**Spring**

IoC ilk başta , inversion of control yani kontrolün çevrilmesi adıyla ortaya

çıkmış olsa da , Martin Fowler bir eğitimci bu terimin anlamını irdelemiştir.”kontrolü

hangi açıdan çeviriyorlar?” diye bir soru yönelten Fowler daha sonra terimin

Dependency Injection olarak değiştirilmesini önermiştir ve bu terim daha sonra birçok çevre tarafından kabul edilmiştir.

IoC şu an birçok framework’de kullanılan ve frameworklerde yeni yeni

vurgulanan fakat aslında karakteristik olarak birçok framework’ün içinde çoğu kez

kullanılmış olan etkili bir tekniktir.

IoC : Inversion of Control

AOP : Aspect Oriented Programming

EJB başlangıç tarihi Mart 1998

(svn co <https://anonsvn.springframework.org/svn/spring-samples/petclinic/trunk/>)

pom.xml’in bulunduğu dizine gidip mvn eclipse:eclipse yazılsın. Başka seçenekler için maven sitesine bakılabilir.

Spring Recipies A Problem Solution (2010)

Spring is an open source framework, originally created by Rod Johnson and described in his book *Expert One-on-One: J2EE Design and Development*. Spring was created to address the complexity of enterprise application development, and makes it possible to use plain-vanilla JavaBeans to achieve things that were previously only possible with EJBs. But Spring’s usefulness isn’t limited to server-side development. Any Java application can benefit from Spring in terms of simplicity, testability, and loose coupling.

Problem

You have to instantiate the Spring IoC container for it to create bean instances by reading their configurations. Then, you can get the bean instances from the IoC container to use.

Solution

Spring provides two types of IoC container implementation. The basic one is called *bean factory*. The more advanced one is called *application context*, which is a compatible extension to the bean factory. Note that the bean configuration files for these two types of IoC containers are identical. The application context provides more advanced features than the bean factory while keeping the basic features compatible. So we strongly recommend using the application context for every application unless the resources of this application are restricted, such as when running in an applet or a

mobile device. The interfaces for the bean factory and the application context are **BeanFactory** and **ApplicationContext**, respectively. The interface **ApplicationContext** is a subinterface of **BeanFactory** for maintaining compatibility.

**Instantiating an Application Context**

**ApplicationContext** is an interface only. You have to instantiate an implementation of it. The **ClassPathXmlApplicationContext** implementation builds an application context by loading an XML configuration file from the classpath. You can also specify multiple configuration files for it.

**ApplicationContext context = new ClassPathXmlApplicationContext("beans.xml");**

Besides **ClassPathXmlApplicationContext**, several other **ApplicationContext** implementations are provided by Spring. **FileSystemXmlApplicationContext** is used to load XML configuration files from the file system or from URLs, while **XmlWebApplicationContext** and **XmlPortletApplicationContext** can be used in web and portal applications only.

Spring allows you to configure your beans in one or more bean configuration files. For a simple application, you can just centralize your beans in a single configuration file. But for a large application with a lot of beans, you should separate them in multiple configuration files according to their functionalities (e.g., controllers, DAO, and JMS). One useful division is by the architectural layer that a given context services.

**Declaring Beans in the Bean Configuration File**

Each bean should provide a unique **name** or **id** and a fully qualified class name for the Spring IoC container to instantiate it. For each bean property of simple type (e.g., **String** and other primitive types),

you can specify a **<value>** element for it. Spring will attempt to convert your value into the declaring type of this property. To configure a property via setter injection, you use the **<property>** element and specify the property name in its **name** attribute. A **<property>** requires that the bean contain a corresponding setter method.

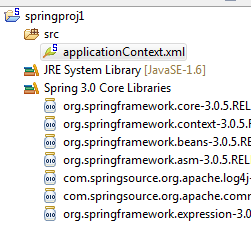
**How It Works**

Suppose that you are going to develop an application for generating sequence numbers. In this application, there may be many series of sequence numbers to generate for different purposes. Each one of them will have its own prefix, suffix, and initial value. So you have to create and maintain multiple generator instances in your application.

**Creating the Bean Class**

In accordance with the requirements, you create the **SequenceGenerator** class that has three properties **prefix**, **suffix**, and **initial**—that can be injected via setter methods or a constructor. The private field **counter** is for storing the current numeric value of this generator. Each time you call the **getSequence()** method on a generator instance, you will get the last sequence number with the prefix and suffix joined. You declare this method as **synchronized** to make it thread-safe.

Düz bir **java projesi** açalım springproj1.



**applicationcontext.xml’**inilk hali

|  |
| --- |
| <?xml version=*"1.0"* encoding=*"UTF-8"*?>  <beans  xmlns=*"http://www.springframework.org/schema/beans"*  xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*  xmlns:p=*"http://www.springframework.org/schema/p"*  xsi:schemaLocation=*"http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-3.0.xsd"*>  </beans> |

**applicationcontext.xml**

|  |
| --- |
| <?xml version=*"1.0"* encoding=*"UTF-8"*?>  <beans xmlns=*"http://www.springframework.org/schema/beans"*  xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"* xmlns:p=*"http://www.springframework.org/schema/p"*  xsi:schemaLocation=*"http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-3.0.xsd"*>  <bean name=*"sequenceGenerator2"* class=*"mypack.SequenceGenerator"*>  <constructor-arg>  <value>30</value>  </constructor-arg>  <constructor-arg>  <value>A</value>  </constructor-arg>  <constructor-arg>  <value>100000</value>  </constructor-arg>  </bean>  </beans> |

|  |
| --- |
| **package** mypack;  **public** **class** SequenceGenerator {  **private** String prefix;  **private** String suffix;  **private** **int** initial;  **private** **int** counter;  **public** SequenceGenerator() {  }  **public** SequenceGenerator(String prefix, String suffix, **int** initial) {  **this**.prefix = prefix;  **this**.suffix = suffix;  **this**.initial = initial;  }  **public** **void** setPrefix(String prefix) {  **this**.prefix = prefix;  }  **public** **void** setSuffix(String suffix) {  **this**.suffix = suffix;  }  **public** **void** setInitial(**int** initial) {  **this**.initial = initial;  }  **public** **synchronized** String getSequence() {  StringBuffer buffer = **new** StringBuffer();  buffer.append(prefix);  buffer.append(initial + counter++);  buffer.append(suffix);  **return** buffer.toString();  }  } |

Test.java

|  |
| --- |
| **import** mypack.SequenceGenerator;  **import** org.springframework.context.ApplicationContext;  **import** org.springframework.context.support.ClassPathXmlApplicationContext;  **public** **class** Test {  **public** **static** **void** main(String[] args) {  ApplicationContext context = **new** ClassPathXmlApplicationContext(  "applicationContext.xml");  SequenceGenerator generator = (SequenceGenerator) context  .getBean("sequenceGenerator1");  System.*out*.println(generator.getSequence());  System.*out*.println(generator.getSequence());  }  } |

30100000A

30100001A

**Declaring Beans in the Bean Configuration File**

Each bean should provide a unique **name** or **id** and a fully qualified class name for the Spring IoC container to instantiate it. For each bean property of simple type (e.g., **String** and other primitive types), you can specify a **<value>** element for it. Spring will attempt to convert your value into the declaring type of this property. To configure a property via setter injection, you use the **<property>** element and specify the property name in its **name** attribute. A **<property>** requires that the bean contain a corresponding

setter method.

|  |
| --- |
| <bean name=*"sequenceGenerator1"* class=*"mypack.SequenceGenerator"*>  <property name=*"prefix"*>  <value>30</value>  </property>  <property name=*"suffix"*>  <value>A</value>  </property>  <property name=*"initial"*>  <value>100000</value>  </property>  </bean> |

You can also configure bean properties via constructor injection by declaring them in the

**<constructor-arg>** elements. There’s not a **name** attribute in **<constructor-arg>**, because constructor arguments are position-based.

|  |
| --- |
| <bean name=*"sequenceGenerator2"* class=*"mypack.SequenceGenerator"*>  <constructor-arg>  <value>30</value>  </constructor-arg>  <constructor-arg>  <value>A</value>  </constructor-arg>  <constructor-arg>  <value>100000</value>  </constructor-arg>  </bean> |

In the Spring IoC container, each bean’s name should be unique, although duplicate names are allowed for overriding bean declaration if more than one context is loaded. A bean’s name can be defined by the **name** attribute of the **<bean>** element. Actually, there’s a preferred way of identifying a bean: through the standard XML **id** attribute, whose purpose is to identify an element within an XML document. In this way, if your text editor is XML-aware, it can help to validate each bean’s uniqueness at design time.

**<bean** id="sequenceGenerator"

**class="commypack.SequenceGenerator">**

**...**

**</bean>**

However, XML has restrictions on the characters that can appear in the XML id attribute. But usually, you won’t use those special characters in a bean name. Moreover, Spring allows you to specify multiple names, separated by commas, for a bean in the name attribute. But you can’t do so in the id attribute because commas are not allowed there.

In fact, neither the bean name nor the bean ID is required for a bean. A bean that has no name defined is called an anonymous bean. You will usually create beans like this that serve only to interact with the Spring container itself, that you are sure you will only inject by type later on, or that you will nest, inline, in the declaration of an outer bean.

**Defining Bean Properties by Shortcut**

Spring supports a shortcut for specifying the value of a simple type property. You can present a **value** attribute in the **<property>** element instead of enclosing a **<value>** element inside.

|  |
| --- |
| <bean id=*"sequenceGenerator3"* class=*"mypack.SequenceGenerator"*>  <property name=*"prefix"* value=*"30"* />  <property name=*"suffix"* value=*"A"* />  <property name=*"initial"* value=*"100000"* />  </bean> |

This shortcut also works for constructor arguments.

|  |
| --- |
| <bean name=*"sequenceGenerator4"* class=*"mypack.SequenceGenerator"*>  <constructor-arg value=*"30"* />  <constructor-arg value=*"A"* />  <constructor-arg value=*"100000"* />  </bean> |

Since Spring 2.0 another convenient shortcut to define properties was added. It consists of using the **p** schema to define bean properties as attributes of the **<bean>** element. This can shorten the lines of XML configuration.

|  |
| --- |
| <bean id=*"sequenceGenerator5"* class=*"mypack.SequenceGenerator"*  p:prefix=*"30"* p:suffix=*"A"* p:initial=*"100000"* /> |

**Configuring Collections for Your Beans**

**List**, **Set**, and **Map** are the core interfaces representing the three main types of collections in the Java SDK, part of a framework called the Java Collections framework. For each collection type, Java provides several implementations with different functions and characteristics from which you can choose. In Spring, these types of collections can be easily configured with a group of built-in XML tags, such as

<list>, <set>, and <map>.

Suppose you are going to allow more than one suffix for your sequence generator. The suffixes will be appended to the sequence numbers with hyphens as the separators. You may consider accepting suffixes of arbitrary data types and converting them into strings when appending to the sequence numbers.

**Lists, Arrays, and Sets**

First, let’s use a **java.util.List** collection to contain your suffixes. A list is an ordered and indexed collection whose elements can be accessed either by index or with a for-each loop.

|  |
| --- |
| **package** mypack;  **import** java.util.List;  **public** **class** SequenceGenerator1 {  **private** List<Object> suffixes;  **private** **int** initial;  **private** **int** counter;  **public** SequenceGenerator1() {  }  **public** SequenceGenerator1(List<Object> suffixes, **int** initial) {  **super**();  **this**.suffixes = suffixes;  **this**.initial = initial;    }  **public** **int** getInitial() {  **return** initial;  }  **public** **void** setInitial(**int** initial) {  **this**.initial = initial;  }  **public** List<Object> getSuffixes() {  **return** suffixes;  }  **public** **void** setSuffixes(List<Object> suffixes) {  **this**.suffixes = suffixes;  }  **public** **synchronized** String getSequence() {  StringBuffer buffer = **new** StringBuffer();  buffer.append(initial + counter++);  **for** (Object suffix : suffixes) {  buffer.append("-");  buffer.append(suffix);  }  **return** buffer.toString();  }  } |

To define a property of the interface java.util.List in the bean configuration, you specify a <list> tag that contains the elements. The elements allowed inside the <list> tag can be a simple constant value specified by <value>, a bean reference by <ref>, an inner bean definition by <bean>, an ID reference definition by <idref>, or a null element by <null>. You can even embed other collections in a collection.

|  |
| --- |
| <bean id=*"sequenceGenerator6"* class=*"mypack.****SequenceGenerator1****"*>  <property name=*"initial"* value=*"100000"* />  <property name=*"suffixes"*>  <list>  <value>A</value>  <bean class=*"java.net.URL"*>  <constructor-arg value=*"http"* />  <constructor-arg value=*"www.apress.com"* />  <constructor-arg value=*"/"* />  </bean>  <null />  </list>  </property>  </bean> |

|  |
| --- |
| **import** mypack.SequenceGenerator1;  **import** org.springframework.context.ApplicationContext;  **import** org.springframework.context.support.ClassPathXmlApplicationContext;  **public** **class** Test {  **public** **static** **void** main(String[] args) {  ApplicationContext context = **new** ClassPathXmlApplicationContext(  "applicationContext.xml");  SequenceGenerator1 generator = (SequenceGenerator1) context  .getBean("sequenceGenerator6");  System.*out*.println(generator.getSequence());  System.*out*.println(generator.getSequence());  }  } |

100000-A-http://www.apress.com/-null

100001-A-http://www.apress.com/-null

Conceptually, an array is very similar to a list in that it’s also an ordered and indexed collection that can be accessed by index. The main difference is that the length of an array is fixed and cannot be extended dynamically. In the Java Collections framework, an array and a list can be converted to each other through the **Arrays.asList()** and **List.toArray()** methods. For your sequence generator, you can use an **Object[]** array to contain the suffixes and access them either by index or with a for-each loop.

**mypack;**

**...**

**public class SequenceGenerator {**

**...**

**private Object[] suffixes;**

**public void setSuffixes(Object[] suffixes) {**

**this.suffixes = suffixes;**

**}**

**...**

**}**

The definition of an array in the bean configuration file is identical to a list denoted by the <list> tag. Another common collection type is a set. Both the java.util.List interface and the java.util.Set interface extend the same interface: java.util.Collection. A set differs from a list in that it is neither ordered nor indexed, and it can store unique objects only. That means no duplicate element can be contained in a set. When the same element is added to a set for the second time, it will replace the old one. The equality of elements is determined by the equals() method.

|  |
| --- |
| **package** mypack;  **import** java.util.List;  **import** java.util.Set;  **public** **class** SequenceGenerator2 {  **private** Set<Object> suffixes;  **private** **int** initial;  **private** **int** counter;  **public** SequenceGenerator2() {  }  **public** SequenceGenerator2(Set<Object> suffixes, **int** initial) {  **super**();  **this**.suffixes = suffixes;  **this**.initial = initial;    }  **public** **int** getInitial() {  **return** initial;  }  **public** **void** setInitial(**int** initial) {  **this**.initial = initial;  }  **public** Set<Object> getSuffixes() {  **return** suffixes;  }  **public** **void** setSuffixes(Set<Object> suffixes) {  **this**.suffixes = suffixes;  }  **public** **synchronized** String getSequence() {  StringBuffer buffer = **new** StringBuffer();  buffer.append(initial + counter++);  **for** (Object suffix : suffixes) {  buffer.append("-");  buffer.append(suffix);  }  **return** buffer.toString();  }  } |

To define a property of **java.util.Set** type, use the **<set>** tag to define the elements in the same way as a list.

|  |
| --- |
| <bean id=*"sequenceGenerator7"* class=*"mypack.SequenceGenerator2"*>  <property name=*"initial"* value=*"100000"* />  <property name=*"suffixes"*>  <set>  <value>A</value>  <bean class=*"java.net.URL"*>  <constructor-arg value=*"http"* />  <constructor-arg value=*"www.apress.com"* />  <constructor-arg value=*"/"* />  </bean>  <null />  </set>  </property>  </bean> |

Although there’s no order concept in the original set semantics, Spring preserves the order of your elements by using **java.util.LinkedHashSet**, an implementation of the **java.util.Set** interface that does preserve element order.

|  |
| --- |
| **import** mypack.SequenceGenerator2;  **import** org.springframework.context.ApplicationContext;  **import** org.springframework.context.support.ClassPathXmlApplicationContext;  **public** **class** Test {  **public** **static** **void** main(String[] args) {  ApplicationContext context = **new** ClassPathXmlApplicationContext(  "applicationContext.xml");  SequenceGenerator2 generator = (SequenceGenerator2) context  .getBean("sequenceGenerator7");  System.*out*.println(generator.getSequence());  System.*out*.println(generator.getSequence());  }  } |

100000-A-http://www.apress.com/-null

100001-A-http://www.apress.com/-null

**Maps and Properties**

A map interface is a table that stores its entries in key/value pairs. You can get a particular value from a map by its key, and also iterate the map entries with a for-each loop. Both the keys and values of a map can be of arbitrary type. Equality between keys is also determined by the equals() method. For example, you can modify your sequence generator to accept a java.util.Map collection that contains suffixes with keys.

**public class SequenceGenerator {**

**...**

**private Map<Object, Object> suffixes;**

**public void setSuffixes(Map<Object, Object> suffixes) {**

**this.suffixes = suffixes;**

**}**

**public synchronized String getSequence() {**

**StringBuffer buffer = new StringBuffer();**

**...**

for (Map.Entry entry : suffixes.entrySet()) {

buffer.append("-");

buffer.append(entry.getKey());

buffer.append("@");

buffer.append(entry.getValue());

}

**return buffer.toString();**

**}**

**}**

|  |
| --- |
| **package** mypack;  **import** java.util.Map;  **public** **class** SequenceGenerator3 {  **private** **int** initial;  **private** **int** counter;  **private** Map<Object, Object> suffixes;  **public** SequenceGenerator3() {  }  **public** SequenceGenerator3(Map<Object, Object> suffixes, **int** initial) {  **super**();  **this**.suffixes = suffixes;  **this**.initial = initial;  }  **public** **int** getInitial() {  **return** initial;  }  **public** **void** setInitial(**int** initial) {  **this**.initial = initial;  }  **public** Map<Object, Object> getSuffixes() {  **return** suffixes;  }  **public** **void** setSuffixes(Map<Object, Object> suffixes) {  **this**.suffixes = suffixes;  }  **public** **synchronized** String getSequence() {  StringBuffer buffer = **new** StringBuffer();  buffer.append(initial + counter++);  **for** (Map.Entry entry : suffixes.entrySet()) {  buffer.append("-");  buffer.append(entry.getKey());  buffer.append("@");  buffer.append(entry.getValue());  }  **return** buffer.toString();  }  } |

In Spring, a map is defined by the <map> tag, with multiple <entry> tags as children. Each entry contains a key and a value. The key must be defined inside the <key> tag. There is no restriction on the type of the key and value, so you are free to specify a <value>, <ref>, <bean>, <idref>, or <null> element for them. Spring will also preserve the order of the map entries by using **java.util.**LinkedHashMap. There are shortcuts to defining map keys and values as attributes of the <entry> tag. If they are simple constant values, you can define them by key and value. If they are bean references, you can

define them by key-ref and value-ref.

|  |
| --- |
| <bean id=*"sequenceGenerator8"* class=*"mypack.SequenceGenerator3"*>  <property name=*"initial"* value=*"100000"* />  <property name=*"suffixes"*>  <map>  <entry>  <key>  <value>type</value>  </key>  <value>A</value>  </entry>  <entry>  <key>  <value>url</value>  </key>  <bean class=*"java.net.URL"*>  <constructor-arg value=*"http"* />  <constructor-arg value=*"www.apress.com"* />  <constructor-arg value=*"/"* />  </bean>  </entry>  </map>  </property>  </bean> |

In all the collection classes seen thus far, you used values to set the properties. Sometimes, the desired goal is to configure a **null** value using a **Map** instance. Spring’s XML configuration schema includes explicit support for this. Here is a map with null values for the value of an entry:

**<property name="nulledMapValue">**

**<map>**

**<entry>**

**<key>** <value>null</value> **</key>**

**</entry>**

**</map>**

**</property>**

|  |
| --- |
| **import** mypack.SequenceGenerator3;  **import** org.springframework.context.ApplicationContext;  **import** org.springframework.context.support.ClassPathXmlApplicationContext;  **public** **class** Test {  **public** **static** **void** main(String[] args) {  ApplicationContext context = **new** ClassPathXmlApplicationContext(  "applicationContext.xml");  SequenceGenerator3 generator = (SequenceGenerator3) context  .getBean("sequenceGenerator8");  System.*out*.println(generator.getSequence());  System.*out*.println(generator.getSequence());  }  } |

A **java.util.Properties** collection is very similar to a map. It also implements the **java.util.Map** interface and stores entries in key/value pairs. The only difference is that the keys and values of a **Properties** collection are always strings.

|  |
| --- |
| **package** mypack;  **import** java.util.Map;  **import** java.util.Properties;  **public** **class** SequenceGenerator4 {  **private** **int** initial;  **private** **int** counter;  **private** Properties suffixes;  **public** SequenceGenerator4() {  }  **public** SequenceGenerator4(Properties suffixes, **int** initial) {  **super**();  **this**.suffixes = suffixes;  **this**.initial = initial;  }  **public** **int** getInitial() {  **return** initial;  }  **public** **void** setInitial(**int** initial) {  **this**.initial = initial;  }  **public** Properties getSuffixes() {  **return** suffixes;  }  **public** **void** setSuffixes(Properties suffixes) {  **this**.suffixes = suffixes;  }  **public** **synchronized** String getSequence() {  StringBuffer buffer = **new** StringBuffer();  buffer.append(initial + counter++);  **for** (Map.Entry entry : suffixes.entrySet()) {  buffer.append("-");  buffer.append(entry.getKey());  buffer.append("@");  buffer.append(entry.getValue());  }  **return** buffer.toString();  }  } |

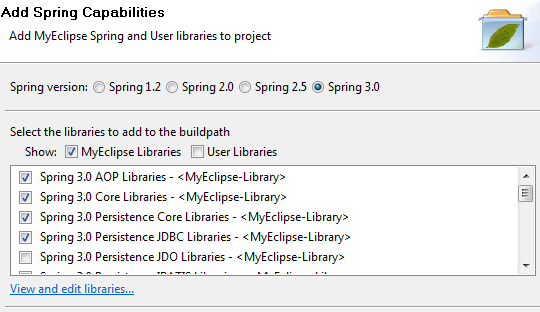
|  |
| --- |
| <bean id=*"sequenceGenerator9"* class=*"mypack.SequenceGenerator4"*>  <property name=*"initial"* value=*"100000"* />  <property name=*"suffixes"*>  <props>  <prop key=*"type"*>A</prop>  <prop key=*"url"*>http://www.apress.com/</prop>  <prop key=*"null"*>null</prop>  </props>  </property>  </bean> |

|  |
| --- |
| **import** mypack.SequenceGenerator4;  **import** org.springframework.context.ApplicationContext;  **import** org.springframework.context.support.ClassPathXmlApplicationContext;  **public** **class** Test {  **public** **static** **void** main(String[] args) {  ApplicationContext context = **new** ClassPathXmlApplicationContext(  "applicationContext.xml");  SequenceGenerator4 generator = (SequenceGenerator4) context  .getBean("sequenceGenerator9");  System.*out*.println(generator.getSequence());  System.*out*.println(generator.getSequence());  }  } |

100000-null@null-url@http://www.apress.com/-type@A

100001-null@null-url@http://www.apress.com/-type@A

**Spring ve JDBC**

****

JDBC Libraries işaretlenince birkaç kütüphane otomatik işaretleniyor (AOP ve Persistence Core libraries)

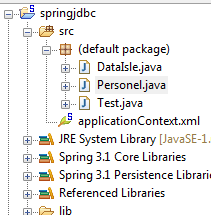
**applicationContext.xml**

|  |
| --- |
| <?xml version=*"1.0"* encoding=*"UTF-8"*?>  <beans xmlns=*"http://www.springframework.org/schema/beans"*  xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"* xmlns:p=*"http://www.springframework.org/schema/p"*  xsi:schemaLocation=*"http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-3.0.xsd"*>  <bean id=*"dataSource"*  class=*"org.springframework.jdbc.datasource.DriverManagerDataSource"*>  <property name=*"driverClassName"* value=*"sun.jdbc.odbc.JdbcOdbcDriver"* />  <property name=*"url"* value=*"jdbc:odbc:deneme"* />  <property name=*"username"* value=*""* />  <property name=*"password"* value=*""* />  </bean>  </beans> |

|  |
| --- |
| **public** **class** Person {  **private** String ad, soyad;  **public** String getAd() {  **return** ad;  }  **public** **void** setAd(String ad) {  **this**.ad = ad;  }  **public** String getSoyad() {  **return** soyad;  }  **public** **void** setSoyad(String soyad) {  **this**.soyad = soyad;  }  } |

|  |
| --- |
| import java.sql.ResultSet;  import java.sql.SQLException;  import java.util.List;  import javax.sql.DataSource;  import org.springframework.context.ApplicationContext;  import org.springframework.context.support.ClassPathXmlApplicationContext;  import org.springframework.jdbc.core.JdbcTemplate;  import org.springframework.jdbc.core.RowMapper;  public class Test {  public static void main(String[] args) {  ApplicationContext ac = new ClassPathXmlApplicationContext(  "applicationContext.xml");  // DataSource dataSource = (DataSource) ac.getBean("dataSource");  DataSource dataSource = (DataSource) ac.getBean("dataSource");  JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);  //int rowCount = jdbcTemplate.queryForInt("select count(0) from kimlik");  //System.out.println(rowCount);  List<Person> persons = jdbcTemplate.query(  "select ad, soyad from kimlik", new RowMapper() {  public Object mapRow(ResultSet rs, int rowNum)  throws SQLException {  Person person = new Person();  person.setAd(rs.getString("ad"));  person.setSoyad(rs.getString("soyad"));  return person;  }  });  for (Person person : persons) {  System.out.println(person.getAd());  }  }  } |

**Derby Örneği**

****

|  |
| --- |
| <?xml version=*"1.0"* encoding=*"UTF-8"*?>  <beans xmlns=*"http://www.springframework.org/schema/beans"*  xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"* xmlns:p=*"http://www.springframework.org/schema/p"*  xsi:schemaLocation=*"http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-3.1.xsd"*>  <bean id=*"dataSource"*  class=*"org.springframework.jdbc.datasource.DriverManagerDataSource"*>  <property name=*"driverClassName"* value=*"org.apache.derby.jdbc.ClientDriver"* />  <property name=*"url"* value=*"jdbc:derby://localhost:1527/myeclipse"* />  <property name=*"username"* value=*"classiccars"* />  <property name=*"password"* value=*"classiccars"* />  </bean>  </beans> |

|  |
| --- |
| **public** **class** Personel {  **private** String ad, soyad;    **public** Personel() {  // **TODO** Auto-generated constructor stub  }  **public** Personel(String ad, String soyad) {  **this**.ad = ad;  **this**.soyad = soyad;  }  **public** String getAd() {  **return** ad;  }  **public** **void** setAd(String ad) {  **this**.ad = ad;  }  **public** String getSoyad() {  **return** soyad;  }  **public** **void** setSoyad(String soyad) {  **this**.soyad = soyad;  }      } |

|  |
| --- |
| **import** java.sql.ResultSet;  **import** java.sql.SQLException;  **import** org.springframework.jdbc.core.RowMapper;  **public** **class** DataIsle **implements** RowMapper<Personel> {  **private** **static** **int** *i*;  **public** Personel mapRow(ResultSet rs, **int** sutunSay) **throws** SQLException {  System.*out*.println(++*i*);  Personel personel = **new** Personel(rs.getString(2), rs.getString(2));  **return** personel;  }  } |

|  |
| --- |
| **import** java.util.List;  **import** org.springframework.context.ApplicationContext;  **import** org.springframework.context.support.ClassPathXmlApplicationContext;  **import** org.springframework.jdbc.core.JdbcTemplate;  **import** org.springframework.jdbc.datasource.DriverManagerDataSource;  **public** **class** Test {  /\*\*  \* **@param** args  \*/  **public** **static** **void** main(String[] args) {  ApplicationContext context = **new** ClassPathXmlApplicationContext(  "applicationContext.xml");  DriverManagerDataSource source = (DriverManagerDataSource) context  .getBean("dataSource");  JdbcTemplate temp = **new** JdbcTemplate(source);  List<Personel> donen = temp.query("select \* from employee",  **new** DataIsle());  **for** (Personel personel : donen) {  System.*out*.println(personel.getAd());  }  }  } |

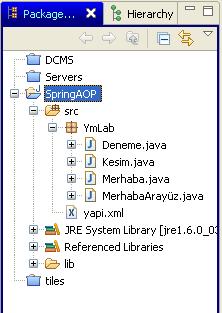
**Spring AOP ve AspectJ Desteği**

**Kesim Yönelimli Programlama Kavramı**

Uygulama geliştiriciler, geliştirdikleri sistemleri çeşitli sebeplerden dolayı bileşenler üzerine kurarlar. Çeşitli sebeplerden bazıları, uygulama kodunun karmaşık olmamasının istenmesi, sınama ve bakım yapmanın kolay olmasının istenmesi, daha da önemlisi bileşen tabanlı sistemlerin tasarlanması ve uygulamasının daha kolay, daha az riskli olması gibi sebeplerdir. Bu bileşenler üzerine dağıtılmış çeşitli kesimler (aspects) bulunur. Eğer sistem geliştiriciler, bu kesimleri nesneye yönelik teknikler kullanarak uygulamaya çalışırlara karmaşık geliştirmeden kaçalım derken sistemi iyice karmaşıklaştırırlar. Bunun için bağımlılıklardan kurtulup işlevsel tanımlar yapmamız gerekir. Kesim Yönelimli Progrmalama ile işlevsel tanımlamalar yapıp; kolay tasarlanıp uygulanabilen, sınaması ve bakımı kolay yapılabilen uygulamalar geliştiririz. Kesim Yönelimli Programlama, program yapısına farklı bir bakış açısı getirerek Nesneye Yönelik Programlamayı tamamlar. Nesneye Yönelik Programlama'da anahtar modülerlik birimi sınıf iken, Kesim Yönelimli Programlama'da modülerlik birimi kesimlerdir. Kesimler, çoklu türleri ve nesneleri kesen ilgilerin modülerize edilmesini etkinleştirir. Kesim Yönelimli Proglamlama'da potansiyel olarak birçok nesnenin kodunu etkileyen ilgilere crosscutting ilgiler deniliyor. Yazılım sistemleri çekirdek ve sistem düzeyi kesimlerinden oluşur. Örnek olarak bir doysa paylaşım sistemini düşünelim. Bu sistemdeki kesimler ve işlevleri şunlar olabilir;  
  
• Çekirdek Kesim  
• Dosyaların işlenmesi  
• Sistem Düzeyi Kesimleri   
• Erişim haklarının düzenlenmesi  
• Rollerin tanımlanması  
• Sistem kayıtlarının tutulması  
  
Şimdi Kesim Yönelimli Proglamlama ile ilgili bazı tanımlara bakalım.  
  
• Kesim (Aspect)  
Birçok sınıfı kesen bir ilginin modülerize edilmesi. İşlem yönetimi, J2EE uygulamalarındaki crosscutting ilgilere iyi bir örnektir. Kesim Yönelimli Programlama'da, kesimler ya düzenli sınıflar kullanılarak ya da düzenl sınıflar ile @AspectJ açıklamaları (annotationlarla) kullanılarak uygulanırlar.   
• Birleşim Noktası (Join Point)  
Uygulamanın işletildiği andaki bir noktadır. Bir fonksiyonun işletilmesi veya bir istisna durumun yakalanması örnek olarak verilebilir. Kesim Yönelimli Proglamlama'da bir nokta her zaman bir fonksiyonun işletilmesidir.   
• Tavsiye (Advice)  
Bir kesimin belirli bir noktada aldığı eylemdir. Tavsiyenin türleri vardır ve bunlardan aşağıda, Tavsiye Türleri adlı başlık altında bahsedilmiştir. Kesim Yönelimli Programlama, tavsiyeyi durdurucu-kesici (interceptor) olarak olarak modeller.   
• Nokta Kümleri (Pointcut)  
Birleşim noktalarını eşleyen bir belirtimdir. Tavsiye, nokta kümeleri ifadesiyle ilgilidir ve nokta kümesi tarafından eşlenmiş herhangi bir birleşim noktasında çalışır. Nokta kümelerinin eşlediği birleşme noktası Kesim Yönelimli Programlamanın merkezinde yer alır. Spring'de öntanımlı olarak AspectJ nokta kümesi ifadeleri kullanılır.   
• Takdim (Introduction)  
Bir tür lehine ek işlevler veya alanlar bildirilmesi. Spring AOP, herhangi bir bildirilmiş nesneye yeni arayüzler takdim etmemize izin verir. Mesala ön belleklemeyi basitleştirmede, isModified arayüzünün bean uygulaması için bir takdim kullanabiliriz.   
• Hedef Nesne (Target Object)  
Bir ya da daha fazla kesim tarafından bildirilen, tavsiye edilen nesne. Tavsiye edilen nesne (advised object) olarak da bilnir. Spring AOP, çalışma zamanlı vekiller kullanılarak uygulandığında bu nesne her zaman vekil olmuş nesne olacaktır.   
• AOP Vekili (AOP Proxy)  
Tavsiye fonksiyonlarının işletilmesi gibi kesim kontratlarının uygulanması için AOP çatısının oluşturduğu nesne. JDK dinamik vekili veya CGLIB vekili Spring uygulama çatısındaki bir AOP vekil örneği olabilir.   
• Dolama (Weaving)  
Tavsiye nesnesi oluşturmak için kesimlerin diğer nesnelerle veya uygulama türleriyle bağlanmasıdır. Bu derleme zamanında, yükleme zamanında ya da çalışma zamanında olabilir. Spring AOP'ta dolama çalışma zamanında olur.   
  
Tavsiye Türleri  
  
• Önceki Tavsiye (Before Advice)  
Bir birleşme noktasından önce çalışan tavsiyedir. İstisna durumu olmadığı sürece, işletim akışının birleşim noktasına gitmesini engelleme yeteneği yoktur.   
• Dönüşten Sonraki Tavsiye (After Returning Advice)  
Bir birleşim noktasının normal olarak tamamlanmasının dönüşünde işletilen tavsiye türüdür. Normal olarak demekten kastedilen istisna olmadan dönüş olabilir mesala.   
• Fırlatmadan Sonraki Tavsiye (After Throwing Advice)  
Bir fonksiyon istisna yakalayınca bunu fırlatır ve sonlanır. Bundan sonra çalışacak olan tavsiyedir.   
• Sonraki (Son) Tavsiye (After, Finally Advice)  
Birleşim noktası sonunda çıkış türüne bakmadan işletilen tavsiyedir. Çıkış türü normal ya da istisnadan kaynaklanan çıkış olabilir.   
• Çevre Tavsiyesi (Around Advice)  
Bir birleşim noktasını bir fonksiyonun çağrılması gibi sarmalayan tavsiyedir. En güçlü tavsiye türüdür. Fonksiyonun çağrılmasından önce ve sonra özel davranışlar yerine getirir.

### Örnek Uygulama

Bu kesimde yukarda anlatılanlar çerçevesine hazırlanmış bir merhaba dünya örneği verilecektir. Örnekteki çalıştırılabilir, küütük Deneme.java'dır. Bunun içinde Merhaba bean alınıyor ve merhabaDe fonksiyonu çağrılıyor. (Burada proje dizininde bulunan yapi.xml kütüğü kullanılıyor ve merhaba iletisi xml kütüğünden alınıyor.). Yapı kütüğündeki bilgilere göre, merhaba iletisi gösterilmeden önce, önceki tavsiye çalışıyor ve ilgili ileti yaılıyor. Daha sonra, merhaba iletisi yazılıyor ve sonra da sonraki tavsiye ile ilgili olan ileti yazılıyor. Aşağıda uygulamanın kodları ve kütüklerin yapısını gösteren bir resim var. Burda lib dizini altına spring ve aop ile ilgili kütüphaneler konulmuştur.



package YmLab;

public interface MerhabaArayüz {

public void merhabaDe();

}

Yukardaki kodda Merhaba uygulamalarımız için bir arayüz tanımı yapılmıştır. Bu arayüzü uygulayan uygulamalarımız merhabaDe() işlevini uygularlar.

package YmLab;

public class Merhaba implements MerhabaArayüz {

private String merhabailetisi;

public String getIleti() {

return merhabailetisi;

}

public void setIleti(String merhabailetisi) {

this.merhabailetisi = merhabailetisi;

}

public void merhabaDe() {

System.out.println(merhabailetisi);

}

}

Merhaba adlı bir uygulamanın kodunu yukarda görüyüruz. Bu uygulamızı MerhabaArayüzden uyguluyoruz. İçinde, basit bir string türünde değişken var ve bu türe ait getter setterler bulunuyor. Arayüzden uygulanan işlevin içinde sistem standard çıktısına gelen ileti yazdırılıyor.

package YmLab;

import org.aspectj.lang.ProceedingJoinPoint;

public class Kesim {

public Object cevreTavsiyeIslevi(ProceedingJoinPoint pjp) throws Throwable {

System.out.println("Cevreden Tavsiyesi Islevinin Oncesi");

Object obj = pjp.proceed();

System.out.println("Cevreden Tavsiyesi Islevinin Sonrasi");

return obj;

}

public void firlatmadanSonrakiTavsiyeIslev() {

System.out.println("Firlatmadan Sonraki Tavsiye Islevi");

}

public void oncekiTavsiyeIslevi() {

System.out.println("Onceki Tavsiye Islevi");

}

public void sonrakiTavsiyeIslevi() {

System.out.println("Sonraki Tavsiye Islevi");

}

}

Yukardaki kodda bir kesim sınıfı tanımlıyoruz. Bu kesimde, dört tane tavsiye işlemi bulunuyor ve bu tavsiyelerin içinde hangi tavsiye türü oldukları sistemin standard çıktısına yazdırılıyor.

package YmLab;

import org.springframework.beans.factory.\*;

import org.springframework.beans.factory.xml.XmlBeanFactory;

import org.springframework.core.io.FileSystemResource;

public class Deneme {

public static void main(String[] args) {

BeanFactory factory = new XmlBeanFactory(new FileSystemResource(

"C:\\work\\eclipse\\SpringAOP\\src\\yapi.xml"));

Merhaba meraba = (Merhaba) factory.getBean("Merhaba");

meraba.merhabaDe();

}

}

Yukardaki kodumuz bir main işlevi içeriyor ve örnek uygulamamızı denemek için kullanılıyor. İlk önce projemizde tanımladığımız yapı kütüğünden kesimlerle ve tavsiyelerle ilgili bilgileri alıyoruz. Daha sonra Merhaba adlı uygulama sınfımızı örnekleyip uyguladığımız merhabaDe() yordamını çağırıyoruz. Yapı kütüğündeki bilgilere göre, merhaba iletisi gösterilmeden önce, önceki tavsiye çalışıyor ve ilgili ileti yaılıyor. Daha sonra, merhaba iletisi yazılıyor ve sonra da sonraki tavsiye ile ilgili olan ileti yazılıyor.

<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"

"http://www.springframework.org/dtd/spring-beans.dtd">

<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.xsd

http://www.springframework.org/schema/aop

http://www.springframework.org/schema/aop/spring-aop.xsd">

<bean id="Merhaba" class="YmLab.Merhaba">

<property name="merhabailetisi"><value>merhabaaa</value></property>

</bean>

<bean id="kesimNo" class="YmLab.Kesim" />

<aop:config>

<aop:aspect ref="kesimNo">

<aop:pointcut id="noktaKumseiNo" expression="execution(\* merhabaDe(..))" />

<aop:before method="oncekiTavsiyeIslevi" pointcut="execution(\* merhabaDe(..))" />

<aop:after-returning method="sonrakiTavsiyeIslevi" pointcut-ref="noktaKumseiNo" />

<aop:after-throwing method="firlatmadanSonrakiTavsiyeIslev" pointcut-ref="noktaKumseiNo"/>

</aop:aspect>

</aop:config>

</beans>

Yapılandırma kütüğümüzün yapısı yukarda görülüyor. Uygulamamızı sınamak için Merhaba isimli bir bean ile onun kullanacağı ileti parametresi tanımlanmıştır. Daha sonra bir kesim tanımı yapılıyor. Kesimimiz için üç tavsiye tanımı yapıyoruz ve bu tavsiyelerimiz türüne göre yordamdan önce, sonra ya da istisna durumunda çalışacaktır.

Let us begin by defining some central AOP concepts and terminology. These terms are not Spring-specific... unfortunately, AOP terminology is not particularly intuitive; however, it would be even more confusing if Spring used its own terminology.

• *Aspect*: a modularization of a concern that cuts across multiple classes. Transaction management is a good example of a crosscutting concern in enterprise Java applications. In Spring AOP, aspects are implemented using regular classes (the schema-based approach) or regular classes annotated with the @Aspect annotation (the @AspectJ style).

• *Join point*: a point during the execution of a program, such as the execution of a method or the handling of an exception. In Spring AOP, a join point *always* represents a method execution.

• *Advice*: action taken by an aspect at a particular join point. Different types of advice include "around," "before" and "after" advice. (Advice types are discussed below.) Many AOP frameworks, including Spring, model an advice as an *interceptor*, maintaining a chain of interceptors *around* the join point.

• *Pointcut*: a predicate that matches join points. Advice is associated with a pointcut expression and runs at any join point matched by the pointcut (for example, the execution of a method with a certain name). The concept of join points as matched by pointcut expressions is central to AOP, and Spring uses the AspectJ pointcut expression language by default.

• *Introduction*: declaring additional methods or fields on behalf of a type. Spring AOP allows you to introduce new interfaces (and a corresponding implementation) to any advised object. For example, you could use an introduction to make a bean implement an IsModified interface, to simplify caching. (An introduction is known as an inter-type declaration in the AspectJ community.)

• *Target object*: object being advised by one or more aspects. Also referred to as the *advised* object. Since Spring AOP is implemented using runtime proxies, this object will always be a *proxied* object.

• *AOP proxy*: an object created by the AOP framework in order to implement the aspect contracts (advise method executions and so on). In the Spring Framework, an AOP proxy will be a JDK dynamic proxy or a CGLIB proxy.

• *Weaving*: linking aspects with other application types or objects to create an advised object. This can be done at compile time (using the AspectJ compiler, for example), load time, or at runtime. Spring AOP, like other pure Java AOP frameworks, performs weaving at runtime.

Types of advice:

• *Before advice*: Advice that executes before a join point, but which does not have the ability to prevent execution flow proceeding to the join point (unless it throws an exception).

• *After returning advice*: Advice to be executed after a join point completes normally: for example, if a method returns without throwing an exception.

• *After throwing advice*: Advice to be executed if a method exits by throwing an exception.

• *After (finally) advice*: Advice to be executed regardless of the means by which a join point exits (normal or exceptional return).

• *Around advice*: Advice that surrounds a join point such as a method invocation. This is the most powerful kind of advice. Around advice can perform custom behavior before and after the method invocation. It is also responsible for choosing whether to proceed to the join point or to shortcut the advised method execution by returning its own return value or throwing an exception. Around advice is the most general kind of advice. Since Spring AOP, like AspectJ, provides a full range of advice types, we recommend that you use the least powerful advice type that can implement the required behavior. For example, if you need only to update a cache with the return value of a method, you

are better off implementing an after returning advice than an around advice, although an around advice can accomplish the same thing. Using the most specific advice type provides a simpler programming model with less potential for errors. For example, you do not need to invoke the proceed() method on the JoinPoint used for around advice, and hence cannot fail to invoke it. In Spring 2.0, all advice parameters are statically typed, so that you work with advice parameters of the appropriate type (the type of the return value from a method execution for example) rather than Object arrays.

The concept of join points, matched by pointcuts, is the key to AOP which distinguishes it from older technologies offering only interception. Pointcuts enable advice to be targeted independently of the Object-Oriented hierarchy. For example, an around advice providing declarative transaction management can be applied to a set of methods spanning multiple objects (such as all business operations in the service layer).

**Spring AOP capabilities and goals**

Spring AOP is implemented in pure Java. There is no need for a special compilation process. Spring AOP does not need to control the class loader hierarchy, and is thus suitable for use in a Servlet container or application server. Spring AOP currently supports only method execution join points (advising the execution of methods on

Spring beans). Field interception is not implemented, although support for field interception could be added without breaking the core Spring AOP APIs. If you need to advise field access and update join points, consider a language such as AspectJ.

Spring AOP's approach to AOP differs from that of most other AOP frameworks. The aim is not to provide the most complete AOP implementation (although Spring AOP is quite capable); it is rather to provide a close integration between AOP implementation and Spring IoC to help solve common problems in enterprise applications. Thus, for example, the Spring Framework's AOP functionality is normally used in conjunction with the Spring IoC container. Aspects are configured using normal bean definition syntax (although this allows powerful "autoproxying" capabilities): this is a crucial difference from other AOP implementations. There are some things you cannot do easily or efficiently with Spring AOP, such as advise very fine-grained objects (such as domain objects typically): AspectJ is the best choice in such cases. However, our

experience is that Spring AOP provides an excellent solution to most problems in enterprise Java applications that are amenable to AOP. Spring AOP will never strive to compete with AspectJ to provide a comprehensive AOP solution. We believe that both proxy-based frameworks like Spring AOP and full-blown frameworks such as AspectJ

are valuable, and that they are complementary, rather than in competition. Spring 2.0 seamlessly integrates Spring AOP and IoC with AspectJ, to enable all uses of AOP to be catered for within a consistent Spring-based application architecture. This integration does not affect the Spring AOP API or the AOP Alliance API: Spring AOP remains backward-compatible.

**Note**

One of the central tenets of the Spring Framework is that of *non-invasiveness*; this is the idea that you should not be forced to introduce framework-specific classes and interfaces into your business/domain model. However, in some places the Spring Framework does give you the option to introduce Spring Framework-specific dependencies into your codebase: the rationale in giving you such options is because in certain scenarios it might be just plain easier to read or code some specific piece of functionality in such a way. The Spring Framework (almost) always offers you the choice though: you have the freedom to make an informed decision as to which option best suits your particular use case or scenario. You have the choice of AspectJ and/or Spring AOP, and you also have

the choice of either the @AspectJ annotation-style approach or the Spring XML

configuration-style approach.

|  |
| --- |
| **package** org.csystem.calculator;  **public** **interface** ArithmeticCalculator {  **public** **double** add(**double** a, **double** b);  **public** **double** sub(**double** a, **double** b);  **public** **double** mul(**double** a, **double** b);  **public** **double** div(**double** a, **double** b);  } |

|  |
| --- |
| **package** org.csystem.calculator;  **public** **interface** UnitCalculator {  **public** **double** kilogramToPound(**double** kilogram);  **public** **double** kilometerToMile(**double** kilometer);  } |

|  |
| --- |
| **package** org.csystem.calculator;  **public** **class** ArithmeticCalculatorImpl **implements** ArithmeticCalculator {  @Override  **public** **double** add(**double** a, **double** b) {  **double** result = a + b;  System.*out*.println(a + " + " + b + " = " + result);  **return** result;  }  @Override  **public** **double** sub(**double** a, **double** b) {  **double** result = a - b;  System.*out*.println(a + " - " + b + " = " + result);  **return** result;  }  @Override  **public** **double** mul(**double** a, **double** b) {  **double** result = a \* b;  System.*out*.println(a + " \* " + b + " = " + result);  **return** result;  }  @Override  **public** **double** div(**double** a, **double** b) {  **if** (b == 0) {  **throw** **new** IllegalArgumentException("Division by zero");  }  **double** result = a / b;  System.*out*.println(a + " / " + b + " = " + result);  **return** result;  }  } |

|  |
| --- |
| **package** org.csystem.calculator;  **public** **class** UnitCalculatorImpl **implements** UnitCalculator {  @Override  **public** **double** kilogramToPound(**double** kilogram) {  **double** pound = kilogram \* 2.2;  System.*out*.println(kilogram + " kilogram = " + pound + " pound");  **return** pound;  }  @Override  **public** **double** kilometerToMile(**double** kilometer) {  **double** mile = kilometer \* 0.62;  System.*out*.println(kilometer + " kilometer = " + mile + " mile");  **return** mile;  }  } |

|  |
| --- |
| package org.csystem.calculator;  import org.apache.commons.logging.Log;  import org.apache.commons.logging.LogFactory;  import org.aspectj.lang.annotation.Before;  @Aspect  public class CalculatorLoggingAspect {  //private Log log = LogFactory.getLog(this.getClass());  @Before("execution(\* ArithmeticCalculator.add(..))")  public void logBefore() {  //log.info("The method add() begins");  System.out.println("toplamdan once cagrildi");  }  } |

|  |
| --- |
| <?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*  *http://www.springframework.org/schema/aop*  *http://www.springframework.org/schema/aop/spring-aop-3.0.xsd"*>  <aop:aspectj-autoproxy />  <bean id=*"arithmeticCalculator"* class=*"org.csystem.calculator.ArithmeticCalculatorImpl"* />  <bean id=*"unitCalculator"* class=*"org.csystem.calculator.UnitCalculatorImpl"* />  <bean class=*"org.csystem.calculator.CalculatorLoggingAspect"* />  </beans> |

|  |
| --- |
| **package** org.csystem.calculator;  **import** org.springframework.context.ApplicationContext;  **import** org.springframework.context.support.ClassPathXmlApplicationContext;  **public** **class** Main {  **public** **static** **void** main(String[] args) {  ApplicationContext context = **new** ClassPathXmlApplicationContext(  "applicationContext.xml");  ArithmeticCalculator arithmeticCalculator = (ArithmeticCalculator) context  .getBean("arithmeticCalculator");  arithmeticCalculator.add(1, 2);  arithmeticCalculator.sub(4, 3);  arithmeticCalculator.mul(2, 3);  arithmeticCalculator.div(4, 2);  UnitCalculator unitCalculator = (UnitCalculator) context  .getBean("unitCalculator");  unitCalculator.kilogramToPound(10);  unitCalculator.kilometerToMile(5);  }  } |

**Ekran görüntüsü**

log4j:WARN No appenders could be found for logger (org.springframework.context.support.ClassPathXmlApplicationContext).

log4j:WARN Please initialize the log4j system properly.

1.0 + 2.0 = 3.0

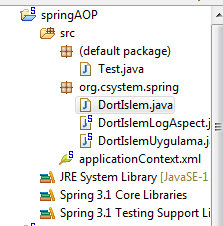
4.0 - 3.0 = 1.0

2.0 \* 3.0 = 6.0

4.0 / 2.0 = 2.0

10.0 kilogram = 22.0 pound

5.0 kilometer = 3.1 mile



|  |
| --- |
| <?xml version=*"1.0"* encoding=*"UTF-8"*?>  <beans xmlns=*"http://www.springframework.org/schema/beans"*  xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"* xmlns:p=*"http://www.springframework.org/schema/p"*  xmlns:aop=*"http://www.springframework.org/schema/aop"*  xsi:schemaLocation=*"http://www.springframework.org/schema/beans*  *http://www.springframework.org/schema/beans/spring-beans-3.1.xsd*  *http://www.springframework.org/schema/aop*  *http://www.springframework.org/schema/aop/spring-aop-3.0.xsd"*>  <aop:aspectj-autoproxy />  <bean name=*"dortislem"* class=*"org.csystem.spring.DortIslemUygulama"*/>    <bean class=*"org.csystem.spring.DortIslemLogAspect"*/>  </beans> |

|  |
| --- |
| **package** org.csystem.spring;  **public** **interface** DortIslem {  **public** **double** topla(**double** a, **double** b);  **public** **double** cikarma(**double** a, **double** b);  **public** **double** carpma(**double** a, **double** b);  **public** **double** bolme(**double** a, **double** b);  } |

|  |
| --- |
| **package** org.csystem.spring;  **public** **class** DortIslemUygulama **implements** DortIslem {  @Override  **public** **double** topla(**double** a, **double** b) {  System.*out*.println("topla");  **return** a+b;  }  @Override  **public** **double** cikarma(**double** a, **double** b) {  System.*out*.println("cikar");  **return** a-b;  }  @Override  **public** **double** carpma(**double** a, **double** b) {  System.*out*.println("carp");  **return** a\*b;  }  @Override  **public** **double** bolme(**double** a, **double** b) {  System.*out*.println("bol");  **return** a/b;  }  } |

|  |
| --- |
| **package** org.csystem.spring;  **import** org.aspectj.lang.annotation.After;  **import** org.aspectj.lang.annotation.Aspect;  **import** org.aspectj.lang.annotation.Before;  @Aspect  **public** **class** DortIslemLogAspect {    @Before("execution(\* DortIslem.topla(..))")  **public** **void** topladanOnce(){  System.*out*.println("topla'dan once cagrildi");  }  @After("execution(\* DortIslem.bolme(..))")  **public** **void** bolmedenSonra(){  System.*out*.println("bolmeden sonra cagrildi..");  }  } |

|  |
| --- |
| **import** org.csystem.spring.DortIslem;  **import** org.springframework.context.ApplicationContext;  **import** org.springframework.context.support.ClassPathXmlApplicationContext;  **public** **class** Test {  /\*\*  \* **@param** args  \*/  **public** **static** **void** main(String[] args) {  ApplicationContext context = **new** ClassPathXmlApplicationContext(  "applicationContext.xml");  DortIslem d = (DortIslem) context.getBean("dortislem");  d.bolme(1, 2);  d.topla(3, 2);  }  } |

log4j:WARN No appenders could be found for logger (org.springframework.core.env.StandardEnvironment).

log4j:WARN Please initialize the log4j system properly.

log4j:WARN See http://logging.apache.org/log4j/1.2/faq.html#noconfig for more info.

bol

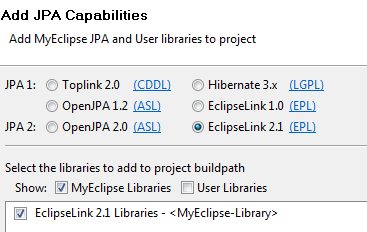
bolmeden sonra cagrildi..

topla'dan once cagrildi

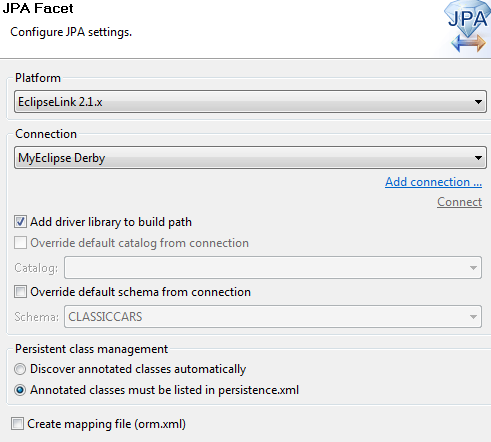
topla

**JPA Spring Projesi**

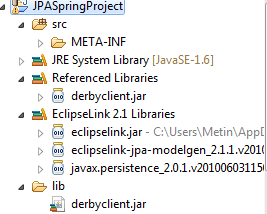
MyEclipse 9.0 kullanılacaktır. Bir java projesi oluşturalım.: **JPASpringProject.** Projenin düğümü üzerinde iken *MyEclipse/Project Capabilities/Add JPA Capabilies* seçelim. Herşeyi default bırakalım.

****

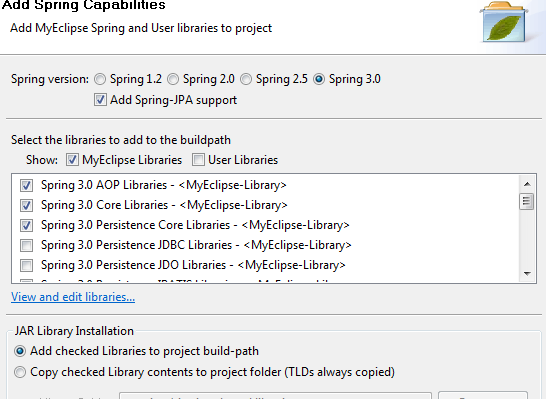
Next düğmesine bastıktan sonra gelen diyalog penceresini şekildeki gibi dolduralım.

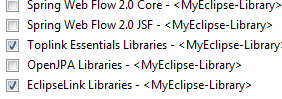


Finish düğmesine basalım. Proje aşağıdaki gibi görülecektir.



Şimdide Spring capabiliy ekleyelim. Gerekli kütüphaneleri de ekleyelim.

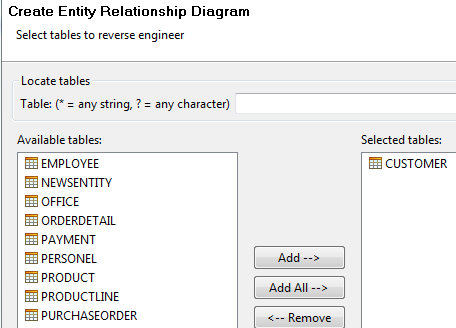




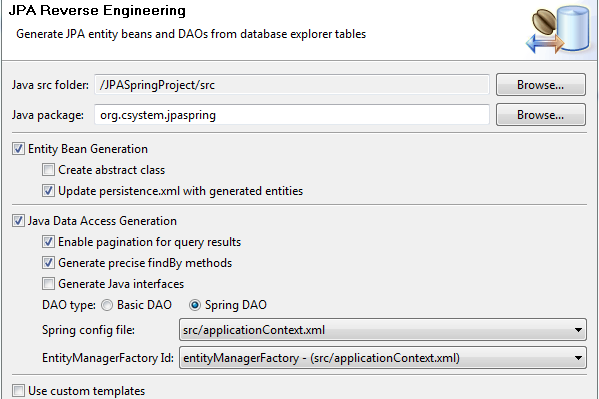
Next/next/finish diyelim. Src dizininde org.csystem.jpaspring paketi oluşturalım. Paket üzerinde iken sağ tuşla myeclipse/Generate Entities & DAO seçelim. Gelen diyalog penceresinde Customer tablosunu seçelim.

**applicationContext.xml**

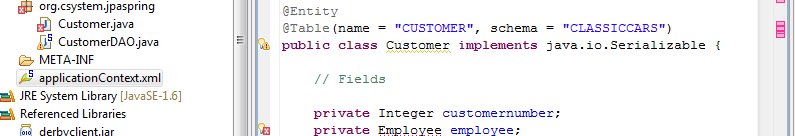
|  |
| --- |
| <?xml version=*"1.0"* encoding=*"UTF-8"*?>  <beans  xmlns=*"http://www.springframework.org/schema/beans"*  xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*  xmlns:p=*"http://www.springframework.org/schema/p"*  xsi:schemaLocation=*"http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-3.0.xsd http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx.xsd"* xmlns:tx=*"http://www.springframework.org/schema/tx"*>  <bean id=*"entityManagerFactory"*  class=*"org.springframework.orm.jpa.LocalEntityManagerFactoryBean"*>  <property name=*"persistenceUnitName"* value=*"JPASpringProject"* />  </bean>  <bean id=*"transactionManager"*  class=*"org.springframework.orm.jpa.JpaTransactionManager"*>  <property name=*"entityManagerFactory"*  ref=*"entityManagerFactory"* />  </bean>  <tx:annotation-driven transaction-manager=*"transactionManager"* /><bean  id=*"CustomerDAO"* class=*"org.csystem.jpaspring.CustomerDAO"*>  <property name=*"entityManagerFactory"*  ref=*"entityManagerFactory"* />  </bean></beans> |



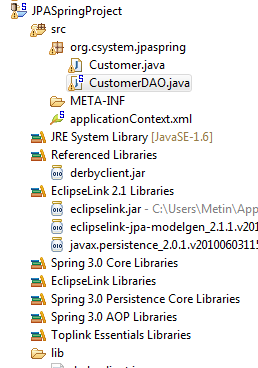
Bir sonraki pencerede seçenekleri şekildeki gibi ayarlayalım.



Customer ile ilintili diğer tablo elemanları silinmeli.



Projenin son durumu:



Bir test sınıf yazıp deneyelim.

|  |
| --- |
| package org.csystem.jpaspring;  import org.springframework.context.ApplicationContext;  import org.springframework.context.support.ClassPathXmlApplicationContext;  public class Test {  /\*\*  \* @param args  \*/  public static void main(String[] args) {    ApplicationContext ac = new ClassPathXmlApplicationContext(  "applicationContext.xml");  CustomerDAO dao = (CustomerDAO) ac.getBean("CustomerDAO");  System.out.println(dao);  Customer cus = dao.findById(103);  System.out.println(cus.getContactfirstname());  }  } |

Ekran çıktısı:

log4j:WARN No appenders could be found for logger (org.springframework.context.support.ClassPathXmlApplicationContext).

log4j:WARN Please initialize the log4j system properly.

org.csystem.jpaspring.CustomerDAO@2d95b3

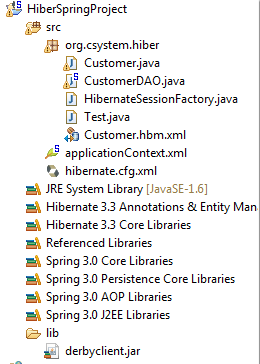
[EL Info]: 2011-08-29 12:49:51.243--ServerSession(23623672)--EclipseLink, version: Eclipse Persistence Services - 2.1.1.v20100817-r8050

[EL Info]: 2011-08-29 12:49:51.454--ServerSession(23623672)--file:/C:/java/IleriJava/workspace\_My9\_spring/JPASpringProject/bin/\_JPASpringProject login successful

Carine

**Hibernate Spring Projesi**

Yukarıdaki örneğin aynısı Hibernate için tekrarlanacak.



**applicationContext.xml**

|  |
| --- |
| <?xml version=*"1.0"* encoding=*"UTF-8"*?>  <beans  xmlns=*"http://www.springframework.org/schema/beans"*  xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*  xmlns:p=*"http://www.springframework.org/schema/p"*  xsi:schemaLocation=*"http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-3.0.xsd"*>  <bean id=*"sessionFactory"*  class=*"org.springframework.orm.hibernate3.LocalSessionFactoryBean"*>  <property name=*"configLocation"*  value=*"file:src/hibernate.cfg.xml"*>  </property>  </bean>  <bean id=*"CustomerDAO"* class=*"org.csystem.hiber.CustomerDAO"*>  <property name=*"sessionFactory"*>  <ref bean=*"sessionFactory"* />  </property>  </bean></beans> |

|  |
| --- |
| package org.csystem.hiber;  import org.springframework.context.ApplicationContext;  import org.springframework.context.support.ClassPathXmlApplicationContext;  public class Test {    public static void main(String[] args) {  ApplicationContext ac = new ClassPathXmlApplicationContext(  "applicationContext.xml");  CustomerDAO dao = (CustomerDAO) ac.getBean("CustomerDAO");  System.out.println(dao);  Customer cus = dao.findById(103);  System.out.println(cus.getContactfirstname());  }  } |

**Spring ile Web**

Most implementations of the MVC pattern in web applications introduce an additional

component: the *front controller* (Fowler 2003). The front controller is technical in nature

and coordinates request processing by enforcing a well defined request processing life

cycle: it maps a request onto a particular controller and renders the view selected by

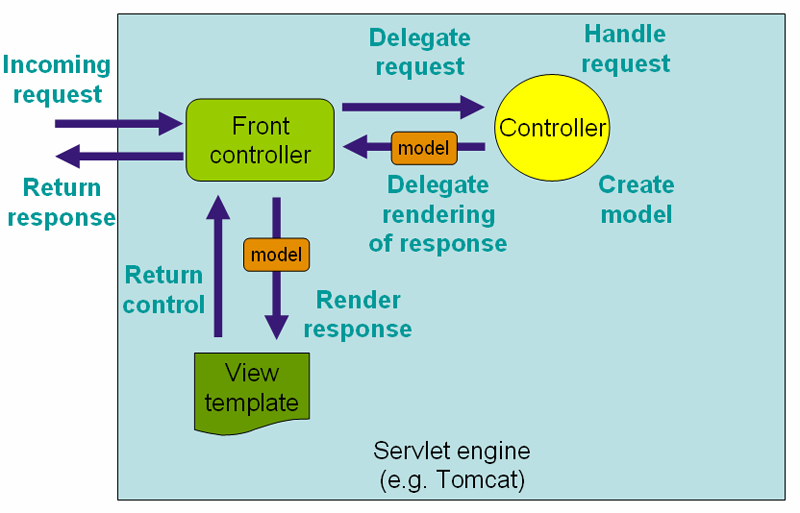
that controller. The front controller can also manage common concerns like security

and internationalization. Well known front controller implementations are the Struts

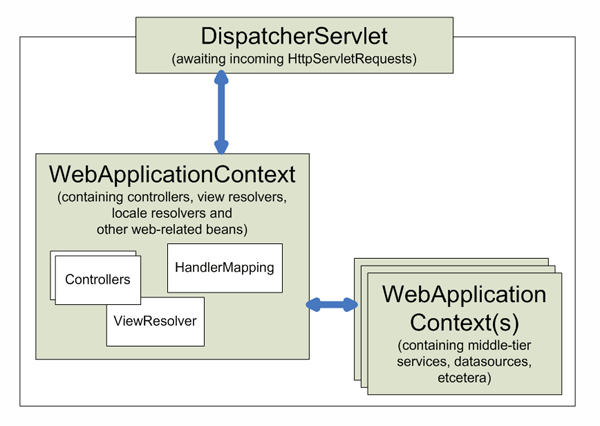
**ActionServlet,** the Spring **MVC Dispatcher Servlet**, or the JSF **FacesServlet**. From the

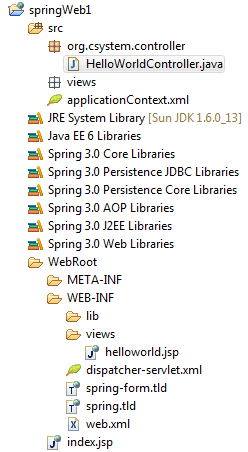
application’s point of view, the controller is in the driver’s seat. Once the front controller

has selected the appropriate controller for a request, that controller decides how to interface with business components, which view to display, and the model data to render.



**Figure: Context hierarchy in Spring Web MVC**





**HelloWorldController.java**

|  |
| --- |
| **package** org.csystem.controller;  **import** org.springframework.web.servlet.ModelAndView;  **import** org.springframework.stereotype.Controller;  **import** org.springframework.web.bind.annotation.RequestMapping;  @Controller  **public** **class** HelloWorldController {  @RequestMapping("/helloworld")  **public** ModelAndView helloWord() {  String message = "deneme";  **return** **new** ModelAndView("helloworld", "message", message);  }  } |

|  |
| --- |
| <?xml version=*"1.0"* encoding=*"UTF-8"*?>  <web-app version=*"3.0"* xmlns=*"http://java.sun.com/xml/ns/javaee"*  xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*  xsi:schemaLocation=*"http://java.sun.com/xml/ns/javaee*  *http://java.sun.com/xml/ns/javaee/web-app\_3\_0.xsd"*>  <display-name></display-name>  <servlet>  <servlet-name>dispatcher</servlet-name>  <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>  <load-on-startup>1</load-on-startup>  </servlet>  <servlet-mapping>  <servlet-name>dispatcher</servlet-name>  <url-pattern>/forms/\*</url-pattern>  </servlet-mapping>  <welcome-file-list>  <welcome-file>index.jsp</welcome-file>  </welcome-file-list>  </web-app> |

dispatcher-servlet.xml

|  |
| --- |
| <?xml version=*"1.0"* encoding=*"UTF-8"*?>  <beans xmlns=*"http://www.springframework.org/schema/beans"*  xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"* xmlns:p=*"http://www.springframework.org/schema/p"*  xmlns: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:component-scan base-package=*"org.csystem.controller"* />  <bean id=*"viewResolver"*  class=*"org.springframework.web.servlet.view.InternalResourceViewResolver"*>  <property name=*"prefix"*>  <value>/WEB-INF/views/</value>  </property>  <property name=*"suffix"*>  <value>.jsp</value>  </property>  </bean>  </beans> |

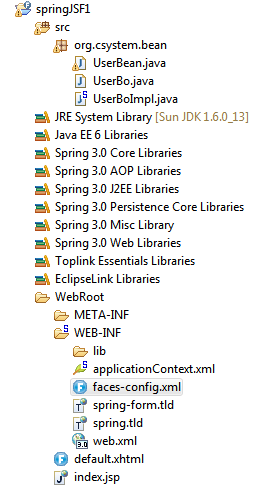
helloworld.jsp

|  |
| --- |
| <body>  ${message}  </body> |

index.jsp

|  |
| --- |
| <body>  <h1>Spring 3, MVC Examples</h1>  <ul>  <li>  <a href=*"forms/helloworld.html"*>Hello World</a>  </li>  </ul>  </body> |

**Spring 3 and JSF 2.0 Integration**



|  |
| --- |
| package org.csystem.bean;  import javax.inject.Inject;  import javax.inject.Named;  import org.springframework.context.annotation.Scope;  @Named  @Scope("session")  public class UserBean {  private String printMsgFromSpring;  @Inject  public UserBo userBo;  public void setUserBo(UserBo userBo) {  this.userBo = userBo;  }  public String getPrintMsgFromSpring() {  return userBo.getMessage();  //return "deneme";  }  } |

|  |
| --- |
| **package** org.csystem.bean;  **public** **interface** UserBo{    **public** String getMessage();    } |

|  |
| --- |
| **package** org.csystem.bean;  **import** javax.inject.Named;  @Named  **public** **class** UserBoImpl **implements** UserBo{    **public** String getMessage() {    **return** "JSF 2 + Spring Integration";    }    } |

applicationContext.xml

|  |
| --- |
| <?xml version=*"1.0"* encoding=*"UTF-8"*?>  <beans  xmlns=*"http://www.springframework.org/schema/beans"*  xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*  xmlns:p=*"http://www.springframework.org/schema/p"*  xsi:schemaLocation=*"http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-3.0.xsd"*>  <bean id=*"userBo"* class=*"org.csystem.bean.UserBoImpl"*>    </bean>  </beans> |

faces-config.xml

|  |
| --- |
| <?xml version=*"1.0"* encoding=*"UTF-8"*?>  <faces-config  xmlns=*"http://java.sun.com/xml/ns/javaee"*  xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*  xsi:schemaLocation=*"http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-facesconfig\_2\_0.xsd"*  version=*"2.0"*>  <application>  <el-resolver>org.springframework.web.jsf.el.SpringBeanFacesELResolver</el-resolver>  </application>  <managed-bean>  <managed-bean-name>user</managed-bean-name>  <managed-bean-class>org.csystem.bean.UserBean</managed-bean-class>  <managed-bean-scope>session</managed-bean-scope>  <managed-property>  <property-name>userBo</property-name>  <value>#{userBo}</value>  </managed-property>  </managed-bean>  </faces-config> |

web.xml

|  |
| --- |
| <?xml version=*"1.0"* encoding=*"UTF-8"*?>  <web-app xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"* xmlns=*"http://java.sun.com/xml/ns/javaee"* xmlns:web=*"http://java.sun.com/xml/ns/javaee/web-app\_2\_5.xsd"* xsi:schemaLocation=*"http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app\_3\_0.xsd"* version=*"3.0"*>  <display-name>JavaServerFaces</display-name>    <!-- Add Support for Spring -->  <listener>  <listener-class>  org.springframework.web.context.ContextLoaderListener  </listener-class>  </listener>  <listener>  <listener-class>  org.springframework.web.context.request.RequestContextListener  </listener-class>  </listener>  <!-- Change to "Production" when you are ready to deploy -->  <context-param>  <param-name>javax.faces.PROJECT\_STAGE</param-name>  <param-value>Development</param-value>  </context-param>  <context-param>  <param-name>contextConfigLocation</param-name>  <param-value>  /WEB-INF/applicationContext.xml  </param-value>  </context-param>      <welcome-file-list>  <welcome-file>index.jsp</welcome-file>  </welcome-file-list>  <servlet>  <servlet-name>Faces Servlet</servlet-name>  <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>  <load-on-startup>1</load-on-startup>  </servlet>  <servlet-mapping>  <servlet-name>Faces Servlet</servlet-name>  <url-pattern>\*.faces</url-pattern>  </servlet-mapping>  </web-app> |

default.xhtml

|  |
| --- |
| <?xml version=*"1.0"* encoding=*"UTF-8"*?>  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">  <html xmlns=*"http://www.w3.org/1999/xhtml"*  xmlns:h=*"http://java.sun.com/jsf/html"*  >    <h:body>    <h1>JSF 2.0 + Spring Example</h1>    #{user.printMsgFromSpring}    </h:body>    </html> |