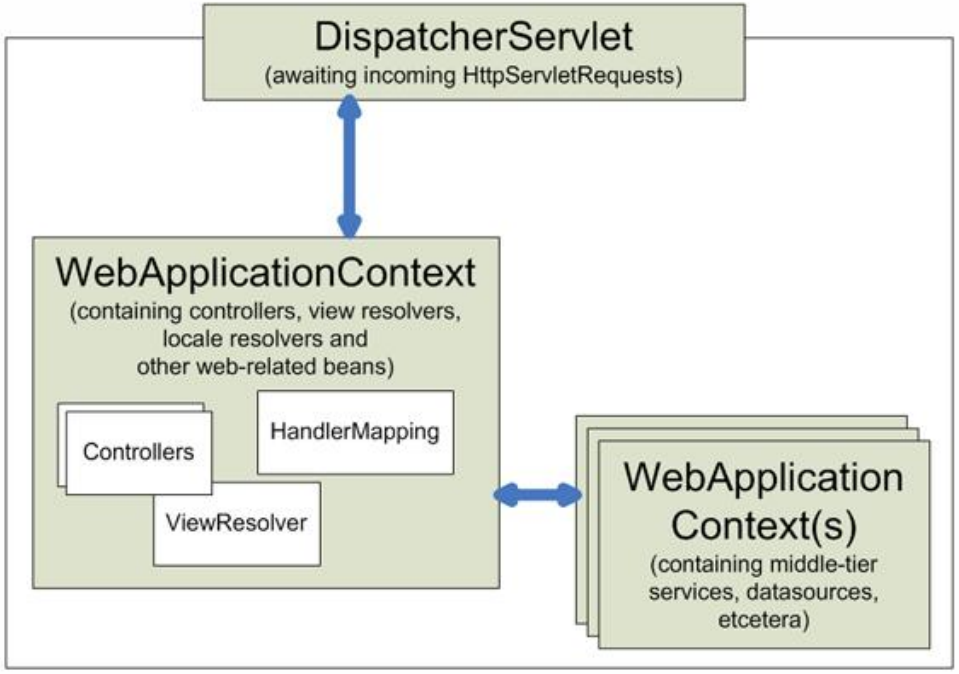
```
<servlet>
        <servlet-name>dispatcher</servlet-name>
        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
        <init-param>
                <param-name>contextConfigLocation</param-name>
                <param-value>WEB-INF/classes/dispatcher-servlet.xml</param-value>
        </init-param>
        <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
        <servlet-name>dispatcher</servlet-name>
        <url-pattern>*.do</url-pattern>
        <url-pattern>/login.do</url-pattern>
</servlet-mapping>
```

DispatcherServlet's WebApplicationContext

```
<servlet>
        <servlet-name>dispatcher</servlet-name>
        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
        <init-param>
                <param-name>contextConfigLocation</param-name>
                <param-value>WEB-INF/classes/dispatcher-servlet.xml</param-value>
        </init-param>
        <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
        <servlet-name>dispatcher</servlet-name>
        <url-pattern>*.do</url-pattern>
        <url-pattern>/login.do</url-pattern>
</servlet-mapping>
```

- In Spring MVC each DispatcherServlet has its own WebApplicationContext. By default, spring always looks for the context at <*dispatcherServletName*-***servlet.xml***>. However this can be overridden with a Servlet init-param
- The *DispatcherServlet* related *WebApplicationContext* should have MVC-specific configurations such as Controllers, HandlerMappings, ViewResolvers etc…,

ROOT WebApplicationContext



- Other non MVC-specific configuration such as the beans for service or persistence layer should be in root *WebApplicationContext*.
- In SpringMVC the root *WebApplicationContext* is bootstrapped by using *ContextLoadListener* specified as Listener in web.xml.
- So the DispatcherServlet WebApplicationContext will inherit (extend) from the ROOT WebApplicationContext

ROOT WebApplicationContext

```
<listener>
        <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
</listener>
<context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>WEB-INF/classes/app-context.xml,WEB-INF/classes/security.xml</param-value>
</context-param>
```

```
WEB-INF
   |
   |--web.xml
   |
   |--classes
          |
          |--dispatcher-servlet.xml
          |
          |--app-context.xml
          |--security.xml
```

Handler Mappings

- HandlerMapping is the class that helps DispatcherServlet to map an incoming request to a particular Controller class.
- There are many HandlerMapping implementations in Spring, however the most used one is the annotated controllers
- HandlerMapping bean has one important property – interceptors to which a user defined handler interceptor can be injected
- **RequestMappingHandlerMapping**
  - This HandlerMapping implementation automatically looks for @RequestMapping annotations on all @Controller beans
  - The RequestMappingHandlerMapping is the only place where a decision is made about which method should process the request
  - **<mvc:annotation-driven />**
    - This annotation in the DispatcherServlet WebApplicationContext (file: /WEB-INF/classes/dispatcher-servlet.xml), will automatically register the RequestMappingHandlerMapping bean

```
<!-- Configures the @Controller programming model -->
<mvc:annotation-driven />
```

WEB-INF/classes/dispatcher-servlet.xml

- Import any other context configuration files. For example to import the main ROOT application context file:

  ```
  <import resource="app-context.xml"/>
  ```

- Configure the @Controller programming model

  ```
  <mvc:annotation-driven />
  ```

- Configure all the View Resolvers
  - *TilesViewResolver (and TilesConfigurer)*
  - *JSPViewResolver*
  - *MultiPartResolver*
  - *ExceptionResolver*
  - *CookieLocaleResolver*

WEB-INF/classes/app-context.xml

- Initiate a component scan

```
<context:component-scan base-package="com.tnsi">
```

- Configure a property place holder configurer

```
<bean class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
        <property name="locations">
                <list>
                        <value>classpath:raise-app.conf</value>
                        <value>classpath:usercompany.properties</value>
                </list>
        </property>
</bean>
```

Note that at run-time, these property files will typically be placed at: /WEB-INF/classes folder so they get detected in the classpath scanning

WEB-INF/classes/app-context.xml

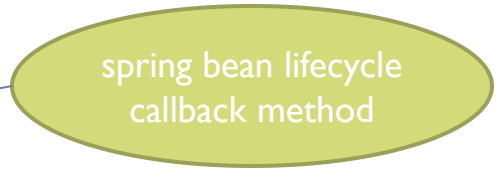
- Property place holder configurer (This is a BeanFactroy PostProcessor)
  - Sample Contents

/WEB-INF/classes/raise-app.conf

```
db.host=raisedb
db.port=1521
db.schema=raise
db.username=raise
db.password=raise
db.driver=oracle.jdbc.driver.OracleDriver
jdbc.url=jdbc:oracle:thin:@${db.host}:${db.port}:${db.schema}
```

  - How to use

```
<bean id="fallbackDataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource"
      destroy-method="close">
       <property name="driverClass" value="${db.driver}" />
       <property name="jdbcUrl" value="${jdbc.url}" />
       <property name="user" value="${db.username}" />
       <property name="password" value="${db.password}" />
       <property name="unreturnedConnectionTimeout" value="300" />
       <property name="minPoolSize" value="3" />
       <property name="maxPoolSize" value="3" />
       <property name="idleConnectionTestPeriod" value="300" />
</bean>
```



WEB-INF/classes/app-context.xml

- Configure Resource Bundle Message Resource (i18n)

```
<bean id="messageSource"
class="org.springframework.context.support.ReloadableResourceBundleMessageSource">
	<property name="basenames">
		<list>
			<value>classpath:messages</value>
			<value>classpath:messages.validation</value>
			<value>classpath:usercompany</value>
		</list>
	</property>
	<property name="defaultEncoding" value="UTF-8" />
</bean>
```

Note that at run-time, these properties files will typically be placed at: /WEB-INF/classes folder so they get detected in the classpath scanning

The basenames property of the messageSource bean will by default look for files with the ".properties" extension.

WEB-INF/classes/app-context.xml

- i18n
  - There should be multiple resource messages files one for each locale
    - messages.properties – Default and English
    - messages_zh_CN.properties – Chinese
  - Sample Contents

```
app.title=Welcome to TNS RAISE Application
app.footer.tns.long.name=Transaction Network Services

carrier.page.title=Carrier Management
carrier.list.header=Carrier Search
```

  - How to use (Say in JSP)
    - We will use the Spring TagLib

```
/WEB-INF/jsp/layout/layout.jsp

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

<title><spring:message code="app.title" /></title>
```

WEB-INF/classes/app-context.xml

- Session Factory (Depends on DataSource)
  - Spring support for hibernate LocalSessionFactory
    - dataSource
    - mappingLocations – List of *.hbm.xml files
    - hibernateProperties – Such as show_sql, dialect etc…,
- Other Service & Repository Beans (One example shown below)

CarrierController

```
@Autowired
private Validator carrierValidator;

@Autowired
private CarrierService carrierService;
```

View Resolver

- All the handler methods in the controller class must resolve to a logical view name explicitly by returning a *String*
- ViewResolver interface
  - Provides a mapping between view names and the actual views
  - There are many implementation is spring, but the prominent ones used are InternalResourceViewResolver and TilesViewResolver

InternalResourceViewResolver

What's internal resource views?

In Spring MVC or any web application, for good practice, it's always recommended to put the entire views or JSP files under "WEB-INF" folder, to protect it from direct access via manual entered URL. Those views under "WEB-INF" folder are named as internal resource views, as it's only accessible by the servlet or Spring's controllers class.

- In Spring MVC, **InternalResourceViewResolver** is used to resolve "internal resource view" (in simple, it's final output, jsp or html page) based on a predefined URL pattern. In additional, it allow you to add some predefined prefix or suffix to the view name (prefix + view name + suffix), and generate the final view page URL.

  If let's say controller returns a string "carrierDetails"; the actual (physical) view it resolves to is

  /WEB-INF/jsp/**carrierDetails**.jsp

```
<bean id="jspViewResolver" class="org.springframework.web.servlet.view.InternalResourceViewResolver">
        <property name="prefix"><value>/WEB-INF/jsp/</value></property>
        <property name="suffix"><value>.jsp</value></property>
        <property name="order"><value>2</value></property>
</bean>
```

Redirecting to views

- Redirect – redirect:
  - A Note on InternalResourceViewResolver
    - Convenient subclass of UrlBasedViewResolver that supports InternalResourceView (in effect, Servlets and JSPs) and subclasses such as JstlView and TilesView. You can specify the view class for all views generated by this resolver by using setViewClass(..).
  - If a view name is returned that has the prefix redirect:, the UrlBasedViewResolver (and all subclasses) will recognize this as a special indication that a redirect is needed. The rest of the view name will be treated as the redirect URL.

```
@RequestMapping(value="/hub/delete.do", method = {RequestMethod.POST})
public ModelAndView deleteHubAgreement(@RequestParam int delAgreementId)
{
        agreementService.deleteHubAgreement(delAgreementId);

        return new ModelAndView("redirect:/agreements/hub/list.do");
}
```

  - In this example, the controller handler method does a "delete" operation and after this it is desirable that the user is presented back with the list of objects. So here the response is delegated to another controller method – Here in this case, the list.do URL is called again by the client.

ExceptionResolver - Handling Exceptions in a generic way

- SimpleMappingExceptionResolver
  - Takes the class name of any exception (ex: java.lang.Exception) that might be thrown from a handler method and map it to a view name.
    - Map exception class names to view names
    - Specify a default (fallback) error page for any exception not handled anywhere else
    - Configuration

```
<bean id="exceptionResolver"
	class="org.springframework.web.servlet.handler.SimpleMappingExceptionResolver">
	<property name="exceptionMappings">
		<props>
			<prop key="java.lang.Exception">friendlyError</prop>
		</props>
	</property>
</bean>
```

    - annotation - @ExceptionHandler(Exception.class) – Place this on the @Controller method(s)
    - friendlyError.jsp

```
<h3>${exception.message}</h3>
```

Annotations, annotations and annotations everywhere….

Implementing Controllers

- Controllers provide access to the application behavior that you typically define through a service interface
- Controllers interpret user input and transform it into a model that is represented to the user by the view

```
@Controller
public class CarrierController {

        @RequestMapping(value = "/viewCarrier.do", method = {RequestMethod.GET})
        public ModelAndView listCarrier(
                        @RequestParam(value="carrier", required=false) String carrierCode,
                        @RequestParam(value="carrierName", required=false) String carrierName,
                        final Model model)
        {
                return new ModelAndView("viewCarrier", "carrierList", carrierService.findByCarrierCode(carrierCode,true));
        }
}
```

- In this example, the method is called for a URL – "…/app-name/viewCarrier.do"

  Call from a JSP: <spring:message code='carrier.label' />

- The methods takes Model as input parameter and return a Model and View Object – The View is a string which is mapped either to a tiles definition name or an actual jsp file

@Controller

- The @Controller annotation indicates that a particular class serves the role of a *controller*
- Is a stereotype annotation extending from @Component.
- The DispatcherServlet scans such annotated classes for mapped methods and detects @RequestMapping annotations (see next slides)

```
<context:component-scan base-package="com.tnsi" />
```

@RequestMapping

- @RequestMapping is used to map URLs such as */viewCarrier.do* onto a methods in the Controller class
- Can be used both at class level and methods levels
- Typically the class-level annotation maps a specific request path (or path pattern) onto a form controller, with additional method-level annotations narrowing the primary mapping for a specific HTTP method request method ("GET", "POST", etc.) or an HTTP request parameter condition.

- Spring 3.1+
  - The ***RequestMappingHandlerMapping*** is the only place where a decision is made about which method should process the request
  - Think of controller methods as a collection of unique service endpoints with mappings for each method derived from type (class level) and method-level @RequestMapping information

Consumable Media Types

You can narrow the primary mapping by specifying a list of consumable media types. The request will be matched only if the *Content-Type* request header matches the specified media type. For example:

```
@Controller
@RequestMapping(value = "/pets", method = RequestMethod.POST, consumes="application/json")
public void addPet(@RequestBody Pet pet, Model model) {
    // implementation omitted
}
```

Note that in the above example, the request data is sent as part of the HTTP request body which is bind to the method parameter using the @RequestBody annotation

Producible Media Types

You can narrow the primary mapping by specifying a list of producible media types. The request will be matched only if the *Accept* request header matches one of these values. Furthermore, use of the *produces* condition ensures the actual content type used to generate the response respects the media types specified in the *produces* condition. For example:

```
@Controller
@RequestMapping(value = "/pets/{petId}", method = RequestMethod.GET, produces="application/json")
@ResponseBody
public Pet getPet(@PathVariable String petId, Model model) {
    // implementation omitted
}
```

Note that in the above example, the response data is sent back to the client as HTTP response (Not to a view) using the @ResponseBody annotation.

Spring does the automatic conversion of the returned object to a HTTP Response to a format of JSON or XML – Spring uses Jackson for JSON and JAXB for XML

URI Template Patterns - @PathVariable

- URI Template
  - Is a URI-like string, containing one or more variable names

    http://www.example.com/users/{*userId*} – variable name is *userId*
  - When you substitute values for these variables, the template becomes a URI

    Assigning the value *arun* to the variable yields http://www.example.com/users/*arun*.

Use the **@PathVariable** annotation on a method argument to bind it to the value of a URI template variable

```
@RequestMapping(value="/owners/{ownerId}", method=RequestMethod.GET)
public String findOwner(@PathVariable String ownerId, Model model) {
    Owner owner = ownerService.findOwner(ownerId);
    model.addAttribute("owner", owner);
    return "displayOwner";
}
```

The URI Template " /owners/{ownerId}" specifies the variable name ownerId. When the controller handles this request, the value of ownerId is set to the value found in the appropriate part of the URI. For example, when a request comes in for /owners/arun, the value of ownerId is arun

@RequestParam

- Use the @RequestParam annotation to bind request parameters to a method parameter in your controller

```
@RequestMapping(value = "/carrier.do", method = {RequestMethod.GET, RequestMethod.HEAD})
public ModelAndView editCarrier(
            @RequestParam(value="id", required=false) Integer carrierId,
            @RequestParam(value="action", required=false) String action
)
{
      //
}
```

- The above method can be invoked something like below:

```
var actionUrl =  "carrier.do?action=" + action + "&id=" + id;
```

Note that the request params (action & id) are passed as part of the request URL

- Parameters using this annotation are required by default, but you can specify that a parameter is optional by setting @RequestParam's required attribute to false
- Type conversion is applied automatically if the target method parameter type is not String

@RequestBody

- The @RequestBody method parameter annotation indicates that a method parameter should be bound to the value of the HTTP request body. For example:

```
@RequestMapping(value = "/something", method = RequestMethod.PUT)
public void handle(@RequestBody String body, Writer writer) throws IOException {
    writer.write(body);
}
```

- HttpMessageConverter is responsible for converting from the HTTP request message to an object and converting from an object to the HTTP response body.
- The RequestMappingHandlerAdapter supports the @RequestBody (and @ResponseBody) annotations with the a set of default HttpMessageConverters

- The <mvc:annotation:driven /> configuration in the DispatcherServlet context configuration will enable the RequestMappingHandlerMapping and RequestMappingHandlerAdapter beans.

@ResponseBody

- This annotation can be put on a method and indicates that the return type should be written straight to the HTTP response body (and not placed in a Model, or interpreted as a view name). For example:

```
@RequestMapping(value = SID_CARRIER_LIST_URL, method={RequestMethod.GET})
public @ResponseBody List<SidCarrierInfo> getSidCodeList(
                @RequestParam(value="sidCode", required=true) String sidCode)
{
        LOG.debug("sidCode entered: " + sidCode);
        List<SidCarrierInfo> result = helperService.getSidCodeList(sidCode);
        LOG.debug("found " + result.size() + " sid/carrier objects.");
        return result;
}
```

- The above example will result in the object "SidCarrierInfo" representation (Example: JSON, XML etc..,) being written to the HTTP response stream.
- As with @RequestBody, Spring converts the returned object to a response body by using an HttpMessageConverter.

@ModelAttribute

- An @ModelAttribute on a method argument indicates the argument should be retrieved from the model.
- For the object in the model, the argument's fields should be populated from all request parameters that have matching names. This is known as data binding in Spring MVC.

```
@RequestMapping(value = CARRIER_DETAIL_URL, method = {RequestMethod.POST})
public @ResponseBody ModelAndView editCarrier(
            @ModelAttribute(value="carrier") @Valid Carrier editCarrier,
            BindingResult bindingResult)
{
            //
}
```

- This is usually used when the form-backing (or command) object is filled up and submitted from the client form.

@InitBinder

- Annotating controller methods with @InitBinder allows you to configure web data binding within your controllers
- @InitBinder identifies methods that initialize the WebDataBinder that will be used to populate command and form object arguments of annotated handler methods

- The following example demonstrates the use of @InitBinder to configure a CustomDateEditor for all java.util.Date **form** properties. It also sets the validator implementation to the binder (More on this later)

```
@InitBinder
public void initBinder(WebDataBinder binder) {
        binder.setValidator(carrierValidator);

    SimpleDateFormat dateFormat = new SimpleDateFormat("MM/dd/yyyy");
    dateFormat.setLenient(false);

    binder.registerCustomEditor(Date.class, new CustomDateEditor(dateFormat, true));
    binder.registerCustomEditor(Boolean.class, new CustomBooleanEditor(false));
    binder.registerCustomEditor(String.class, new StringTrimmerEditor(true));
}
```

@Valid (JSR 303)

- In Spring, you can enable "mvc:annotation-driven" to support JSR303 bean validation via @Valid annotation on the handler method parameter, if any JSR303 validator framework is in the classpath
- Hibernate Validator is the RI for JSR303

- On the model class, add hibernate validator annotations such as @NotNull, @NotEmpty, @Range, @Min, @Max etc…,

```
public class Carrier {

        @NotNull
        @Min(0)
        @Max(9999)
        private int carrierId;

        @NotNull
        @Pattern(regexp="^([Cc]\\d{4})")
        private String carrierCode;

        @NotNull
        @Size(min=1, max=80)
        private String carrierName;

        ....
}
```

0 to 9999

@Pattern (regex)
Start with "C" or "c"
Followed by exactly 4 digits

1 to 80 chars

@Valid (JSR 303)

- An @RequestBody method parameter can be annotated with @Valid, in which case it will be validated using the configured Validator instance.
- The validator was initialized using @InitBinder (from previous slides)

```
@RequestMapping(value = CARRIER_DETAIL_URL, method = {RequestMethod.POST})
public @ResponseBody ModelAndView editCarrier(
                @ModelAttribute(value="carrier") @Valid Carrier editCarrier,
                BindingResult bindingResult)
{
        if(!bindingResult.hasErrors())
        {
                // No errors
        }
        else {
                LOG.debug("Carrier has validation error:" + bindingResult.toString());
        }
        return new ModelAndView("carrier/carrierDetail", "carrier", editCarrier);
}
```

- Validator runs the validation code and based on PASS/FAIL, it sets the BindingResult
- BindingResult will now examined in the Controller as shown above for any errors

@Valid (JSR 303) – JSP

- If there are any validation errors, they will be automatically bind to the model object
- On the JSP Page, using Spring custom tags, we can display the error messages:

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

<form:form method="POST" commandName="customer" action="customer/signup">
		<form:errors path="*" cssClass="errorblock" element="div" />
		<table>
			<tr>
				<td>Customer Name :</td>
				<td><form:input path="name" /></td>
				<td><form:errors path="name" cssClass="error" /></td>
			</tr>
...
</form>
```

@Valid (JSR 303) – Error Messages

- There are default error messages that will be displayed for each of the JSR 303 validation annotations.
- This can be overridden in the message resources as key value pairs.
  - Key is @annotationName.object.feildName
    - objectname here refers to the command/model object name (reference) used in the JSP

```
@NotNull
@Size(min=1, max=80)
private String carrierName;

Size.carrier.carrierName=Carrier Name size must be between {1} and {2} characters

@NotNull
private Date startDate;

NotValid.carrier.startDate=Effective Date cannot occur after End Date

@NotNull
@Pattern(regexp="^([Cc]\\d{4})")
private String carrierCode;

Pattern.carrier.carrierCode=Carrier Code must begin with C followed by 4 digits
```

References

- Theory
  - http://docs.spring.io/spring/docs/3.0.x/reference/mvc.html
  - http://docs.spring.io/spring/docs/2.5.6/reference/mvc.html
- Examples
  - http://www.mkyong.com/tutorials/spring-mvc-tutorials/

Back Up Slides for Spring MVC 2.5.x

SimpleFormController

Property Name	Default	Description	Found In
formView	null	The name of the view that contains the form.	SimpleFormController
successView	null	The name of the view to display on successful completion of form submission.	SimpleFormController
bindOnNewForm	false	Should parameters be bound to the form bean on initial views of the form? (Parameters will still be bound on form submission.)	AbstractFormController
sessionForm	false	Should the form bean be stored in the session between form view and form submission?	AbstractFormController
commandName	"command"	Logical name for the form bean.	BaseCommandController
commandClass	null	The class of the form bean.	BaseCommandController
validator(s)	null	One or more Validators that can handle objects of type configured by commandClass property.	BaseCommandController
validateOnBinding	true	Should the Controller validate the form bean after binding during a form submission?	BaseCommandController

formView

1. Create an instance of the form object (a.k.a command bean)
   - Using the **formBackingObject()** method
   - By default this method will return an instance of the class specified with setCommandClass() (remember commandName & commandClass properties?)
   - formBackingObject() can be overridden to manipulate the form object before it enters the formView
2. Creates the DataBinder and calls the **initBinder()**
   - Override this method to register any custom PropertyEditors required when binding to the form object
3. At this point, the view (formView) is ready to be rendered to the user. But before this…
   - **referenceData()** call-back method is called
   - This life cycle method allows you to assemble and return any auxiliary objects required to render the view. The form object will automatically be sent to the view, so use this method to put anything else into the model the form page might need.

Form Submission with the **onSubmit()** method

1. Create an instance of the form object (a.k.a command bean)
   - Using the formBackingObject() method
2. Creates the DataBinder and calls the initBinder()
   - Override this method to register any custom PropertyEditors required when binding to the form object

With the form bean created , and the DataBinder created , the request parameters are now bound to the form bean

3. Allow each configured validator (thru the **validator** property) to validate the form bean object
4. After validation, the controller makes a decision based on whether any error exists
   - If there are any errors – display the original form view
     - The referenceData() method is called once more to populate the model with object for the form. Finally, the form view is displayed again, with the errors and the form bean
   - If no error exists – Then the form bean can finally be processed in the onSubmit()method

Spring custom tags

<spring:bind path="comamndName.attributeName" > .... </spring:bind>

```
<spring:bind path="carrier.carrierName">
    <td><input name="<c:out value="${status.expression}"/>" value="<c:out value="${status.value}"/>" /></td>
</spring:bind>
```

The <spring:bind> tag extracts information about a single field ("carrierName" in this example) from the command object (named "carrier" in this case) and places it in a status variable.
${status.expression} contains the name that the field should have and ${status.value} contains the value of the field (blank initially, but populated with previously submitted values if the form is redisplayed after submission errors).

New Way

```
<form:form commandName="carrier">
    <td><form:input path="carrierName" /></td>
</form:form>
```

The "comandName" attribute tells the form the name of the command object that contains our form data (specified in your form controller by calling setCommandName())
The "path" attribute gives the command object property that will be bound to the input field. Unlike the "path" attribute in <spring:bind> I didn't have to specify the command name here because the <form:form> tag already identifies the command object we'll binding to.