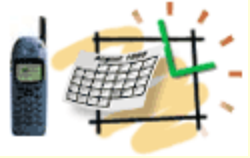
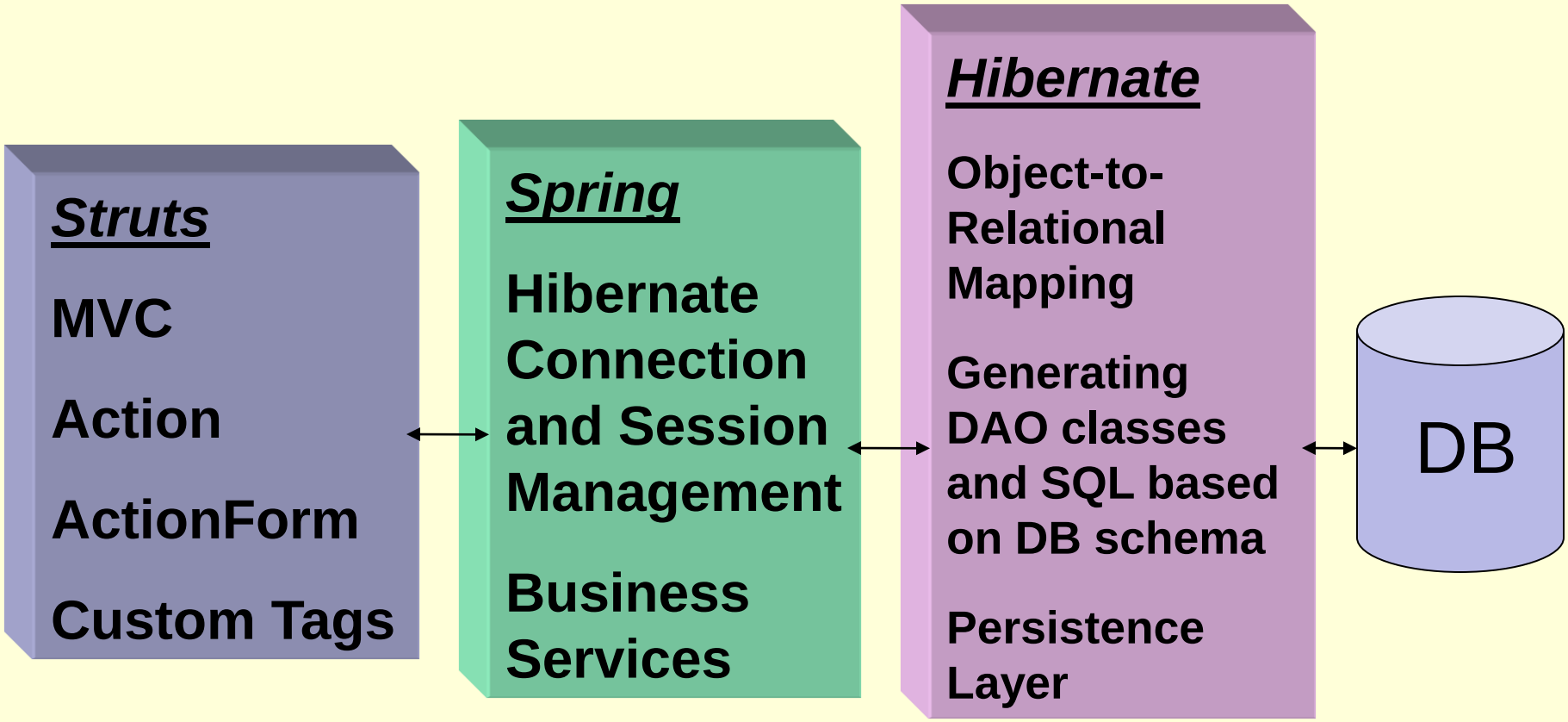


Spring & SOA

Jeff Zhuk, Greg Sternberg, Chris Justice



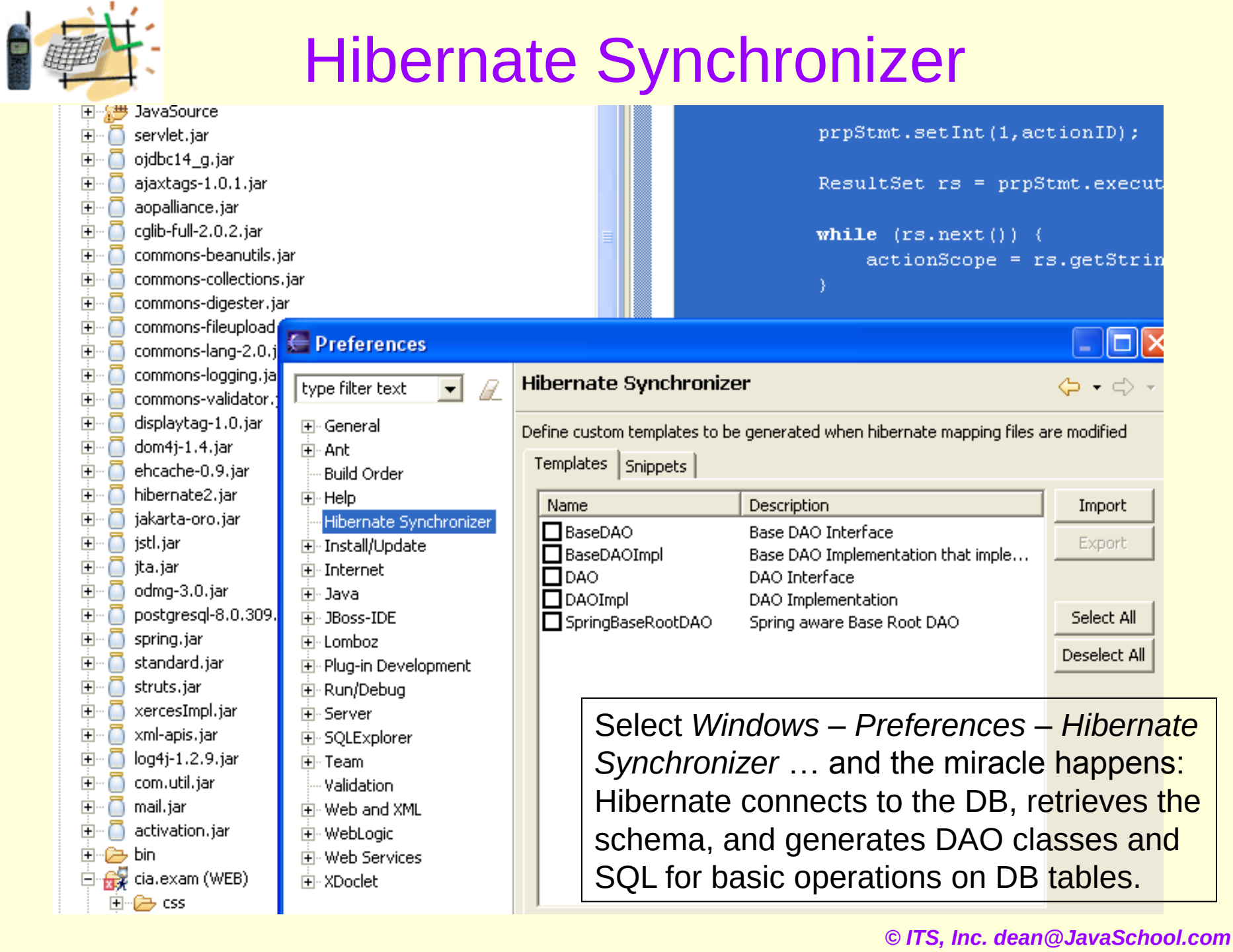
Consider Spring as an engine driving the show with several other software stars

Let's start with Hibernate, continue with Spring, Hollywood principle & more ☺

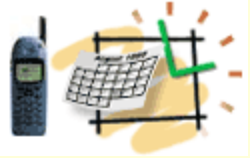
Hibernate

- Uses OO query language called HQL
- Uses objects instead of tables and fields instead of columns
- Provides object-to-relational mapping for most DBs
- Separates data layer from business logics
- Uses DB connection info to retrieve DB schema
- Generates DAO beans with data fields mapping table columns
- Generates Insert/Update/Delete/Select statements for DB tables

Hibernate Synchronizer



Select *Windows – Preferences – Hibernate Synchronizer* … and the miracle happens: Hibernate connects to the DB, retrieves the schema, and generates DAO classes and SQL for basic operations on DB tables.



Spring's Map to Hibernate

App-name.war

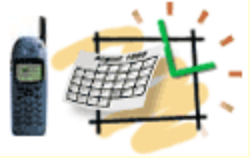
-WEB-INF

-- applicationContext.xml

```
<beans>
  <!-- == PERSISTENCE DEFINITIONS ===== -->

  <bean id="myDataSource"
  class="org.springframework.jndi.JndiObjectFactoryBean">
      <property name="resourceRef"><value>true</value></property>
      <property name="jndiName">
         <value>jdbc/javatest</value>
      </property>
   </bean>

  <!– Connect to Hibernate and match your "dataSource" definition -->
  <bean id="mySessionFactory"
class="org.springframework.orm.hibernate.LocalSessionFactoryBean">
     <property name="mappingResources">
       <list>
         <value>CIAExamAnswer.hbm.xml</value>
         <value>UserRoles.hbm.xml</value>
         <value>InstructorCategory.hbm.xml</value>
       </list>
     </property>
```

Spring Maps Data Source Dialect and Provides Transaction Management for Hibernate Operations

```
<property name="hibernateProperties">
        <props>
          <prop key="hibernate.dialect">
             net.sf.hibernate.dialect.PostgreSQLDialect</prop>
          <prop key="hibernate.show_sql">true</prop>
          <prop key="hibernate.cglib.use_reflection_optimizer">true</prop>
        </props>
      </property>

      <property name="dataSource">
        <ref bean="myDataSource"/>
      </property>
    </bean>
 <!-- Transaction manager for a single Hibernate SessionFactory -->
    <bean id="myTransactionManager"
class="org.springframework.orm.hibernate.HibernateTransactionManager">
      <property name="sessionFactory">
        <ref local="mySessionFactory"/></property>
    </bean>
```



Spring and Hibernate Reduce Business Code

The sessionFactory property and the mySessionFactory bean are related in the Spring configuration file.

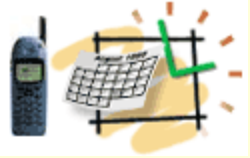
Spring creates described objects and factories that instantiate Hibernate DAO classes at run-time.

Spring simplifies the Hibernate configuration that otherwise would be stored in the *hibernate.cfg.xml* file.

The bottom line: Spring and Hibernate working together reduce your business code, ***especially when you operate with simple data records that reflect full table structure***.

Spring Framework:

- Allows software components to be first developed and tested in isolation capable of assembling a complex system from a set of loosely-coupled components in a consistent and transparent fashion.
- Conceals much complexity from the developer
- You can use all of Spring's functionality in any J2EE server
  - Postgress on JBoss on Linux
  - MySQL on Tomcat/Websphere on WinXP
- Objects can be reused across J2EE environments (web or EJB), standalone applications, test environments, etc with little hassle.
- Uses Inversion of Control. Also called the Hollywood Principle: "Don't call me, I'll call you."



Inversion of Control

- IoC moves the responsibility for making things happen into the framework, and away from application code. Non-ioc code calls a traditional class library, an IoC framework calls your code.
  - Similar ones – Windows event programming, Messaging Servers, …

```
<bean id="payloadMapping"
    class="org.springframework.ws.server.endpoint.mapping.PayloadRootQNameEndpointMapping">
   <property name="mappings">
      <props>
         <prop key="{http://127.0.0.1/myApp/}authenticationRequest">authentication</prop>
      </props>
   </property>
</bean>

<!-- Map bean names to objects -->
<bean id="authentication" class="com.its.usermanagement.Web.AuthenticationEndpoint">
   <constructor-arg>
      <bean class="com.its.usermanagement.services.Impl.AuthenticationServiceImpl"/>
   </constructor-arg>
</bean>
```

Inversion of Control

- Spring is most closely identified with Inversion of Control known as Dependency Injection. Dependency Injection is a form of IoC that removes explicit dependence on container APIs; ordinary Java methods are used to inject dependencies such as collaborating objects or configuration values into application object instances.

```
<bean id="authentication" class="com.its.usermanagement.Web.AuthenticationEndpoint">
   <constructor-arg>
      <bean class="com.its.usermanagement.services.Impl.AuthenticationServiceImpl"/>
   </constructor-arg>
</bean>

public class AuthenticationEndpoint extends AbstractDomPayloadEndpoint
{
   @Override
   protected Element invokeInternal (Element request_element, Document doc_base)
   {
      parseRequest (request_element,
                    username,
                    password);

      // Is the user a valid user ?
      String id = _service.authentication (new String (username),
                                           new String (password));

      Element response = createResponse (doc_base,
                                         new String (username),
                                         id);

      return (response);
   }
}
```

Aspect Oriented Programming (AOP)

- When thinking of an object and its relationship to other objects we often think in terms of inheritance.
  - Base object – **MovieMonster**
  - As we identify similar classes but with unique *behaviors* of their own, we often use inheritance to extend the functionality. For instance, if we identified a **JapaneseMonster** we could say a **JapaneseMonster** 'is-a' **MovieMonster**, so **JapaneseMonster** inherits **MovieMonster**.
  - So what happens when we define a *behavior* later on that we label as Thinking Monster? Not all **MovieMonsters** are thinking, so the **MovieMonster** class should not contain the thinking behavior. Furthermore, if we were to create a ThinkingMonster class that inherited from **MovieMonster**, then where would a **JapaneseMonster** fit in that hierarchy? A **JapaneseMonster** 'is-a' **MovieMonster**, but a **JapaneseMonster** may or may not be thinking; does **JapaneseMonster** then inherit from **MovieMonster**, or does **JapaneseMonster** inherit from Thinking Monster?
  - It is better to look at thinking as an aspect that we apply to any type of **MovieMonster** that is thinking, as opposed to inappropriately forcing that behavior in the **MovieMonster** hierarchy.
- In software terms, aspect-oriented programming allows us the ability to apply aspects that alter behavior to classes or objects independent of any inheritance hierarchy. We can then apply these aspects either during runtime or compile time.

AspectJ

- Joinpoint - Well defined point in the code

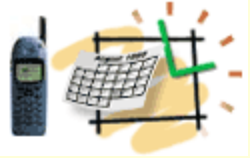
```
try {
} catch (ObjectNotFoundException onfe) {
} catch (Exception e) {
}
```

- Pointcut - A way of specifying a joinpoint

```
private pointcut handlingAnException(Throwable xcpt):
    handler(*)
    && (! within(com.its.Aspects..*))
    && args(xcpt);
```

- Advice – The cross cutting action that needs to occur

```
before(Throwable xcpt): handlingAnException(xcpt)
{
    log.error ("Caught " + xcpt, xcpt);
}
```



AspectJ Results

- Given the following code:

```
try {
    throw new RuntimeException ("only a test");
} catch (Exception e) {
    // We forgot to log it
}
```

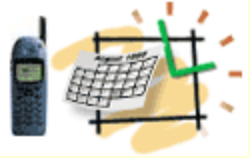
- this shows up in the log file:

```
2008-04-15 08:07:55,157 ERROR [main] Caught
   java.lang.RuntimeException: only a test
java.lang.RuntimeException: only a test
   at com.its.myApp.splashScreen(myApp.java:134)
   at com.its.myApp.main(myApp.java:170)
```

- Key point: This means we no longer have to insert code in every try/catch block throughout the code to log exceptions.

More AspectJ

```
private work (Object stuff)
{
   _logger.debug ("Entering package.class.work(" + stuff + ")";

   // Do work

   _logger.debug ("Exiting package.class.work ()");
}

private String authentication ()
{
   _logger.debug ("Entering authentication()");

   try {
      work (stuff);
   } catch (TransformerConfigurationException tce) {
      _logger.error ("Unable to configure DOM transformer", tce);

      ByteArrayOutputStream trace = new ByteArrayOutputStream ();
      tce.printStackTrace (new PrintStream (trace));
      _logger.error (trace.toString());
   } catch (TransformerException te) {
      _logger.error ("Unable to transform DOM into string", te);
      ByteArrayOutputStream trace = new ByteArrayOutputStream ();
      te.printStackTrace (new PrintStream (trace));
      _logger.error (trace.toString());
   }

   _logger.debug ("Exiting authentication(" + token + ")");

   return (token);
}
```



Even more AspectJ

```
private work (Object stuff)
{
   // Do work
}

private String authentication
()
{
   try {
      work (stuff);
   } catch
(TransformerConfigurationExce
ption tce) {
      _logger.error ("Unable
to configure DOM transformer
");
   } catch
(TransformerException te) {
      _logger.error ("Unable
to transform DOM into string
");
   }

   return (token);
}
```

JUnit (NUnit, fUnit, CPPUnit)

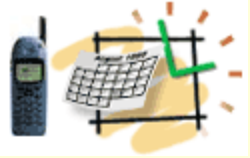
- **Test the methods in a class**

```
@BeforeClass
public static void setUp ()
{
}

@AfterClass
public static void tearDown ()
{
}

@Test
public void testParseRequest ()
{
   try {
      String expected_username = null;
      String expected_password = null;
…
      StringBuilder password = new StringBuilder ();
      StringBuilder username = new StringBuilder ();
…
      assertEquals (expected_username, new String (username));
      assertEquals (expected_password, new String (password));
   } catch (Exception e) {
      e.printStackTrace ();
      fail ("Caught " + e);
   }
}
```

More Testing

- **Test a bunch of classes**

```
@RunWith (Suite.class)
@Suite.SuiteClasses ({
  AuthenticationEndpointTest.class,

    AuthenticationServiceImplTest.cl
    ass,
  AuthorizationEndpointTest.class,

    AuthorizationServiceImplTest.cla
    ss,
  ConfigurationEndpointTest.class,

    ConfigurationServiceImplTest.cla
    ss,
})
public class
    UserManagementServicesTests
{
}
```



XML Runtime vs. Compile time binding

- XPath
  - Slower
  - More flexible
  - Adaptable to 'unimportant' XML changes
  - Doesn't require strict XSD adherence
    - Validation done elsewhere
  - Good when XML is changing

```
try {
   XPathExpression expr =
   xpath.compile
   ("/tns:authenticationRequest/t
   ns:username");
   username = expr.evaluate
   (request);

   expr = xpath.compile
   ("/tns:authenticationRequest/t
   ns:password");
   password = expr.evaluate
   (request);

} catch (XPathExpressionException
   xpee) {
   _logger.error ("Wasn't able to
   handle XPATH expression");
}
```

- Castor
  - Faster
  - Less flexible
  - Unable to adapt to 'unimportant' XML changes
  - Strict adherence to XSD
  - Good when XML is static

```
SourceGenerator srcGen = new
   SourceGenerator ();

srcGen.generateSource
   ("Person.xsd", "bindtest");

FileReader reader = new FileReader
   ("persondataxml.xml");

Person person = Person.unmarshal
   (reader);
person.setName("Sireen");

FileWriter writer = new FileWriter
   ("genXML.xml");

person.marshal (writer);
```

Hibernate

- POJO

```
public class User
{
    …
    // Constructor
    public User () { }

    // Getters and setters
    public void setId (String id) { }
    public String getId () { }
    public void setUsername (String username) { }
    public String getUsername () { }
    public void setPassword (String passwd) { }
    public String getPassword () { }
    public void setFirstname (String first_name) {
    }
    public String getFirstname () { }
    public void setLastname (String last_name) { }
    public String getLastname () { }
}
```

```
+----------+-------------+------+-----+---------+
| Field    | Type        | Null | Key | Default |
+----------+-------------+------+-----+---------+
| ID       | varchar(32) | YES  |     | NULL    |
| USERNAME | varchar(20) | YES  |     | NULL    |
| PASSWORD | varchar(20) | YES  |     | NULL    |
| FIRSTNAME| varchar(100)| YES  |     | NULL    |
| LASTNAME | varchar(100)| YES  |     | NULL    |
+----------+-------------+------+---------------
```

Map from the POJO to/from the Database

```
<hibernate-mapping>
   <class name="com.its.usermanagement.User"
table="USERS">
      <id name="username" column="USERNAME"
type="string"/>
      <property name="id" column="ID"/>
      <property name="password" column="PASSWORD"/>
      <property name="firstname"
column="FIRSTNAME"/>
      <property name="lastname" column="LASTNAME"/>
   </class>
</hibernate-mapping>
```

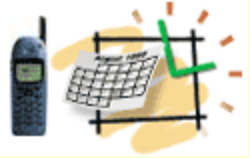
Spring IoC With Hibernate

- **Switching from one to the other is as simple as changing a line of configuration:**

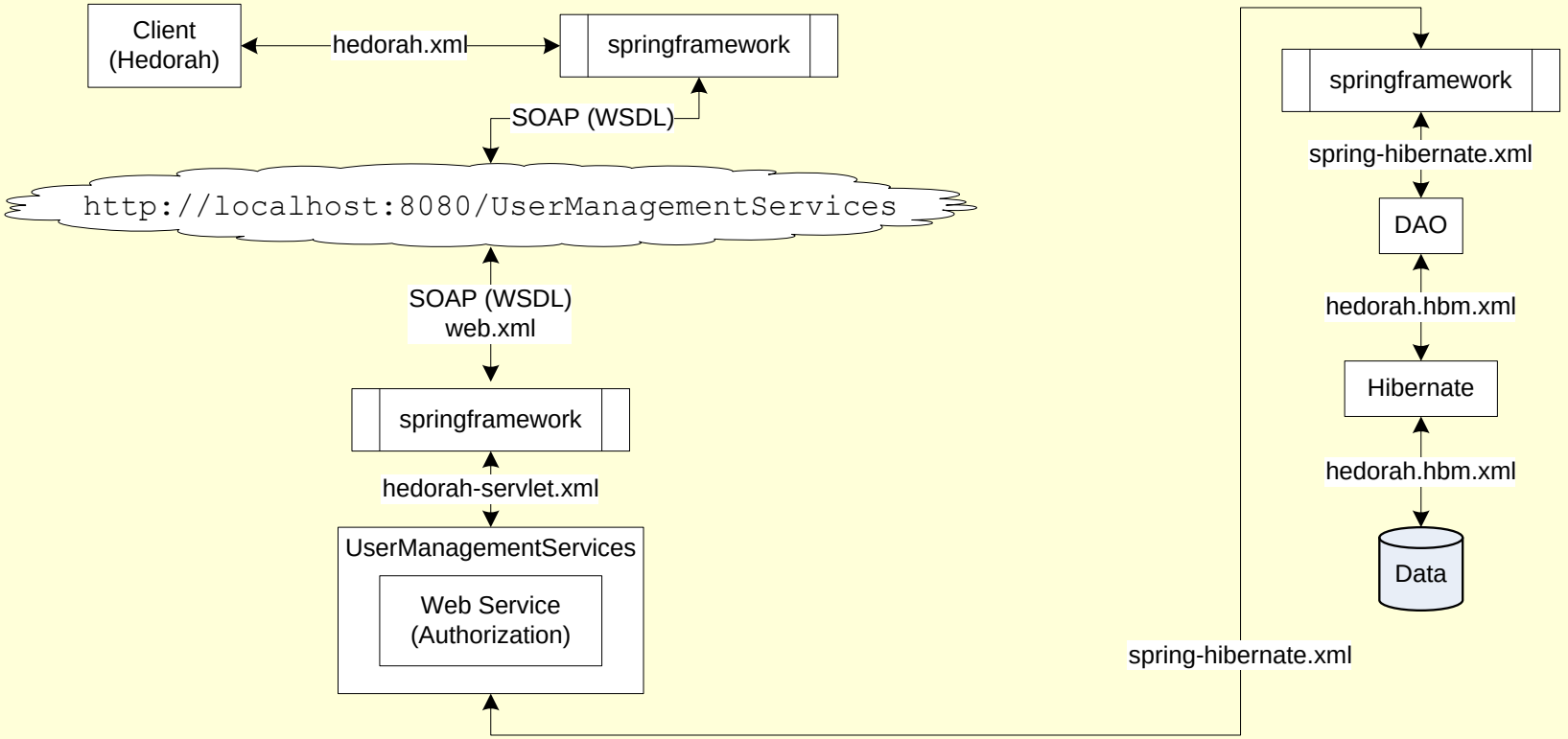
```
<bean id="hibernateTemplate"
      class="org.springframework.orm.hibernate3.Hib
      ernateTemplate">
  <property name="sessionFactory">
    <ref bean="mysqlSessionFactory"/>
<!--
    <ref bean="postgresSessionFactory"/>
-->
  </property>
</bean>

<!-- DAO bean definitions -->
<bean id="userDao"
      class="com.its.usermanagement.Dao.Impl.Use
      rDaoImpl">
  <property name="hibernateTemplate">
    <ref bean="hibernateTemplate"/>
  </property>
</bean>
```

- **Development code doesn't change at all:**

```
public class UserDaoImpl implements IUserDao
{
   User getUserDao (String username, String passwd)
   {
      HibernateCallback callback = new
    HibernateCallback () {
        public Object doInHibernate (Session session)
    throws HibernateException, SQLException
        {
           Object rc = new User ();

           try {
              rc = session.load (User.class,
    username);
           } catch (ObjectNotFoundException onfe) {
           }
           return (rc);
        }
      };
      return ((User)hibernateTemplate.execute
    (callback);
   }

   public void saveOrUpdate (final User user)
   {
      HibernateCallback callback = new
    HibernateCallback () {
        public Object doInHibernate (Session session)
    throws HibernateException, SQLException
        {
           session.saveOrUpdate (user);
           return (null);
        }
      };
      hibernateTemplate.execute (callback);
   }
}
```



How It Hooks Together



Define the Interface

- **Currently only used by Web Service; will eventually be used by client as well**

```
<element name="authenticationRequest">
  <complexType>
    <sequence>
      <element name="username" type="string"/>
      <element name="password" type="string"/>
      <choice>
        <sequence>
          <element name="corporateCode" type="string"/>
          <element name="airline" type="tns:AirlineCodeType" minOccurs="0"/>
        </sequence>
        <element name="airline" type="tns:AirlineCodeType"/>
      </choice>
    </sequence>
  </complexType>
</element>
<element name="authenticationResponse">
  <complexType>
    <sequence>
      <element name="username" type="string"/>
      <element name="id" type="string"/>
    </sequence>
  </complexType>
</element>
<element name="authenticationError">
  <complexType>
    <sequence>
      <element name="username" type="string"/>
      <element name="error" type="string"/>
    </sequence>
  </complexType>
```

Write the Client POJO

- **Invoke the authentication web service**

```
// Create the root element
Element root = doc.createElementNS ("http://127.0.0.1/myApp/", "tns:authenticationRequest");

root.setAttribute ("xmlns:tns", "http://127.0.0.1/myApp/");
doc.appendChild (root);

// Create the child elements
Element username = doc.createElementNS ("http://127.0.0.1/myApp/", "tns:username");
username.setTextContent (_username.getText());
root.appendChild (username);

Element passwd = doc.createElementNS ("http://127.0.0.1/myApp/", "tns:password");
passwd.setTextContent (new String (_password.getPassword()));
root.appendChild (passwd);

// Tie doc to source

webServiceTemplate.sendSourceAndReceiveToResult (source, result);

// Retrieve the result

if (root.getNodeName().equals("tns:authenticationResponse")) {
   XPathExpression expr = myApp._xpath.compile ("//tns:authenticationResponse/id");
   id = (String)expr.evaluate (doc, XPathConstants.STRING);
} else if (root.getNodeName().equals("tns:authenticationError")) {
   XPathExpression expr = myApp._xpath.compile ("//tns:authenticationError/error");
   String error = (String)expr.evaluate (doc, XPathConstants.STRING);
}
```

Wire Client Code

- Tie/Wire Client Code to the URL where the service resides

```
<bean id="loginDialog" class="com.its.LoginDialog">
<property name="userManagementURI"
   value="http://localhost:8080/UserManagementServices/"/>
</bean>
```

Create User POJO

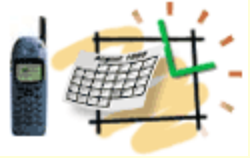
```
public class User
{
   private String/*UUID*/ _id;
   private String _username;
   private String _password;
   private String _first_name;
   private String _last_name;

   // Constructor
   public User () { }

   // Getters and setters
   public void setId (String id) { }
   public String getId () { }
   public void setUsername (String username) { }
   public String getUsername () { }
   public void setPassword (String passwd) { }
   public String getPassword () { }
   public void setFirstname (String first_name) { }
   public String getFirstname () { }
   public void setLastname (String last_name) { }
   public String getLastname () { }
}
```

Create Database

```
+-----------+--------------+------+-----+---------+
| Field     | Type         | Null | Key | Default |
+-----------+--------------+------+-----+---------+
| ID        | varchar(32)  | YES  |     | NULL    |
| USERNAME  | varchar(20)  | YES  |     | NULL    |
| PASSWORD  | varchar(20)  | YES  |     | NULL    |
| FIRSTNAME | varchar(100) | YES  |     | NULL    |
| LASTNAME  | varchar(100) | YES  |     | NULL    |
+-----------+--------------+------+-----+---------+
```

Tie Database to POJO

```
<hibernate-mapping>
   <class name="com.its.usermanagement.User"
  table="USERS">
      <id name="username" column="USERNAME"
  type="string"/>
      <property name="id" column="ID"/>
      <property name="password" column="PASSWORD"/>
      <property name="firstname" column="FIRSTNAME"/>
      <property name="lastname" column="LASTNAME"/>
   </class>
</hibernate-mapping>
```



Wire Hibernate & Spring

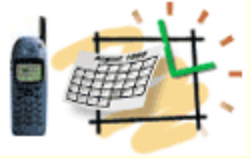
```
<bean id="mysqlDataSource" class="org.apache.commons.dbcp.BasicDataSource">
      <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
      <property name="url" value="jdbc:mysql://localhost:3306/myApp"/>
      <property name="username" value="root"/>
      <property name="password" value="password"/>
   </bean>

   <bean id="mysqlSessionFactory"
    class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
      <property name="dataSource" ref="mysqlDataSource"/>
      <property name="mappingResources">
         <list>
            <value>myApp.hbm.xml</value>
         </list>
      </property>
      <property name="hibernateProperties">
         <value>hibernate.dialect=org.hibernate.dialect.HSQLDialect</value>
      </property>
   </bean>

<bean id="hibernateTemplate"
    class="org.springframework.orm.hibernate3.HibernateTemplate">
      <property name="sessionFactory">
         <ref bean="mysqlSessionFactory"/>
</property>
   </bean>

   <!-- DAO bean definitions -->
   <bean id="userDao" class="com.its.usermanagement.Dao.Impl.UserDaoImpl">
      <property name="hibernateTemplate">
         <ref bean="hibernateTemplate"/>
      </property>
   </bean>
```

Write Hibernate Layer

```
public class UserDaoImpl implements IUserDao
{
   public User getUserDao (final String username,
                           final String passwd)
   {
      HibernateCallback callback = new HibernateCallback () {
         public Object doInHibernate (Session session) throws
   HibernateException, SQLException
         {
            Object rc = null;
            try {
               rc = session.load (User.class, username);
               _logger.debug ("Loaded " + (User)rc);
            } catch (ObjectNotFoundException onfe) {
               _logger.debug ("User '" + username + "' doesn't exist");
               rc = new User ();
            } catch (Exception e) {
               _logger.debug ("Error getting user " + username, e);
               rc = new User ();
            }
            return (rc);
         }
      };
      User rc = (User)hibernateTemplate.execute (callback);
      return (rc);
   }
}
```

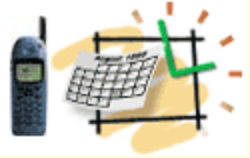
Write Web Service

```
public class AuthenticationEndpoint extends AbstractDomPayloadEndpoint
{
   private void parseRequest (Element request, StringBuilder username, StringBuilder password)
   {
      xpath.setNamespaceContext (new myAppNamespaceContext());
      try {
         XPathExpression expr = xpath.compile ("/tns:authenticationRequest/tns:username");
         username.append (expr.evaluate(request, XPathConstants.STRING));
         expr = xpath.compile ("/tns:authenticationRequest/tns:password");
         password.append (expr.evaluate(request, XPathConstants.STRING));
      } catch (XPathExpressionException xpee) {
      }
   }
   Element createResponse (Document doc_base, String username, String id)
   {
      if (id == null) {
         rc = doc_base.createElementNS ("http://127.0.0.1/myApp/", "tns:authenticationError");
         child = doc_base.createElementNS ("http://127.0.0.1/myApp/", "tns:error");
         child.setTextContent ("Invalid user");
      } else {
         rc = doc_base.createElementNS ("http://127.0.0.1/myApp/",
    "tns:authenticationResponse");
         child = doc_base.createElementNS ("http://127.0.0.1/myApp/", "tns:id");
         child.setTextContent (id);
      }
      rc.appendChild (child);
      child = doc_base.createElementNS ("http://127.0.0.1/myApp/", "tns:username");
      child.setTextContent (username);
      rc.appendChild (child);
      return (rc);
   }
   @Override
   protected Element invokeInternal (Element request_element, Document doc_base)
   {
      parseRequest (request_element, username, password);
      String id = _service.authentication (new String (username), new String (password));
      Element response = createResponse (doc_base, new String (username),  id);
      return (response);
   }
}
```



Write Authentication Service

```
public class AuthenticationServiceImpl implements
    IAuthenticationService
{
   public AuthenticationServiceImpl ()
   {
      Resource resource = new ClassPathResource ("spring-
   hibernate.xml");
      BeanFactory bean_factory = new XmlBeanFactory (resource);

      _user_dao = (IUserDao)bean_factory.getBean ("userDao");
   }

    public String authentication (String username, String password)
    {
       User user = _user_dao.getUserDao (username, password);

       String id = null;

       if (user.getId() != null) {
          id = user.getId();
       }

       return (id);
    }
}
```



Write unit test

```
@Test
public void testInvokeInternalElementDocument ()
{
   DOMParser parser = new DOMParser();
   File fh = new File
    ("src/com/gws/myApp/usermanagement/Tests/data/AuthenticationEndpointTest.xml")
    ;

   try {
      parser.parse (new InputSource (new FileInputStream (fh)));

      Document result = parser.getDocument ();
      Element request = result.getDocumentElement ();

      IAuthenticationService service = new AuthenticationServiceImpl ();
      AuthenticationEndpoint endpoint = new AuthenticationEndpoint (service);

      Element actual = (Element)UnitTestUtils.invokePrivateMethod (endpoint,

    "invokeInternal",
                                                                  request,
                                                                  result);

      assertTrue (actual != null);
      assertEquals ("tns:authenticationResponse", actual.getTagName());
      assertTrue (actual.getTextContent().contains(_valid_user.getId()));
   } catch (Exception e) {
      e.printStackTrace ();
      fail ("Caught " + e);
   }
}
```

Wire Web Service To Request

```
<bean id="payloadMapping"
   class="org.springframework.ws.server.endpoint.mapping.PayloadRootQNameEndpoi
   ntMapping">
   <property name="interceptors">
      <list>
         <ref local="validatingInterceptor"/>
      </list>
   </property>
   <property name="mappings">
      <props>
         <prop
   key="{http://127.0.0.1/myApp/}authenticationRequest">authentication</prop>
      </props>
   </property>
</bean>
<bean id="validatingInterceptor"
   class="org.springframework.ws.soap.server.endpoint.interceptor.PayloadValida
   tingInterceptor">
   <property name="schema" value="/WEB-INF/xsd/UserManagement.xsd"/>
   <property name="validateRequest" value="true"/>
   <property name="validateResponse" value="true"/>
</bean>
<bean id="authentication"
   class="com.its.usermanagement.Web.AuthenticationEndpoint">
   <constructor-arg>
      <bean
   class="com.its.usermanagement.services.Impl.AuthenticationServiceImpl"/>
   </constructor-arg>
</bean>
```

Wire Service to WSDL

- **Autogenerate for simplicity**

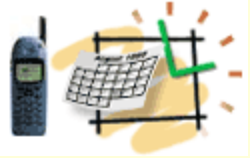
```
<bean id="userManagement"
   class="org.springframework.ws.wsdl.wsdl11.DynamicWsdl11Definition">
   <property name="builder">
      <bean
   class="org.springframework.ws.wsdl.wsdl11.builder.XsdBasedSoap11Wsdl4jDefini
   tionBuilder">
         <property name="schema" value="/WEB-INF/xsd/UserManagement.xsd"/>
         <property name="portTypeName" value="myApp"/>
         <property name="locationUri"
   value="http://localhost:8080/UserManagementServices/"/>
      </bean>
   </property>
</bean>
```

- **Have all requests and responses handled by Spring**

```
<servlet>
   <servlet-name>myApp</servlet-name>
   <servlet-
   class>org.springframework.ws.transport.http.MessageDispatcherServlet</servle
   t-class>
   <init-param>
      <param-name>transformWsdlLocations</param-name>
      <param-value>true</param-value>
   </init-param>
</servlet>
<servlet-mapping>
   <servlet-name>myApp</servlet-name>
   <url-pattern>/*</url-pattern>
</servlet-mapping>
```

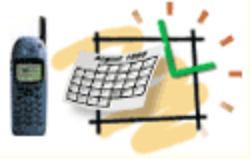
Resources

- Eclipse - http://www.eclipse.org/
- Hibernate – http://www.hibernate.org/
- Spring – http://www.springframework.org/
- AspectJ – http://www.eclipse.org/aspectj/



Spring JMS Integration

- This briefing will show how to use MDP/MDB to implement a service, using a point to point messaging model. See the Spring documentation for information on setting up a *publish – subscribe* messaging model.
- For a service, it is useful if the payload of the message is XML based.



MDP/MDB

- A Message Driven POJO (MDP) is just a java class that processes a message. The only requirement is that it implement the onMessage() from javax.jms.MessageListener.
- A Message Driven Bean (MDB) is an enterprise message bean that runs in a JEE container.

Creating a ConnectionFactory

```
<bean id="getMITSJMSFactory"
    class="org.apache.activemq.pool.PooledConnectionFactory" destroy-
    method="stop">
  <property name="connectionFactory">
    <bean class="org.apache.activemq.ActiveMQConnectionFactory">
      <property name="brokerURL">
        <value>vm://localhost?broker.persistent=false</value>
      </property>
    </bean>
  </property>
</bean>
```

Or

Creating a ConnectionFactory

```
<bean id="getItsServiceJNDITemplate" class="org.springframework.jndi.JndiTemplate">
  <property name="environment">
    <props>
      <prop key="java.naming.factory.initial"> weblogic.jndi.WLInitialContextFactory </prop>
      <prop key="java.naming.provider.url"> ${weblogic.server} </prop>
    </props>
  </property>
 </bean>

 <bean id="getItsServiceConnectionFactory"
     class="org.springframework.jndi.JndiObjectFactoryBean">
  <property name="jndiName">
    <value>weblogic.jms.ConnectionFactory</value>
  </property>
  <property name="jndiTemplate">
    <ref bean="getItsServiceJNDITemplate"/>
  </property>
 </bean>
```

Creating a destination

```
<bean id="getItsRequestQueue"
      class="org.apache.activemq.command.ActiveMQQueue">
  <constructor-arg index="0" value="GET_MITS_REQUEST_QUEUE"/>
</bean>
```

Or

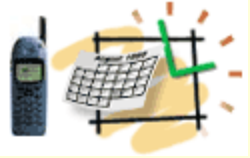
```
 <bean id="getItsServiceRequestQueue"
      class="org.springframework.jndi.JndiObjectFactoryBean">
    <property name="jndiName">
      <value>com.its.jms.queue.GetItsServiceRequestQueue</value>
    </property>
    <property name="jndiTemplate">
      <ref bean="getItsServiceJNDITemplate"/>
    </property>
  </bean>
```

JMSTemplate

- Spring's JmsTemplate class reduces repetitive JMS code. It creates a connection, obtains a session, and deals with the *sending* and *receiving* of messages. This allows you to focus on constructing messages or processing them.
- JmsTemplate converts checked JMSExceptions into unchecked Spring JmsExceptions.
- JmsTemplate uses a ConnectionFactory and a destination.

How to build a MDP

- To build a Message Driven POJO (MDP), you need something to receive a message, and something to reply.
- Spring provides several ListenerContainers that will watch a JMS destination, waiting for a message to arrive, and then passes it onto an injected class that implements javax.jms.MessageListener. This class is your MDP.
- You can then use JmsTemplate to reply to the message from your MDP.



MDP Configuration

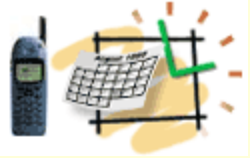
- First, lets create our MDP. Notice it uses a JmsTemplate, which we will discuss in a few more slides.

```
<bean id="getMITSJMSBean" class
    ="com.its.jadservice.getmits.GetItsServiceMDP">
    <property name="responseJMSTemplate">
      <ref bean="getMITSJMSTemplate"/>
    </property>
</bean>
```

Create a MessageListener

- Next we will create a DefaultMessageListenerContainer. This could optionally use a transaction manager, such as JTA.

```
<bean id="getMITSListenerContainer"
    class="org.springframework.jms.listener.DefaultMessageListenerContai
    ner">
  <property name="connectionFactory" ref="getMITSJMSFactory" />
  <property name="destination" ref="getItsRequestQueue" />
  <property name="messageListener" ref="getMITSJMSBean" />
</bean>
```



The MDP

- Now we need to write the MDP that implements MessageListener:

```
public class GetItsServiceMDP implements MessageListener
{

protected static Log LOG = LogFactory.getLog( GetItsServiceMDP.class );

// injected used to send a response back to the requestor
protected JmsTemplate responseJMSTemplate;

// The guts of the MessageListener interface. This is where the message driven
    work
// should be done. By design, this interface does not support exception throwing, so
// all non-runtime exceptions must be dealt with here
public void onMessage(Message message)
{
```

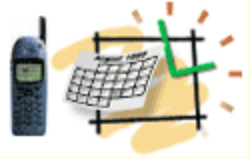
onMessage()

- Next, we need to write the onMess

```
public void onMessage(Message message)
{

String correlationId = "";
correlationId = message.getJMSCorrelationID();

// process messsage here…

// If the payload is XML based, you can marshall it  to ob
// send the response on the reply to the destination
responseJMSTemplate.convertAndSend( replyTo, responseMap, new MessagePostProcessor()
{

public Message postProcessMessage(Message message) throws JMSException
{
// need to set this in case the sender is matching up replies with the request
message.setJMSCorrelationID( correlationId );
return message;
}
} );
```

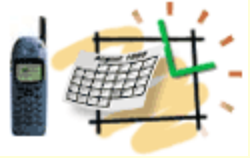
The second argument for convertAndSend() is an object, and it converts this object into message. For a string it create TextMessage, for a byte array BytesMessage, for a Map a mapMessage, and for serializa objects, and ObjectMessage

A MessageProcessor can be used to manipulate the message. In this case we are setting the correlation id.



What about MDBs?

- Creating MDPs are nice, but what if you are running in a JEE container and want to take advantage of Message Driven Beans (MDBs)?
  - Use Spring's AbstractJmsMessageDrivenBean and use it to call your Spring MDP.



Create a Spring enabled MDB

- First create an MDB as you normally would.
- Except have it inherit from Spring's AbstractJMSMessageDrivenBean and implement MessageListener.
- Use Spring's BeanFactory to get your MDP.
- Code the onMessage() to delegate to your MDP.

MDB Code

```
/**
 * @ejb.bean
 *      name = "GetItsServiceMDB"
 *      jndi-name = "com.its.jms.GetItsServiceMDBHome"
 *      transaction-type="Container"
 *      destination-type="javax.jms.Queue"
 *
 * @ejb.transaction
 *      type = "NotSupported"
 *
 * @weblogic.pool
 *          max-beans-in-free-pool     = "1"
 *          initial-beans-in-free-pool = "1"
 *
 * @weblogic.message-driven
 *          destination-jndi-name="com.its.jms.queue.GetItsServiceRequestQueue"
 *
 * @ejb.env-entry
 *      name = "ejb/BeanFactoryPath"
 *      type = "java.lang.String"
 *      value = "get-mits-service-ejb-context.xml"
 *
 *  GetItsService Message Driven Bean which forwards messages to a pojo which supports the
 *  MessageListener Interface. Implementers can use this to deploy an MDB to
 *  a container like WebLogic, but the forward MDB requests to a pojo, which
 *  could be configured to run outside the container as well.  This allows
 *  unit and integration tests to run outside the container, while deploying
 *  actual MDB's for the test and prod environments
 */
public class GetItsServiceMDB
extends AbstractJmsMessageDrivenBean
implements MessageListener
{
```

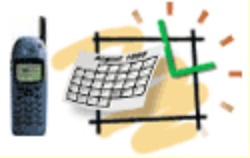
MDB Code

```
private static Log log = LogFactory.getLog(GetItsServiceMDB.class);

private static final long serialVersionUID = -1L;

private MessageListener getItsServiceMDP = null;

/**
 * Hook method for ejbCreate fr
 * Getting the ESLServiceMDP
 * forward the message to mess
 */
protected void onEjbCreate()
{
  getItsServiceMDP =
     (MessageListener)getBeanFactory().getBean("getItsServiceMDP");
}
```

When the EJB is created, we retrieve the BeanFactory, and then retrieve the MDP.

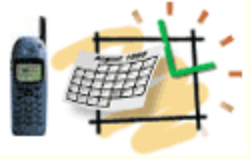
MDB Code

```
/**
 *
 * @ejb.interface-method MessageDrivenBean type = "remote"
*/
public void onMessage(Message message)
{
if (getItsServiceMDP != null)
{
try
{
log.debug("Forwarding message to GetItsServiceMDP");
getItsServiceMDP.onMessage(message);
}
catch (Exception ex)
{
log.error("Exception: " + ex.getMessage(), ex);
}
}
else
{
log.error("Unable to forward request to message driven pojo for processing");
}
}
```



MDB Code

```
/**
 * Setter for GetItsServiceMDP
 *
 * @param getItsServiceMDP
 */
public void setGetItsServiceMDP(MessageListener
    getItsServiceMDP)
{
    this.getItsServiceMDP = getItsServiceMDP;
}
```

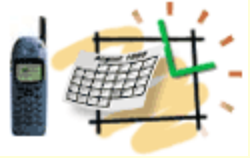


Spring JMS Summary

- Spring allows you to easily build MDPs and MDBs.
- You can use MDPs when you don't want to deploy in a full blown JEE container, and have a JMS provider available. This is also useful for unit testing.
- Use MDBs when you have the services of a JEE Application Server. Delegate to the MDP. By putting the service logic in the MDP you make the logic unit testable without needing to be deployed in an App Server.

Spring Web Services

- Spring provides two ways to build web services
  - Using an XFireExporter
  - Using the Spring Web Services (WS) Framework.
- Using the XFireExporter is simple, but exposes your entire bean to the world.
- Spring WS uses a contract-first approach. You design XML Messages first and then create an XSD . These are then used to create a WSDL dynamically. This is more work, but there are advantages. We will explore how to use Spring WS.



Spring WS First Step

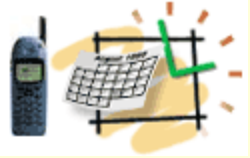
- The first step to creating a Spring Web Service is to create an XSD for use by the soap service, i.e. A request and response message.

XSD Creation

- There are several ways to do this.
  - Create an XML Instance, and use a tool like Trang, XMLSpy or Oxygen to generate a schema
  - Hand code the XSD
  - Ask a friend to do it.

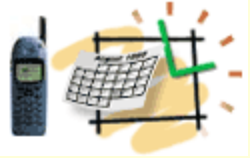
Sample XSD

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
   targetNamespace="http://www.its.com/JAD/GetItsRequest" elementFormDefault="qualified"
   attributeFormDefault="unqualified">
   <xs:element name="GetItsRequest">
      <xs:complexType>
         <xs:sequence maxOccurs="1" minOccurs="1">
            <xs:element name="Provider" nillable="false" minOccurs="1" maxOccurs="1">
               <xs:simpleType>
                  <xs:restriction base="xs:string">
                     <xs:enumeration value="USA, FAA (NFDC)">
                     </xs:enumeration>
                  </xs:restriction>
               </xs:simpleType>
            </xs:element>
            <xs:element name="ReceivedDate" nillable="false" minOccurs="1" maxOccurs="1" type="xs:dateTime">
            </xs:element>
            <xs:element name="FirstEffectiveDate" nillable="false" minOccurs="1" maxOccurs="1" type="xs:dateTime">
            </xs:element>
            <xs:element name="PublishedDate" nillable="false" minOccurs="1" maxOccurs="1" type="xs:dateTime">
            </xs:element>
            <xs:element name="LastEffectiveDate" nillable="false" minOccurs="1" maxOccurs="1" type="xs:dateTime">
            </xs:element>
         </xs:sequence>
      </xs:complexType>
   </xs:element>
   <xs:element name="GetMitsResponse">
      <xs:complexType>
            <xs:sequence minOccurs="1" maxOccurs="1">
            <!-- The Resulting MITS Number -->
            <xs:element name="MitsNumber" type="xs:string" minOccurs="0" maxOccurs="1" nillable="false"/>
            </xs:sequence>
      </xs:complexType>
   </xs:element>
</xs:schema>
```



2nd Step

- Spring WS uses Spring MVC to provide a servlet to handle SOAP messages. Specifically it uses MessageDispatcherServlet, which is a subclass of DispatcherServlet.
- To configure your web server to use it, configure the servlet in web.xml:

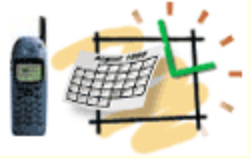


Web.xml

```
<!DOCTYPE web-app PUBLIC
 "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
 "http://java.sun.com/dtd/web-app_2_3.dtd" >

<web-app>
<display-name>MITS SERVICES</display-name>

<servlet>
<servlet-name>mits</servlet-name>
<servlet-
    class>org.springframework.ws.transport.http.MessageDispatcherServlet</servle
    t-class>
<load-on-startup>1</load-on-startup>
</servlet>

<servlet-mapping>
<servlet-name>mits</servlet-name>
<url-pattern>/*</url-pattern>
</servlet-mapping>
</web-app>
```

Endpoints

- The MessageDispatcherServlet, with a little routing help, will send the requests to a Service Endpoint. The Service Endpoint takes the XML message, marshalls it to objects, and passes it to your internal application objects. Once complete, it will unmarshall objects back to XML, and return a response.



Endpoints

- Spring provides several abstract endpoint classes to help you deal with the XML using your choice of parser technology. Endpoint choices include:
  - AbstractDom4jPayloadEndpoint
  - AbstractDomPayloadEndpoint
  - AbstractJDomPayloadEndpoint
  - ***AbstractMarshallingPayloadEndpoint***
  - AbstractSaxPayloadEndpoint
  - AbstractStaxEventPayloadEndpoint
  - AbstractStaxStreamPayloadEndpoint
  - AbstractXomPayloadEndpoint

AbstractMarshalling PayloadEndpoint

- The different endpoints automatically parse the XML messages into xml objects using the applicable XML technology. (i.e. Elements for JDom, etc.)
- AbstractMarshallingPayloadEndpoint lets you use a tool like XMLBeans or Castor to marshall your objects from XML to POJOs. This keeps you from writing a lot of ugly XML parsing code.

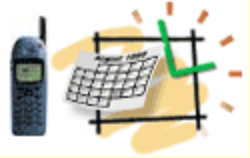
Endpoint code

```
public class ItsServiceMarshallingEndpo                                    Endpoint
{

/* (non-Javadoc)
 * @see
       org.springframework.ws.server.end                                   #invokeInternal(java.lang.Obj
       ect)
 */
@Override
protected Object invokeInternal(Object object) thr

                ent rd = (GetItsRequestDocu
                st = rd.getGetItsRequest();

String mits      er = ItsService.getMits( request );

GetMitsResponseDocument responseDocument = GetMitsResponseDocument.Factory.newInstance();
GetMitsResponse response = GetMitsResponse.Factory.newInstance();
response.setMitsNumber( mitsNumber );
responseDocument.setGetMitsResponse( response );
return responseDocument;
}

// Injected
private ItsService ItsService;
public void setItsService(ItsService ItsService) {
this.ItsService = ItsService;
}
}
```

InvokeEternal() is passed in a pojo that was marshalled from XML for you

Here we call our POJO that actually implements the service. The same one that we used in the JMS Examples!

And here we build a response object.



Wiring it Up

- First we need to tell the MessageDispatcherServlet how to route the incoming SOAP messages to our endpoint.
- We use an endpoint mapper to do this. Spring provides several.
- We are going to use PayloadRootQNameEndpointMapping, which maps incoming SOAP messages to endpoints by examining the qualified name (QName) of the message's payload and looking up the endpoint from its list of mappings (configured through the endpointMap property).

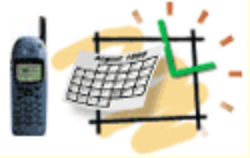
Wiring it Up

- The MessageDispatcherServlet looks for a configuration file in the WEB-INF directory using the instance name of the servlet (mits in our case) and the appends “-servlet.xml” to it.

Wiring it Up

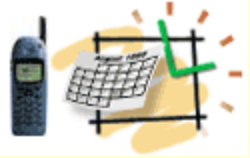
```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd">

<bean id="payloadMapping"
class="org.springframework.ws.server.endpoint.m
    meEndpointMapping">
<property name="endpointMap">
<map>
<entry
key="{http://www.its.com/JAD/GetItsRequest}GetItsRequest"
value-ref="ItsServiceEndpoint" />
</map>
</property>
</bean>
```

This was the namespace from the XSD, and then the request element.



Wiring it Up

```
<bean id="ItsServiceEndpoint"
    class="com.its.jadservice.getmits.ItsServiceMarshalli
    ngEndpoint">
<property name="marshaller" ref="marshaller" />
<property name="unmarshaller" ref="marshaller" />
<property name="ItsService" ref="ItsServiceBean" />
</bean>

<bean id="marshaller"
    class="org.springframework.oxm.xmlbeans.XmlBean
    sMarshaller">
</bean>
```

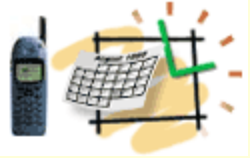


Marshallers

- Spring uses Marshallers to convert XML to objects and back. Several are provided out of the box.
  - Castor XML - org.springframework.oxm.castor.CastorMarshaller
  - JAXB v1 - org.springframework.oxm.jaxb.Jaxb1Marshaller
  - JAXB v2 - org.springframework.oxm.jaxb.Jaxb2Marshaller
  - JiBX - org.springframework.oxm.jibx.JibxMarshaller
  - XMLBeans - org.springframework.oxm.xmlbeans.XmlBeansMarshaller
  - XStream - org.springframework.oxm.xstream.XStreamMarshaller

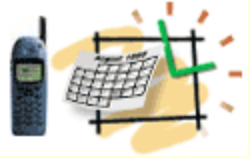
Marshallers

- Our example used XMLBeans, which were compiled against our XSD, and included in the classpath.
- If you used Castor, you would specify what Castor mapping file to use.



What about the WSDL?

- Spring can use a static WSDL you defined. Yuck! Use Spring's ***SimpleWsdl11Definition*** class for this.
- Or Spring will dynamically generate the WSDL using the XSD for your messages. To do this configure Spring's **DynamicWsdl11Definition** class. It works with the **MessageDispatcherServlet**.



Wire up the WSDL Generation

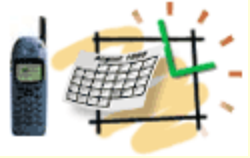
```
<bean id="mits"
  class="org.springframework.ws.wsdl.wsdl11.DynamicWsdl11Definition">
<property name="builder">
<bean
    class="org.springframework.ws.wsdl.wsdl11.builder.XsdBasedSoap11
    Wsdl4jDefinitionBuilder">
<property name="schema" value="/GetMits.xsd" />
<property name="portTypeName" value="Mits" />
<property name="locationUri"
value="http://localhost:7001/Mits-WS/services" />
</bean>
</property>
</bean>
```

The generated WSDL

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
        xmlns:schema="http://www.its.com/JAD/GetItsRequest"
        xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
        targetNamespace="http://www.its.com/JAD/GetItsRequest">
  <wsdl:types>
    <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" attributeFormDefault="unqualified"
       elementFormDefault="qualified" targetNamespace="http://www.its.com/JAD/GetItsRequest">
    <xs:element name="GetItsRequest">
       <xs:complexType>
          <xs:sequence maxOccurs="1" minOccurs="1">
       <xs:element maxOccurs="1" minOccurs="1" name="ReceivedDate" nillable="false" type="xs:dateTime">
             </xs:element>
             <xs:element maxOccurs="1" minOccurs="1" name="FirstEffectiveDate" nillable="false"
       type="xs:dateTime">
             </xs:element>
             <xs:element maxOccurs="1" minOccurs="1" name="PublishedDate" nillable="false"
       type="xs:dateTime">
             </xs:element>
             <xs:element maxOccurs="1" minOccurs="1" name="LastEffectiveDate" nillable="false"
       type="xs:dateTime">
             </xs:element>
          </xs:sequence>
       </xs:complexType>
    </xs:element>
    <xs:element name="GetMitsResponse">
       <xs:complexType>
             <xs:sequence maxOccurs="1" minOccurs="1">
             <!-- The Resulting MITS Number -->
             <xs:element maxOccurs="1" minOccurs="0" name="MitsNumber" nillable="false" type="xs:string"/>
             </xs:sequence>
       </xs:complexType>
    </xs:element>
</xs:schema>
```

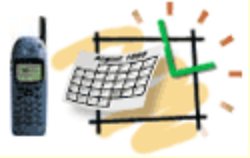
WSDL Continued

```
</wsdl:types>
  <wsdl:message name="GetMitsResponse">
    <wsdl:part element="schema:GetMitsResponse" name="GetMitsResponse">
    </wsdl:part>
  </wsdl:message>
  <wsdl:message name="GetItsRequest">
    <wsdl:part element="schema:GetItsRequest" name="GetItsRequest">
    </wsdl:part>
  </wsdl:message>
  <wsdl:portType name="Mits">
    <wsdl:operation name="GetMits">
      <wsdl:input message="schema:GetItsRequest" name="GetItsRequest">
    </wsdl:input>
      <wsdl:output message="schema:GetMitsResponse" name="GetMitsResponse">
    </wsdl:output>
    </wsdl:operation>
  </wsdl:portType>
  <wsdl:binding name="MitsBinding" type="schema:Mits">
    <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
    <wsdl:operation name="GetMits">
      <soap:operation soapAction=""/>
      <wsdl:input name="GetItsRequest">
        <soap:body use="literal"/>
      </wsdl:input>
      <wsdl:output name="GetMitsResponse">
        <soap:body use="literal"/>
      </wsdl:output>
    </wsdl:operation>
  </wsdl:binding>
  <wsdl:service name="ItsService">
    <wsdl:port binding="schema:MitsBinding" name="MitsPort">
      <soap:address location="http://localhost:7001/Mits-WS/services"/>
    </wsdl:port>
  </wsdl:service>
</wsdl:definitions>
```



Summary

- Spring provides framework support for JMS Services and Web Services.
- Details of JMS and SOAP are handled for you, letting you concentrate on business logic.
- Spring helps building SOA-enable enterprise.



Resources

- Walls, Craig. 2008. *Spring in Action.*
- http:/www.springframework.org
- http://static.springframework.org/spring-ws/site/