

SPRING BASICS & IOC

Introduction to Spring Framework

- December 1996 – JavaBeans makes its appearance.
 - Intended as a general-purpose means of defining reusable application components
 - Used more as a model for building user interface widgets
- Sophisticated applications often require services not directly provided by the JavaBeans specification
- March 1998 – EJB was published.
 - But EJBs are complicated in a different way, that is, they mandate deployment descriptors and plumbing code
- Many successful applications were built based on EJB
 - But EJB never really achieved its intended purpose, which is to simplify enterprise application development
- Java development comes full circle
 - New programming techniques like including aspect-oriented programming (AOP) and inversion of control (IoC) are giving JavaBeans much of the power of EJB

Quick history :

- Spring Framework project founded in Feb 2003
- Release 1.0 in Mar 2004
- Release 1.2 in May 2005
- Release 2.0 in Oct 2006
- Release 2.5 in Nov 2007
- Release 3.0 in Dec 2009

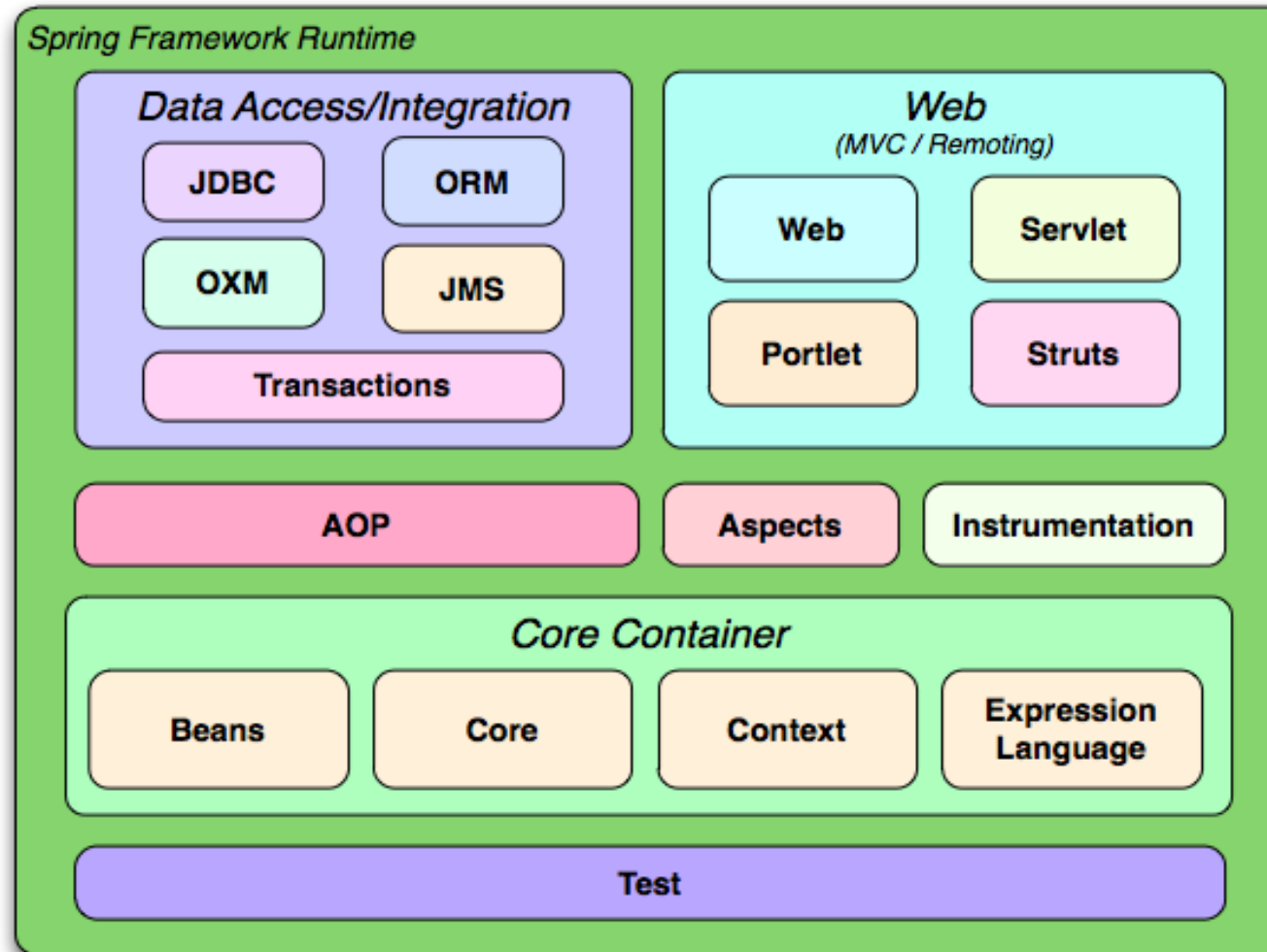
What is Spring?

- Spring is an open source framework created by Rod Johnson, Juergen Hoeller et al
 - Addresses the complexity of enterprise application development
 - Any java application can benefit from Spring in terms of simplicity, testability and loose coupling
- Spring is a lightweight inversion of control and aspect-oriented container framework
 - Lightweight: in terms of both size and overhead
 - Inversion of control: promotes loose coupling
 - Aspect-oriented: enables cohesive development by separating application business logic from system services
 - Container: contains and manages the life cycle and configuration of application objects
 - Framework: possible to configure and compose complex applications from simpler components

Why Spring?

- Spring simplifies Java development
- With Spring, complexity of application is proportional to the complexity of the problem being solved
- Essence of Spring is to provide enterprise services to POJO.
- Spring employs four key strategies:
 - Lightweight and minimally invasive development with plain old Java objects (POJOs)
 - Loose coupling through dependency injection and interface orientation
 - Declarative programming through aspects and common conventions
 - Boilerplate reduction through aspects and templates

Spring 3.0 architecture



Spring Jumpstart with HelloWorld

```
package training.spring;
public class HelloWorld {
    public void sayHello(){
        System.out.println("Hello Spring 3.0");
    }
}
```

```
<?xml .....>
<beans .....>
<bean id="HWBean" class =
    "training.spring.HelloWorld" />
</beans>
```

The Spring
configuration file

```
public class HelloWorldClient {
    public static void main(String[] args) {
        XmlBeanFactory beanFactory = new XmlBeanFactory
            (new ClassPathResource("HelloWorld.xml"));
        HelloWorld bean = (HelloWorld) beanFactory.getBean("HWBean");
        bean.sayHello();
    }
}
```

Output:
Hello Spring 3.0

Injecting dependencies via setter methods

```
public interface CurrencyConverter {  
    public double dollarsToRupees(double dollars);  
}
```

```
public class CurrencyConverterImpl implements CurrencyConverter {  
    private double exchangeRate;  
    public double getExchangeRate() { return exchangeRate; }  
    public void setExchangeRate(double exchangeRate) {  
        this.exchangeRate = exchangeRate;    }  
    public double dollarsToRupees(double dollars) {  
        return dollars * exchangeRate;  
    }  
}
```



Injecting dependencies via setter methods

```
<?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:aop="http://www.springframework.org/schema/aop"
xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

    <bean id="currencyConverter"
        class="training.Spring.CurrencyConverterImpl">
        <property name="exchangeRate" value="44.50" />
    </bean>
</beans>
```

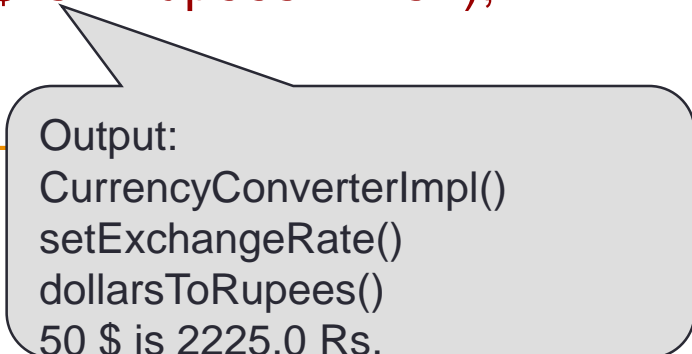
The configuration file
(CurrencyConverter.xml)

Injecting dependencies via setter methods



The client application

```
public class CurrencyConverterClient {  
    public static void main(String args[]) throws Exception {  
        Resource res = new ClassPathResource("currencyconverter.xml");  
        BeanFactory factory = new XmlBeanFactory(res);  
        CurrencyConverter curr = (CurrencyConverter)  
            factory.getBean("currencyConverter");  
        double rupees = curr.dollarsToRupees(50.0);  
        System.out.println("50 $ is "+rupees+" Rs.");  
    }  
}
```



Output:
CurrencyConverterImpl()
setExchangeRate()
dollarsToRupees()
50 \$ is 2225.0 Rs.

Injecting dependencies via constructor

- Bean classes can be programmed with constructors that take enough arguments to fully define the bean at instantiation

```
<bean id="currencyConverter"  
      class="training.Spring.CurrencyConverterImpl">  
  <constructor-arg>  
    <value> 44.50 </value>  
  </constructor-arg>  
</bean>
```

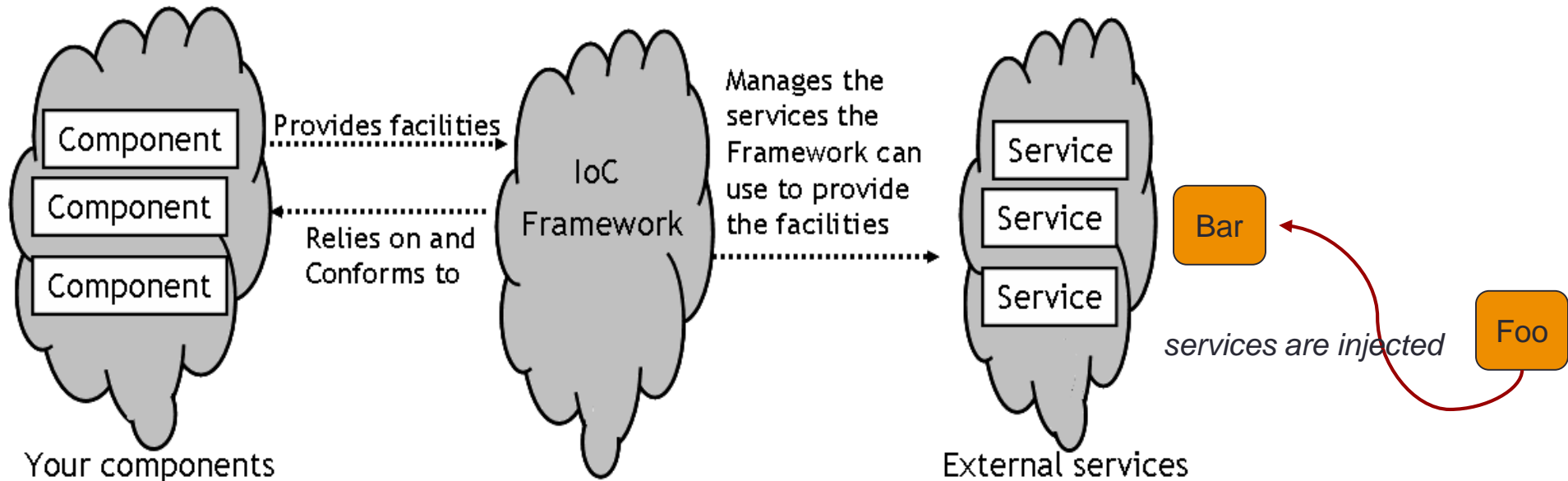
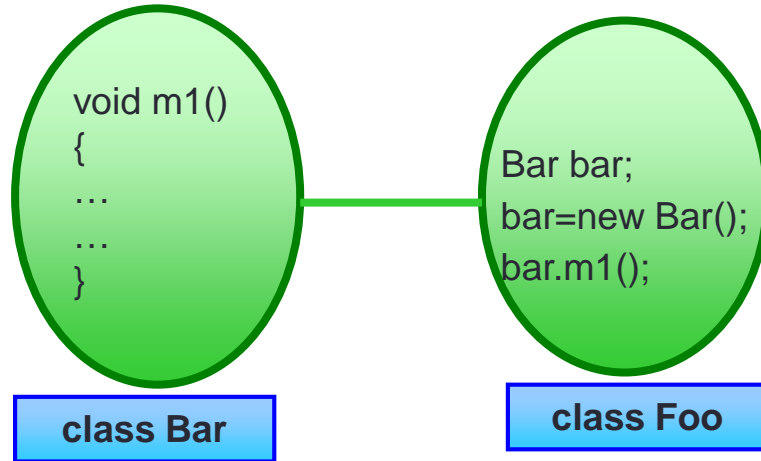
```
public CurrencyConverterImpl(double er) {  
  exchangeRate = er;  
}
```

Injecting dependencies via constructor

- If a constructor has multiple arguments, then ambiguities among constructor arguments can be dealt with in two ways :
 - by index
 - by type

```
<beans>
  <bean id="currencyConverter"
        class="training.Spring.CurrencyConverterImpl3">
    <constructor-arg><value>44.25</value></constructor-arg>
    <!--<constructor-arg index="0"><value>44.25</value></constructor-arg>-->
    <!--<constructor-arg type="double"><value>44.25</value></constructor-arg>-->
  </bean>
</beans>
```

Wiring beans – Inversion of Control (IoC)



IoC, Beans and BeanFactories

- Used to achieve loose coupling between several interacting components in an application.
 - The IoC framework separates facilities that your components are dependent upon and provides the “glue” for connecting the components.
- BeanFactory is the core of Spring’s DI container.
 - In Spring, the term “bean” is used to refer to any component managed by the container.
- IoC pattern uses three different approaches to achieve decoupling of control of services from components:
 - Type 1 : Interface injection
 - Type 2 : Setter Injection
 - Type 3 : Constructor injection

IoC in action: Wiring Beans

- The act of creating associations between application components is known as wiring.
- In Spring, there are many ways of wiring components together, but most commonly used is XML. An example:

```
<bean id="exchangeService" class="ExchangeServiceImpl" />
<bean id="currencyConverter" class="CurrencyConverterImpl">
  <property name="exchangeService">
    <ref bean="exchangeService" />
  </property>
  <!--<property name="exchangeService">
    <ref local="exchangeService" /> </property> -->
  <!--<property name="exchangeService">
    <idref local="exchangeService" /> </property> -->
</bean>
```

Prototyping Vs Singleton

- By default, all Spring beans are singletons.
 - But each time a bean is asked for, prototyping lets the container return a new instance. This is achieved through the scope attribute of <bean>
 - Example: `<bean id="foo" class="com.spring.Foo" scope="prototype" />`
- Additional Bean scopes: request, session , global-session

Scope	What it does
singleton	Scopes the bean definition to a single instance per Spring container (default).
prototype	Allows a bean to be instantiated any number of times (once per use).
request	Scopes a bean definition to an HTTP request. Only valid when used with a web-capable Spring context (such as with Spring MVC).
session	Scopes a bean definition to an HTTP session. Only valid when used with a web-capable Spring context (such as with Spring MVC).
global-session	Scopes a bean definition to a global HTTP session. Only valid when used in a portlet context.

Inner beans

- Another way of wiring bean references is to embed a <bean> element directly in the <property> element

```
<bean id="currencyConverter" class="CurrencyConverterImpl">  
  <property name="exchangeService">  
    <bean class="ExchangeServiceImpl" />  
  </property>  
</bean>
```

- The drawback here is that the instance of inner class cannot be used anywhere else; it is an instance created specifically for use by the outer bean.

Autowiring

- Autowiring allows Spring to wire all bean's properties automatically by setting the autowire property on each <bean> that you want autowired

```
<bean id="foo" class="com.spring.Foo" autowire="autowire type" />
```

- Four types

no	(Default) No autowiring.
byName	Autowiring by property name. Spring looks for a bean with the same name as the property that needs to be autowired
byType	Allows a property to be autowired if exactly one bean of the property type exists in the container. If more than one exists, a fatal exception is thrown, which indicates that you may not use byType autowiring for that bean. If there are no matching beans, nothing happens; the property is not set.
constructor	Analogous to byType, but applies to constructor arguments. If there is not exactly one bean of the constructor argument type in the container, a fatal error is raised.

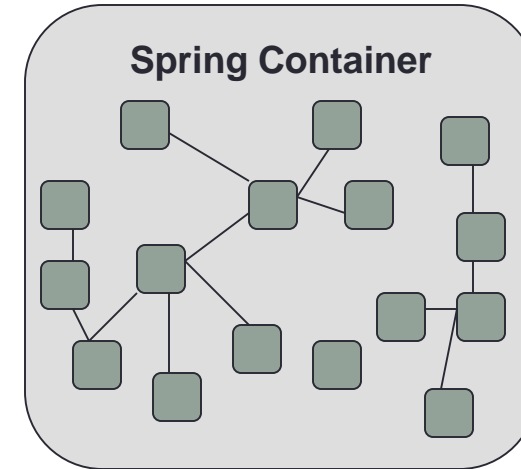
Using collections for injection

- If beans need access to collections of objects, rather than individual beans/values.
 - Spring allows you to inject a collection of objects into your beans.
 - You can choose either <list>, <map>, <set> or <props> to represent a List, Map, Set or Properties instance.

```
<bean id="complexObject" class="example.ComplexObject">
  <property name="people">
    <props>
      <prop key="HarryPotter">The magic property</prop>
      <prop key="JerrySeinfeld">The funny property</prop>
    </props>
  </property>
  <property name="someList">
    <list>
      <value>red</value>
      <value>blue</value>
    </list>
  </property>
  <property name="someMap">
    <map>
      <entry key="an entry" value="just some string"/>
      <entry key="a ref" value-ref="myDataSource"/>
    </map>
  </property></bean>
```

Bean containers: concept

- The container or bean factory is at the core of the Spring framework and uses IoC to manage components.
- Bean factory is responsible to create and dispense beans.
- It takes part in the life cycle of a bean, making calls to custom initialization and destruction methods, if those methods are defined.
- Spring has two types of containers:
 - Bean factories that are the simplest, providing basic support for dependency injection
 - Application contexts that build on bean factory by providing application framework services



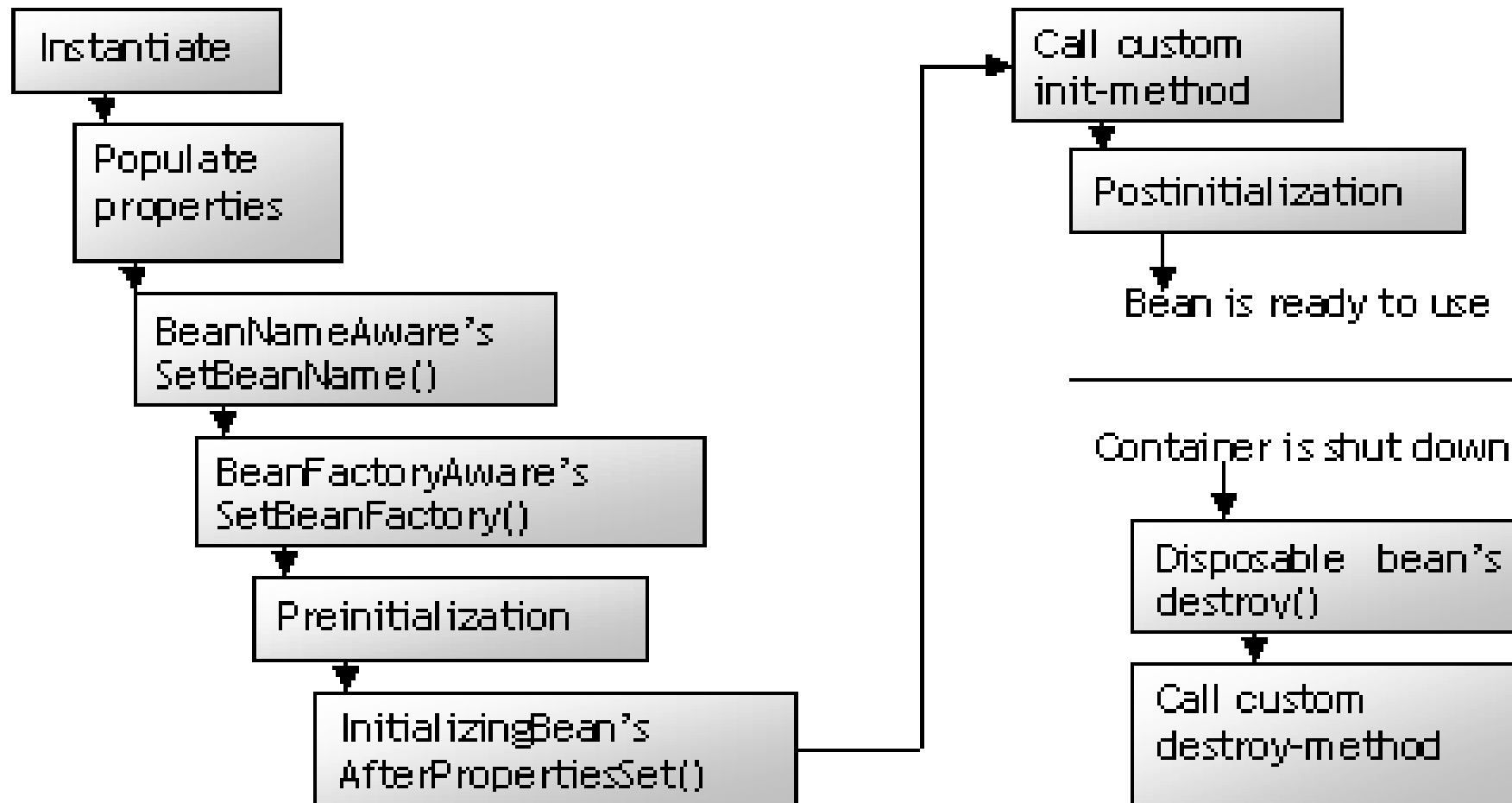
Bean containers: The BeanFactory

- BeanFactory interface is responsible for managing beans and their dependencies
 - Its `getBean()` method allows you to get a bean from the container by name
- It has a number of implementing classes:
 - `DefaultListableBeanFactory`
 - `SimpleJndiBeanFactory`
 - `StaticListableBeanFactory`
 - `XmlBeanFactory`
- The `XmlBeanFactory` : One of the most useful implementations of the bean factory is instantiated via explicit user code as:

```
Resource res = new FileSystemResource("beans.xml");  
XmlBeanFactory factory = new XmlBeanFactory(res);
```

```
Resource res = new ClassPathResource("beans.xml");  
XmlBeanFactory factory = new XmlBeanFactory(res);
```

Life cycle of Beans in Spring factory container



Initialization and Destruction

- When a bean is instantiated, some initialization can be performed to get it to a usable state
- When the bean is removed from the container, some cleanup may be required
- Spring can use two life-cycle methods of each bean to perform this setup and teardown.
- Example:

```
<bean id="foo" class="com.spring.Foo"  
        init-method="setup"  
        destroy-method="teardown" />
```

InitializingBean and DisposableBean

- InitializingBean interface
 - provides afterPropertiesSet() method which is called once all specified properties for the bean have been set.
- DisposableBean interface
 - provides destroy() method which is called when the bean is disposed by the container
- The advantage is that Spring container is able to automatically detect beans without any external configuration.
- The drawback is that the applications' beans are coupled to Spring API.

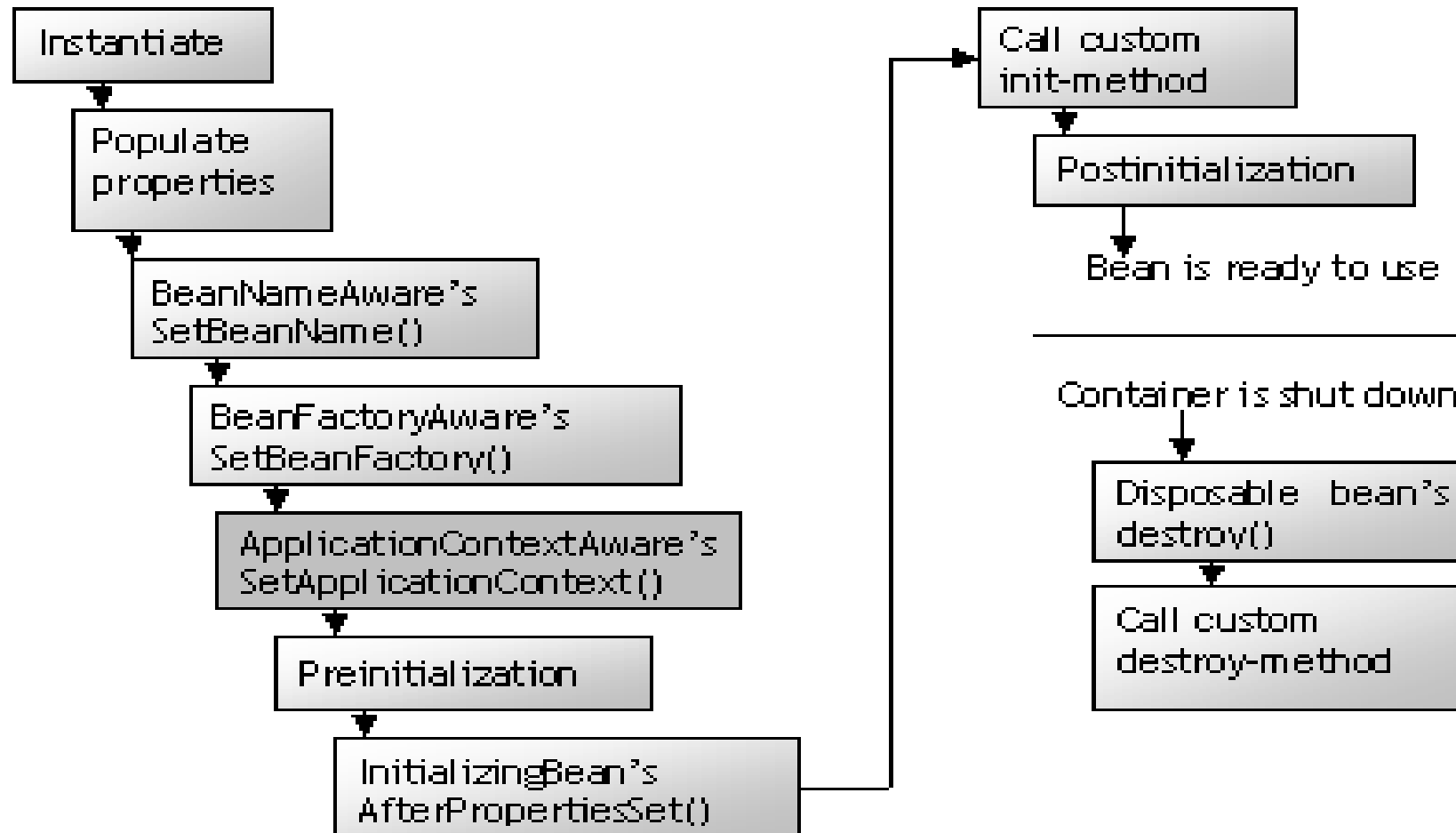
Bean containers : Application context

- Provides application framework services such as :
 - Resolving text messages, including support for internationalization of these messages
 - Load file resources, such as images
 - Publish events to beans that are registered as listeners
- Many implementations of application context exist:
 - ClassPathXmlApplicationContext
 - FileSystemApplicationContext
 - XmlWebApplicationContext

Some examples :

```
ApplicationContext ctx = new ClassPathXmlApplicationContext("app.xml");  
ApplicationContext ctx = new  
FileSystemXmlApplicationContext("/some/file/path/app.xml");  
ApplicationContext ctx = new ClassPathXmlApplicationContext( new  
String[]{"app1.xml","app2.xml"}); // combines multiple xml file fragments
```

ApplicationContext life cycle



Customizing beans with Post Processors

- Post processing involves cutting into a bean's life cycle and reviewing or altering its configuration.
- Occurs after some event has occurred.
- Spring provides two interfaces :
 - BeanPostProcessor interface
 - BeanFactoryPostProcessor interface
- ApplicationContext automatically detects Bean Post-Processor, but these have to manually be explicitly registered for bean factory.

```
ConfigurableBeanFactory bf = new .....; // create BeanFactory
// now register some beans and any needed BeanPostProcessors
MyBeanPostProcessor pp = new MyBeanPostProcessor();
bf.addBeanPostProcessor(pp); // now start using the factory ...
```

Customizing beans with BeanFactoryPostProcessor

- BeanFactoryPostProcessor performs post processing on the entire Spring container.
- It has a single method, which is postProcessBeanFactory().
- Spring offers a number of pre-existing bean factory post-processors:
 - AspectJWeaving
 - CustomAutowireConfigurer
 - CustomEditorConfigurer
 - CustomScopeConfigurer
 - PropertyPlaceholderConfigurer
 - PreferencesPlaceholderConfigurer
 - PropertyOverrideConfigurer

PropertyPlaceholderConfigurer

- It is possible to configure entire application in a single bean wiring file.

```
<bean id="datasource" class="com.spring.ConnectionDataSource" >
    <property name="url">
        <value> jdbc:hsqldb:training </value>
    </property>
    <property name="driverclassname">
        <value> org.hsqldb.jdbcDriver </value>
    </property>
    ....
</bean>
```

- But, sometimes it is beneficial to extract certain pieces of that configuration into a separate property file.

PropertyPlaceholderConfigurer

- Externalizing properties using PropertyPlaceholderConfigurer indicates Spring to load certain configuration from an external property file.

```
<bean id="placeholderConfig" class="org.springframework.beans.  
    factory. config.PropertyPlaceholderConfigurer">  
    <property name="location" value="data.properties" />  
</bean>  
<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource">  
    <property name="driverClassName" value="${jdbc.driverClassName}"/>  
    <property name="url" value="${jdbc.url}"/>  
    <property name="username" value="${jdbc.username}"/>  
    <property name="password" value="${jdbc.password}"/>  
</bean>
```

```
jdbc.driverClassName=oracle.jdbc.driver.OracleDriver  
jdbc.url=jdbc:oracle:thin:@192.168.224.26:1521:trgdb  
.....
```

Internationalization: Resolving text messages

- ApplicationContext interface provides messaging functionality by extending MessageSource interface.
- getMessage() is a basic method used to retrieve a message from the MessageSource.
- On loading, ApplicationContext automatically searches for a MessageSource bean defined in the context.
- ResourceBundleMessageSource is a ready-to-use implementation of MessageSource.

Annotation-based configuration

- Spring has a number of custom annotations:
 - @Required
 - @Autowired
 - @Resource
 - @PostConstruct
 - @PreDestroy
- Annotations to configure beans:
 - @Component
 - @Controller
 - @Repository
 - @Service
- Some other transactions:
 - @Transactional
 - @AspectJ

@Autowired annotation

```
<?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:context="http://www.springframework.org/schema/context"
xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-3.0.xsd">
    <context:annotation-config />

    <context:component-scan base-package="training.spring" />

    <!-- bean declarations go here -->
</beans>
```

SPRING MVC FRAMEWORK

Lesson Objectives

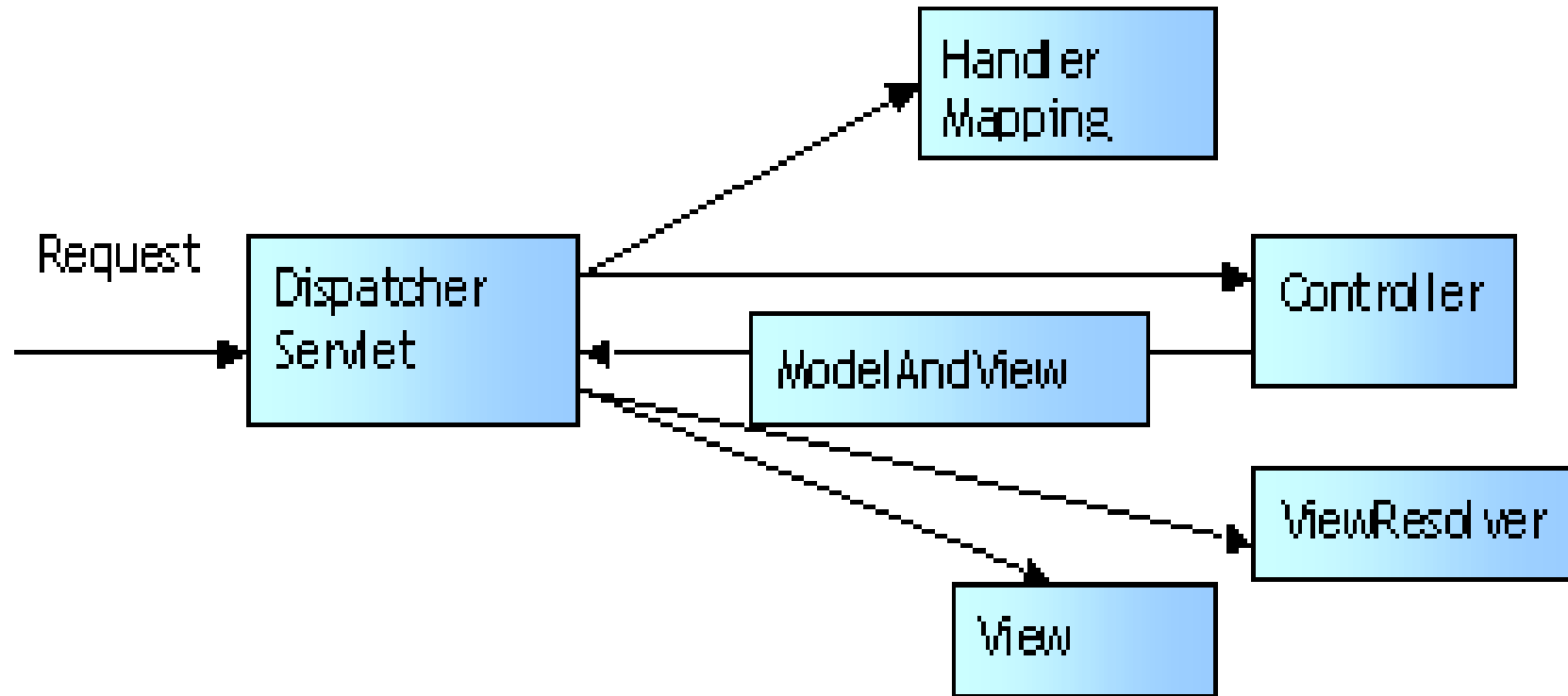
- Introduction to Spring MVC framework
 - Learn how to develop web applications using Spring
 - Understand the Spring MVC architecture and the request cycle of Spring web applications
 - Understand components like handler mappings, ViewResolvers and controllers
 - Use MVC Annotations like @Controller, @RequestMapping and @RequestParam

Spring MVC Framework Features

- Provides you with an out-of-the-box implementations of workflow typical to web applications
- Allows you to use a variety of different view technologies
- Enables you to fully integrate with your Spring based, middle-tier logic through the use of dependency injection
- Displays modular framework, with each set of components having specific roles and completely decoupled from the rest of the framework

Spring MVC lifecycle

- Life cycle of a Request in Spring MVC



Configuring the DispatcherServlet in web.xml

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

<servlet-mapping>
  <servlet-name>basicspring</servlet-name>
  <url-pattern>*.obj</url-pattern>
</servlet-mapping>
```

The servlet-name
given to the servlet
is significant

- Steps to build a homepage in Spring MVC:
 - Write the controller class that performs the logic behind the homepage
 - Configure controller in the DispatcherServlet's context configuration file
 - Configure a view resolver to tie the controller to the JSP
 - Write the JSP that will render the homepage to the user

Breaking Up the Application Context

- **Configuring the ContextLoaderListener in web.xml**

```
<listener>  
  <listener-class>  
    org.springframework.web.context.ContextLoaderListener  
  </listener-class>  
</listener>
```

- **Setting ContextConfigLocation Parameter**

```
<context-param>  
  <param-name>contextConfigLocation</param-name>  
  <param-value>  
    /WEB-INF/basicspring-service.xml  
    /WEB-INF/basicspring-data.xml  
  </param-value>  
</context-param>
```

Building a Homepage in Spring MVC

- Step 1 : Write the controller class that performs the logic
 - Controller handles request and performs business logic
 - The `handleRequest()` returns a `ModelAndView` object

```
public class BasicSpringController implements Controller {  
    public ModelAndView handleRequest(HttpServletRequest request,  
        HttpServletResponse response) throws ServletException, IOException{  
        String now = new java.util.Date().toString();  
        return new ModelAndView("hello","now",now);  
    }  
}
```

- Step 2: Configure the controller in the `DispatcherServlet`'s context configuration file

```
<bean id="basicspringController"  
      class="training.web.spring.BasicSpringController" />
```


Building a Homepage in Spring MVC (Contd..)

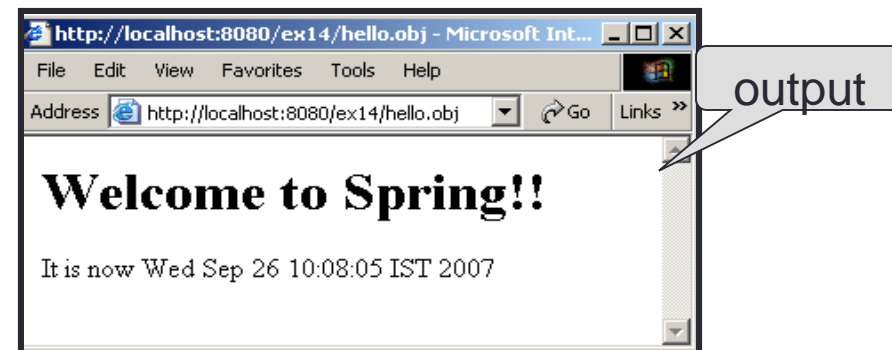
- Step 3 : Configure a view resolver to bind controller to the JSP

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

} /hello.jsp

- Step 4 : Write the JSP that will render the homepage to the user

```
<html>  
  <body>  
    <h1>Welcome to Spring!! </h1>  
    Now it is ${now}  
  </body>  
</html>
```



ModelAndView Class

- This class fully encapsulates the view and model data that is to be displayed by the view. Eg:

```
ModelAndView("hello","now",now);
```

```
Map myModel = new HashMap();  
myModel.put("now",now);  
myModel.put("products",getProductManager().getProducts());  
return new ModelAndView("product","model",myModel);
```

- Every controller returns a ModelAndView
- Views in Spring are addressed by a view name and are resolved by a view resolver

Resolving Views : The ViewResolver

View resolver	How it works
InternalResourceViewResolver	Resolves logical view names into View objects that are rendered using template file resources
BeanNameViewResolver	Looks up implementations of the View interface as beans in the Spring context, assuming that the bean name is the logical view name
ResourceBundleViewResolver	Uses a resource bundle that maps logical view names to implementations of the View interface
XmlViewResolver	Resolves View beans from an XML file that is defined separately from the application context definition files

Demo: Example 14 - BasicSpring

- Demo-1: This demo shows the cycle of a request in Spring MVC.
- Execute :
 - /hello.obj URL
- Demo 2 : This demo demonstrates the creation and returning of a model object.
- Execute:
 - /products.obj URL



Implementing Controllers

@Controller

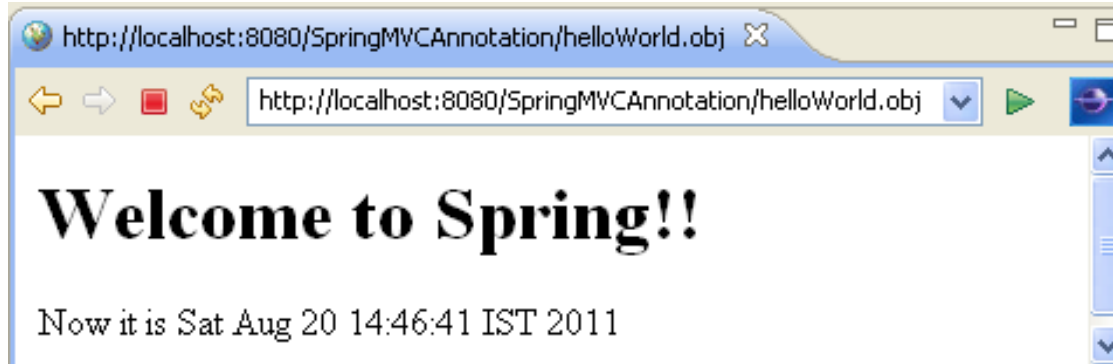
```
public class HelloController1 {  
    @RequestMapping("/helloWorld")  
    public ModelAndView handleRequest(HttpServletRequest request,  
                                     HttpServletResponse response) throws ..... {  
        String now = new java.util.Date().toString();  
        return new ModelAndView("hello", "now", now);  
    }  
}
```

@Controller

```
public class HelloController {  
    @RequestMapping("/helloWorld")  
    public String handleMyRequest(Map<String, Object> model) {  
        String now = new java.util.Date().toString();  
        model.put("now", now);  
        return "hello";  
    }  
}
```

Demo

- Execute the applications in the SpringMVCAnotation web project



Handling User Input

```
@Controller
public class LoginFormController {
    @RequestMapping(value = "/login", method = RequestMethod.GET)
    public String onSubmit(@RequestParam("username") String username,
        @RequestParam("password") String password, Model model) {

        model.addAttribute("username", username);
        if (username.equals("majrul") && password.equals("majrul123"))
            return "success";
        else return "failure";
    }
}
```



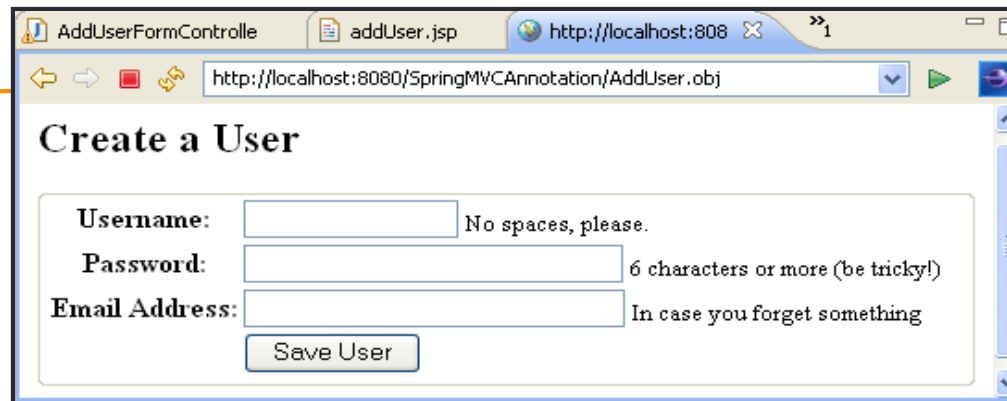
Demo

- <http://localhost:8080/SpringMVCAnotation/login.obj?username=majrul&password=majrul123>
- Vary the username and password to get the failure.jsp page.



Processing forms : The controller class

```
@Controller
public class AddUserController {
    @RequestMapping(value = "/AddUser", method = RequestMethod.GET)
    public String showForm(Model model) {
        model.addAttribute(new User());
        return "addUser";
    }
    @RequestMapping(method = RequestMethod.POST)
    public String processForm(@Valid User user, BindingResult bindingResult) {
        if (bindingResult.hasErrors()) return "failure";
        else {
            // some logic to persist user
            return "success";
        }
    }
}
```



The screenshot shows a web browser window with the address bar displaying `http://localhost:8080/SpringMVCAnotation/AddUser.obj`. The page title is "Create a User". The form contains three input fields with associated validation messages:

- Username:** No spaces, please.
- Password:** 6 characters or more (be tricky!)
- Email Address:** In case you forget something

A "Save User" button is located at the bottom of the form.

addUser.jsp

Processing forms : The JSP

addUser.jsp

```
<%@ taglib prefix="sf" uri="http://www.springframework.org/tags/form"%>
<sf:form method="POST" modelAttribute="user" >
<table cellpadding="0">
<tr>
<th><sf:label path="username">Username:</sf:label></th>
<td><sf:input path="username" size="15" maxlength="15" />
    <small id="username_msg">No spaces, please.</small><br />
    <sf:errors path="username" /></td>
</tr>
<tr>
    <th><sf:label path="password">Password:</sf:label></th>
    <td><sf:password path="password" size="30" showPassword="true"/>
        <small>6 characters or more (be tricky!)</small><br />
        <sf:errors path="password" />
    </td>
</tr>
<tr><th></th>
<td><input name="commit" type="submit" value="Save User" /></td></tr>
</sf:form></div>
```

Validating input : declaring validation rules

```
public class User {  
    @Size(min = 3, max = 20, message = "Username must be between 3 and 20  
characters long.")  
    @Pattern(regex = "[a-zA-Z0-9]+$", message = "Username must be alphanumeric  
with no spaces")  
    private String username;  
  
    @Size(min = 6, max = 20, message = "The password must be at least 6 characters  
long.")  
    private String password;  
  
    @Pattern(regex = "[A-Za-z0-9]+@[A-Za-z0-9.-]+[.][A-Za-z]{2,4}", message =  
"Invalid email address.")  
    private String email;  
  
    //getter and setter methods for all these properties  
}
```

Displaying validation errors

jsp

```
<td>
  <sf:password path="password" size="30" showPassword="true"/>
  <small>6 characters or more (be tricky!)</small><br/>
  <sf:errors path="password" />
</td>
```

controller

```
public String processForm(@Valid User user, BindingResult
bindingResult) {
    if (bindingResult.hasErrors()) {
        return "failure";
    }
    .....
}
```

SPRING EXPRESSION LANGUAGE

What is SpEL?

- The Spring Expression Language is a powerful expression language that supports querying and manipulating an object graph at runtime.
- SpEL supports many functionalities including:
 - Literal expressions
 - Boolean and relational operators
 - Regular and class expressions
 - Accessing properties, arrays, lists, maps
 - Method invocation
 - Calling constructors
 - Bean references
 - Array construction
 - Inline lists
 - User defined functions
 - Templated expressions

Exploring literals and Types

- Working with Literals:

- Wire SpEL expression into a bean's property by using `#{ <exprn-string> }`

```
<property name="count" value="#{5}"/>  
<property name="message" value="The value is #{5}"/>  
<property name="frequency" value="#{89.7}"/>  
<property name="name" value="#{'Chuck'}/>
```

examples

- Working With Types:

- Use the `T()` operator to work with class-scoped methods & constants
- Example: `T(java.lang.Math)` //expresses Java's Math class in SpEL

```
<property name="multiplier" value="#{T(java.lang.Math).PI}"/>  
<property name="randomNumber" value="#{T(java.lang.Math).random()}/>
```

examples

Referencing Beans, Properties, And Methods

- SpEL allows to wire one bean into another bean's property by using the bean ID as the SpEL expression:

```
<property name="exchangeService " value="#{exchangeService}"/>
```

```
<bean id="currencyConverter" class="training.CurrencyConverterImpl">  
  <property name="exchangeRate"  
    value="#{exchangeService.exchangeRate}" />  
</bean>
```

Bean ID

Property name

referencing
beans
properties

```
<property name="exchangeService "  
  value="#{exchangeService.getExchangeRate()}"/>
```

Invoking
methods

Performing operations on SpEL values

- SpEL includes several operators to manipulate the values of an expression.

Operation type	Operators
Arithmetic	+, -, *, /, %, ^
Relational	<, >, ==, <=, >=, lt, gt, eq, le, ge
Logical	and, or, not,
Conditional	?: (ternary), ?: (Elvis)
Regular expression	matches

examples

```
<property name="adjustedAmount" value="#{counter.total + 42}"/>
<property name="result" value="#{2 * T(java.lang.Math).PI * circle.radius}"/>
<property name="fullName" value="#{emp.firstName + ' ' + emp.lastName}"/>
<property name="redCustomer" value="#{account.balance le 100000}"/>
<property name="outOfStock" value="#{!product.available}"/>
<property name="outcome"
    value="#{T(java.lang.Math).random() > .5 ? 'win' : 'lose'}"/>
```

Working with collections

```
package trg.spring;  
public class City {  
    private String name;  
    private String state;  
    private int population;  
    //setter and getter methods for each of these properties  
}
```

```
<util:list id="cities">  
    <bean class="trg.spring.City"  
        p:name="Chicago" p:state="IL" p:population="2853114"/>  
    <bean class="trg.spring.City"  
        p:name="Atlanta" p:state="GA" p:population="537958"/>  
    <bean class="trg.spring.City"  
        p:name="Dallas" p:state="TX" p:population="1279910"/>  
    .....  
</util:list>
```

Accessing collections

1 <property name="customerCity" value="#{cities[2]}"/>

2 <property name="customerCity" value="#{cities['Dallas']}"/>

3 <property name="userName" value="#{userprops['user.name']}"/>

selection

4 <property name="smallCities" value="#{cities.[population lt 100000]}"/>

5 <property name="cityNames" value="#{cities.![name]}"/>

projection

6 <property name="cityNames" value="#{cities.![name + ', ' + state]}"/>

7 <property name="cityNames"
value="#{cities.[population gt 100000].![name + ', ' + state]}"/>

@Value annotation

```
package training.spring.spel;  
@Component("user")  
public class UserBean {  
    @Value("#{userprops.username}")  
    private String username;  
    @Value("#{userprops.password}")  
    private String password;  
    // setter and getter methods for properties
```

inject values from the
properties files using a
SpEL expression

```
<beans xmlns="http://www.springframework.org/schema/beans"  
..... >  
    <context:component-scan base-package="training.spring.spel" />  
    <util:properties id="userprops" location="classpath:user.properties" />  
</beans>
```

properties file

Agenda

- REST Web Services design guidelines
 - URI Opacity
 - Query String Extensibility
 - Deliver Correct Resource Representation
 - Asynchronous Services
 - Receiving and Sending XML
 - Exception handling and propagation

REST Web Services design guidelines

- URI Opacity
- Query String Extensibility
- Deliver Correct Resource Representation
- Asynchronous Services
- Receiving and Sending XML
- Exception handling and propagation

URI Opacity

- URI should be design in a such a way that and users should not derive metadata from the URI
- The query string and fragment of URI have special meaning that can be understood by users.
- There must be a shared vocabulary between a service and its consumers.

Query String Extensibility

- A service provider should ignore any query parameters it does not understand during processing.
- If it needs to consume other services, it should pass all ignored parameters along.
- This practice allows new functionality to be added without breaking existing services

Deliver Correct Resource Representation

- There are four commonly used ways of delivering the correct resource representation to consumers
 - Server-driven negotiation
 - Client-driven negotiation
 - Proxy-driven negotiation
 - URI-specified representation

Asynchronous Services

- An asynchronous service needs to perform the following:
 - Return a receipt immediately upon receiving a request.
 - Validate the request.
 - If the request is valid, the service must act on the request as soon as possible. It must report an error if the service cannot process the request after a period of time defined in the service contract.

Other Guidelines

- Exception handling
 - To deal with exceptions, Spring 3 provides annotation `@ExceptionHandler`
- XML handling
 - follow the principle of strict out and loose in.

Agenda

- RESTFul Spring Web Services
- Spring 3 REST support
- RESTful service design
 - Server
 - Client application

Spring REST support

- Spring's REST support makes the development of RESTful Web services and applications easier.
 - Spring's REST support is based upon Spring's existing annotation based MVC framework
 - Client-side access to RESTful resources is greatly simplified using Spring ***RestTemplate***
-
- RestTemplate follows in the design pattern Spring's other template classes like JdbcTemplate and JmsTemplate.

Creating RESTful services with Spring

- Spring uses the **@RequestMapping** method annotation to define the URI Template for the request

```
@RequestMapping("/users/{userid}", method=RequestMethod.GET)
public String getUser(@PathVariable String userId) {

-----
-----
}
```

<http://www.myapp.com/users/John>

Creating RESTful services with Spring

- The **@PathVariable** annotation is used to extract the value of the template variables and assign their value to a method variable

```
@RequestMapping("/users/{userid}", method=RequestMethod.GET)
public String getUser(@PathVariable String userId) {

    -----

    -----

}
```

Web services development styles

- The **@RequestBody** method parameter annotation is used to indicate that a method parameter should be bound to the value of the HTTP request body

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

The conversion of the request body to the method argument is done using a **HttpMessageConverter**

Returning multiple representations

- A RESTful architecture may expose multiple representations of a resource.
- There are two strategies for a client to inform the server of the representation it is interested in receiving.
 - The first strategy is to use a distinct URI for each resource
 - The second strategy is set the **Accept** HTTP request header

Strategy 1- Use a distinct URI for each resource

- Requests a PDF representation of the user John

<http://www.myapp.com/users/John.pdf>

- Requests a XML representation of the user John

<http://www.myapp.com/users/John.xml>

Strategy 2- set the Accept HTTP request header

- Requests a PDF representation of the user John

<http://www.myapp.com/users/John>

Accept header set to `application/pdf`

- Requests a XML representation of the user John

<http://www.myapp.com/users/John>

Accept header set to `text/xml`

This strategy is known as content negotiation.

Other useful annotations

- `@RequestParam` to inject a URL parameter into the method.
- `@RequestHeader` to inject a certain HTTP header into the method.
- `@RequestBody` to inject an HTTP request body into the method.
- `HttpEntity<T>` to inject into the method automatically if you provide it as a parameter.
- `ResponseEntity<T>` to return the HTTP response with your custom status or headers.

Exception Handling

- The **@ExceptionHandler** method annotation is used.
- Specifies which method will be invoked when an exception of a specific type is thrown

```
@Controller public class SimpleController {  
    @ExceptionHandler(IOException.class)  
    public String myExpHandler(IOException ex, HttpServletRequest  
    request) { return ClassUtils.getShortName(ex.getClass());  
    }  
}
```

Accessing RESTful services on the Client

- The `RestTemplate` is the core class for client-side access to RESTful services.
- It is conceptually similar to other template classes in Spring, such as `JdbcTemplate` and `JmsTemplate`
- `RestTemplate`'s behavior is customized by providing callback methods and configuring the `HttpMessageConverter` used to marshal and unmarshal objects.

RestTemplate methods

HTTP Method	RestTemplate Method
DELETE	<code>delete(String url, String... urlVariables)</code>
GET	<code>getForObject(String url, Class<T> responseType, String... urlVariables)</code>
HEAD	<code>headForHeaders(String url, String... urlVariables)</code>
OPTIONS	<code>optionsForAllow(String url, String... urlVariables)</code>
POST	<code>postForLocation(String url, Object request, String... urlVariables)</code>
PUT	<code>put(String url, Object request, String...urlVariables)</code>