Spring JDBC Overview

- Classic JDBC vs. Spring JDBC
- JDBC Template
- DataSources
- Executing Statements
- Processing Results
- Updates

80

Notes:

Classic JDBC

- Given the following Driver

```
public class Driver {
        public static void main(String[] args) {
                ApplicationContext context =
                        new ClassPathXmlApplicationContext("beans.xml");

                CatalogManager catalog =
                        (CatalogManager) context.getBean("catalog");

                Iterator  iter = catalog.getProducts().iterator();

                while(iter.hasNext()) {
                   Product p = (Product)iter.next();
                   System.out.println(p);
                }
        }
}
```

82

Notes:

The following are sample properties that are used to connect to a database.

jdbc.properties

jdbc.driverClassName=com.ibm.db2.jcc.DB2Driver
jdbc.databaseName=jdbc : db2 : SAMPLE
jdbc.username=db2admin
jdbc.password=db2admin

Some Spring JDBC Classes

- JdbcTemplate (the workhorse of Spring JDBC)
  - MappingSqlQuery
  - SqlUpdate
- JdbcDaoSupport (a wrapper for Jdbc Template)

84

Notes:

Jdbc Template

- *JdbcTemplate* allows you to issue any type of SQL statement and return any type of result
  - It manages a *DataSource* object
  - You configure it from your XML definition file

There are over **70** Jdbc Template methods and over 40 to return various query results

- Some ***JdbcTemplate*** methods:

List execute(String sql, RowMapper rowMapper)
Map queryForMap(String sql)
Object queryForObject(String sql, RowMapper rowMapper)
public List queryForList(String sql)
~~public int queryForInt(String sql)~~
public int update(final String sql)

86

Notes:

Getting Data from JdbcTemplate

```
ApplicationContext context = new
    ClassPathXmlApplicationContext("applicationContext.xml");
CatalogManager catalog =
    (CatalogManager) context.getBean("catalog");

Iterator iter = catalog.getProducts().iterator();

while(iter.hasNext()) {
    Map product = (Map)iter.next();
    System.out.println(product);
}
```

Let's explore this method further ...

88

Notes:

This example grabs and displays products from the catalog by retrieving them through the Spring JDBC API. Here you can see the catalog object invoking *getProducts()*. This method retrieves and returns a collection of product objects.

Getting Data from JdbcTemplate ...

applicationContext.xml

```
<bean id="dataSource"
        class="org.apache.commons.dbcp.BasicDataSource"
        destroy-method="close">
    <property name="driverClassName" value="org.hsqldb.jdbcDriver"/>
    <property name="url" value="jdbc:hsqldb:hsql://localhost"/>
    <property name="username" value="student"/>
    <property name="password" value="student"/>
</bean>

<bean id="catalog"
        class="com.company.springclass.services.
                                        CatalogManagerImplementation">
    <property name="dataSource">
        <ref local="dataSource"/>
    </property>
</bean>
```

The datasource is injected into the Catalog Service

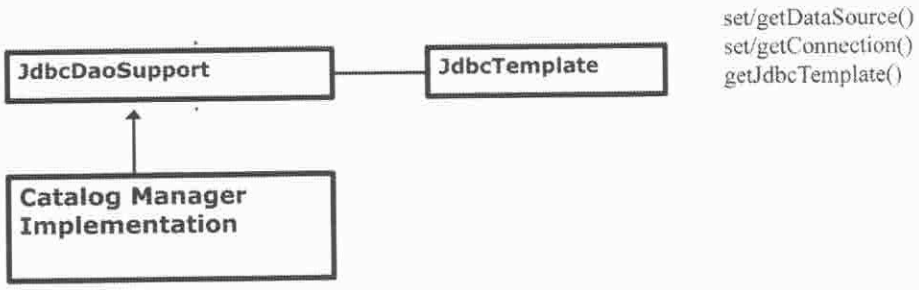
90

Notes:

Notice the Spring config file defines the data source. To use it, we can utilize Spring's dependency injection features.

JdbcDaoSupport

- *JdbcDaoSupport* wraps a JdbcTemplate for easy access to it
  - Your DAO should extend **JdbcDaoSupport**
- The primary **JdbcDaoSupport** methods are:



92

Notes:

Use this class to make Spring JDBC even easier.

Improving Repetitive JDBC

- In JDBC, retrieving a row of data usually involves the same step every time:
  - Take one row from the *ResultSet*
  - Extract the data from it
  - Instantiate a business object and insert the data
- In Spring, we can simplify this repetitive work by defining this task in a class called a ***RowMapper***

94

Notes:

Using the RowMapper

- Spring can now use the *RowMapper* to return collections of business objects rather than collections of Maps
- Remember that Spring calls *mapRow()*, not you!
- Spring passes the **ResultSet** into *mapRow()*

96

Notes:

Obtaining a Single Object

```
catalog.getProduct(1001);
```

This uses the same row mapper as the previous example.

```
public Product getProduct(int productId) {
    String sql = "SELECT * FROM Products WHERE productId=?";
    return (Product) getJdbcTemplate().queryForObject(sql,
                        new Object[] {new Integer(productId) },
                                        new ProductMappingQuery() );
}
```

queryForObject() takes 3 Parameters

- ✓The SQL statement
- ✓Parameters to be inserted
- ✓A RowMapper

98

Notes:

Here, *queryForObject()* is a method of the JdbcTemplate that retrieves a single object when you specify the SQL, any parameters to pass into the SQL (as an array of objects), and the same *ProductMappingQuery()* object as before.

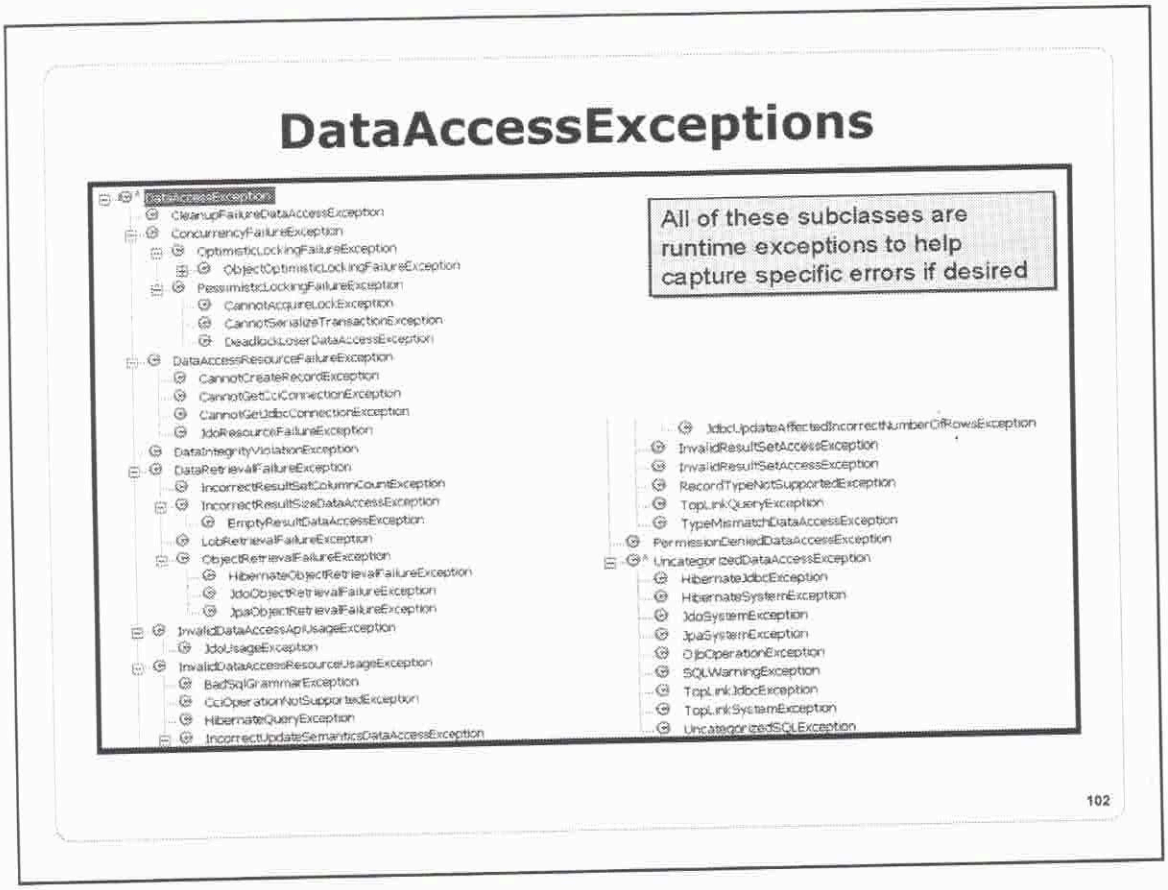
Spring JDBC Error Handling

- ***Spring JDBC*** turns *checked* exceptions into runtime exceptions
  - try-catch blocks are optional, not mandatory
  - Catch the runtime exceptions, if desired
  - Configure your own custom exception codes
- Spring JDBC's root exception is

  org.springframework.dao.DataAccessException
  - Since it is a runtime exception, handling it is only required for recovery and logging purposes

100

Notes:



Notes:

To trap database specific errors, you can configure them in a Spring XML definition file as follows:

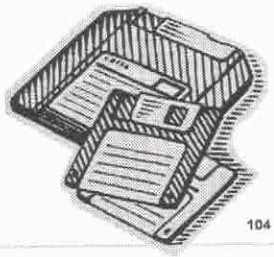
```
<bean id="DB2_9"
            class="org.springframework.jdbc.support.SQLErrorCodes">

<property name="badSqlGrammarCodes">
     <value>11,24,33</value>
 </property>
 <property name="dataIntegrityViolationCodes">
      <value>1,12,17,22</value>
 </property>
 <property name="customTranslations">
 <list>
   <bean
 class="org.springframework.jdbc.support.CustomSQLErrorCodesTranslation">
          <property name="errorCodes">
               <value>942</value></property>
          <property name="exceptionClass">
               <value>com.company.app.exceptions.MyCustomException</value>
          </property>
    </bean>
  </list>
</property>
</bean>
```

Updates

- Two ways to perform an update/insert/delete
  - Either call the *JdbcTemplate's update()* method
  - Create a class that extends *SQLUpdate*, and pre-compile the SQL in its constructor

Both ways are shown...



104

Notes:

These techniques are similar to the ones shown on the previous slides with respect to the retrieval of multiple or single rows of data. In this case, you are inserting rows.

2nd Technique: Updates via SQLUpdate

```
public class OrderInsert extends SqlUpdate {

    public OrderInsert(DataSource ds) {
        super(ds, "INSERT INTO Orders (OrderId,
                    CustomerId, TotalPrice) VALUES (?, ?, ?)");

        declareParameter(new SqlParameter(Types.INTEGER));
        declareParameter(new SqlParameter(Types.INTEGER));
        declareParameter(new SqlParameter(Types.DOUBLE));
        compile();
    }
    public void insert(Order order) {
      Object[] params = new Object[] {
                    new Integer(order.getOrderId()),
                    new Integer(order.getCustomerId()),
                    new Double(order.getTotalPrice())};

      update(params);
     }
}
```

The OrderManager invokes this method

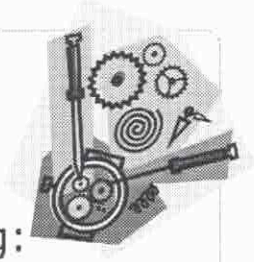
106

Notes:

```
public class OrderManagerImplementation  extends JdbcDaoSupport
implements OrderManager, OrderUpdater
{
    private DataSource dataSource;

    public void saveOrder(Order order)
    {
        new OrderInsert(getDataSource()).insert(order);
    }
}
```

Summary



- Spring simplifies JDBC work by using:
  - dependency injection
  - converting checked exceptions into runtime exceptions
  - handling much of the redundant JDBC work
- Spring offers new classes that simplify the work even further

Notes:

J2EE Integration Overview

- Patterns of Architecture: DAOs
- Integrating Spring and Hibernate
- JPA and Spring

110

Notes:

Wrapping Hibernate

```
public class EmployeeDAO
{
    private Session session;
    public void create(Employee e) throws DAOException {
        try {
            getSession();
            session.saveOrUpdate(e);
            session.getTransaction().commit();
        }
        catch(HibernateException ex) {
            session.getTransaction().rollback();
            ex.printStackTrace();
        }
        finally {
            if (session != null && session.isOpen()) {
                session.getTransaction().rollback();
                session.close();
            }
        }
    }
```

112

Notes:

Wrapping Hibernate ...

```
public void update(Employee e) throws DAOException
}
    try {
       getSession();
       session.saveOrUpdate(e);
       session.getTransaction().commit();
    }
    catch(HibernateException ex) {
       session.getTransaction().rollback();
       ex.printStackTrace();  // for development only
    }
    finally {
       if (session != null && session.isOpen()) {
            session.getTransaction().rollback();
            session.close();
       }
    }
}
```


Notes:

Wrapping Hibernate ...

```
    private Transaction getSession() {
            session = HibernateUtils.getSessionFactory().
                                        getCurrentSession();
            return session.beginTransaction();
            }
}
```

Client code to use the DAO:

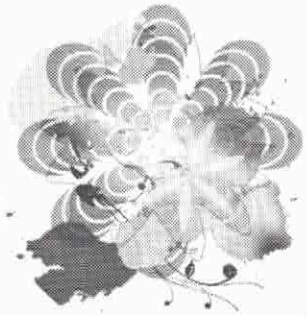
```
public static void main (String[] args) {
    Employee e = new Employee("000300", "Smiley");
    e.setFirstName("Jason");
    EmployeeDAO  dao = new EmployeeDAO();
    dao.create(e);
    Employee eNew = dao.find(e.getEmpNo());
}
```

116

Notes:

AbstractDAO

```
public abstract class AbstractDAO {
    private Session session;
    protected void create(Object o) throws DAOException {
        update(o);
    }
    protected void update(Object o) throws DAOException {
        try {
            getSession();session.saveOrUpdate(o);
            session.getTransaction().commit();
        }
        catch(HibernateException ex) {
            session.getTransaction().rollback();
        }
        finally {
            close();
        }
    }
```



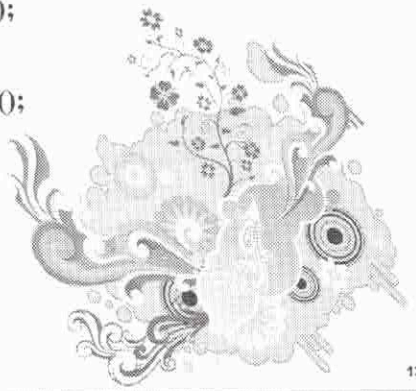
118

Notes:

AbstractDAO ...

```
protected Object find(Class classType, String id)
                                                    throws DAOException {
    Object o = null;

    try {
          getSession();
          o = session.load(classType, id);
          session.getTransaction().commit();
    }
      catch(HibernateException ex) {
          session.getTransaction().rollback();
      }
      finally {
          close();
      }
      return o;
}
```



120

Notes:

Improved EmployeeDAO

```
public class EmployeeDAO extends AbstractDAO {
    public void create(Employee e) throws DAOException {
        super.create(e);
    }
    public void delete(Employee e) throws DAOException {
        super.delete(e);
    }
    public void update(Employee e) throws DAOException {
        super.update(e);
    }
    public List findAll() throws DAOException {
        return super.findAll(Employee.class);
    }
    public Employee find(String id) throws DAOException {
        return (Employee) super.find(Employee.class, id);
    }
}
```

122

Notes:

Using Spring's HibernateTemplate

```
Employee e = new Employee("000350","Trayson");
e.setFirstNme("Anna");

HibernateTemplate ht = new
HibernateTemplate(HibernateUtils.getSessionFactory());

ht.saveOrUpdate(e);

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

124

Notes:

The HibernateTemplate serves as a wrapper for the Hibernate session.

Configuring the SessionFactory

beans.xml

```
<! -- continued from last slide -- >
<bean id="sessionFactory"
    class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">

 <property name="dataSource"><ref local="dataSource"></ref></property>

 <property name="mappingResources">
   <list>
     <value>com/company/springclass/beans/Employee.hbm.xml</value>
     <value>com/company/springclass/beans/Department.hbm.xml</value>
     <value>com/company/springclass/beans/Address.hbm.xml</value>
     <value>com/company/springclass/beans/Customer.hbm.xml</value>
     <value>com/company/springclass/beans/Order.hbm.xml</value>
     <value>com/company/springclass/beans/OrderDetails.hbm.xml</value>
     <value>com/company/springclass/beans/Product.hbm.xml</value>
   </list>
 </property>

          <! -- SessionFactory properties on next slide -- >

</bean>
```

126

Notes:

HibernateDaoSupport

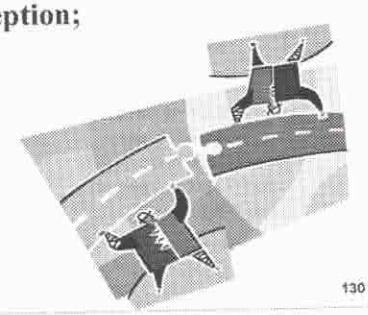
- Spring provides a wrapper for the **HibernateTemplate** called ***HibernateDaoSupport***
  - Just like JdbcDaoSupport
- Modify *AbstractDAO* to extend *HibernateDaoSupport*
  - Use this class to get the **HibernateTemplate** by calling ***getHibernateTemplate()***
  - This is very similar to what is done using Spring JDBC

128

Notes:

The DAO Interface

```
public interface DAO
{
    public void create(Object o) throws DAOException;
    public void update(Object o) throws DAOException;
    public void delete(Object o) throws DAOException;
    public Object find(Class c, String id) throws DAOException;
    public Object find(Class c, Long id) throws DAOException;
    public List findAll(Class c) throws DAOException;
}
```



130

Notes:

EmployeeDAO Configuration

- The *EmployeeDAO* is exactly as before
  - It must be configured in *beans.xml* to use it

```
<bean id="employeeBean"
                class="com.company.springclass.dao.EmployeeDAO">
    <property name="sessionFactory">
        <ref local="sessionFactory"/>
    </property>
</bean>
```

132

Notes:

Lab 4: Hibernate and Spring

- Working from the **SpringLab04** project, integrate Spring and Hibernate frameworks
- Follow the specific steps provided in ***instructions.html f***ound within the root of the project.

134

Notes:

JPA Integration ...

- Use Spring's *LocalContainerEntityManagerFactoryBean*

```
<bean id="entityManagerFactory" class=
        "org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
        <property name="dataSource" ref="myDataSource"/>
        <property name="loadTimeWeaver">
            <bean class=
            "org.springframework.instrument.classloading.InstrumentationLoadTimeWeaver"/>
        </property>
</bean>
```

136

Notes:

This last approach requires the most configuration, but affords the most flexibility, including the ability to provide an externally managed data source, if desired. Implementation using this technique can lead to additional configuration of the classloaders if Tomcat (or WebLogic, or several other app servers) are used.

JpaDaoSupport

- Provides a means for easily obtaining the *JpaTemplate*
- Here's how:
  1. Configure an *EntityManagerFactory* bean within Spring's configuration,
  2. Supply it to your persistence unit name
  3. Pass it into your DAO as a local reference
  4. Invoke ***getJpaTemplate()*** from within your DAO

138

Notes:

The JpaDaoSupport class is the same as the other Support classes. Your DAO should extend this class, making it very easy to manipulate an EntityManagerFactory and retrieve the JpaTemplate object.

Configuring the DAO with JPA

```
<bean id="employeeDAO" class="com.company.crm.dao.EmployeeDAO">
    <property name="entityManagerFactory">
        <ref local="myEntityManagerFactory"/>
    </property>
</bean>

<bean id="myEntityManagerFactory"
    class="org.springframework.orm.jpa.LocalEntityManagerFactoryBean">
        <property name="persistenceUnitName"
                                        value="SpringJPAExample"/>
</bean>
```

140

Notes:

The value, "SpringJPAExample" is the name of the persistence unit found in the META-INF directory of your src folder. This entity manager factory is then passed into the EmployeeDAO via dependency injection. The Employee DAO inherits from JpaDaoSupport and, therefore, inherits a *setEntityManagerFactory()* method. Nothing special needs to happen to your DAO class due to the assistance provided by the JpaDaoSupport class.

Lab 4b: JPA and Spring

- Working from the **SpringLab04** project, integrate Spring and the JPA using Hibernate as the underlying provider
- Follow the specific steps provided in ***instructions.html*** found within the root of the project
- Create a solution that finds customers by their id by making requests through the *FindService* service and the *CustomerDAO*

142

Notes:

Overview

- MVC Architecture Overview
- SpringMVC Overview
- Configuring web.xml
- Creating a Spring MVC config file
- Mapping Requests to Controllers
- Integrating the Service Layer

144

Notes:

What is MVC?

- A model-view-controller, or MVC, architecture is a design pattern that makes developing, maintaining, and testing applications easier.
- The MVC pattern dates back to the late 1970s
  - Originates from the Smalltalk world
- Today it is heavily implemented
  - Particularly in Java-based web applications

146

Notes:

Model-View-Controller Pattern ...

- The Controller
  - Represents the "traffic cop" in the application
  - Routes requests to specific tasks (actions)
    - Often consists of a centralized Servlet
    - Ensures proper view document is called
    - Usually has other helpers to assist in work delegation
- Model
  - Business Component Interaction
    - Consists of data components such as JavaBeans (POJOs)

148

Notes:

In a Java-based MVC architecture, the controller is not always a servlet. While this is true in many frameworks like Struts, JSF, and Spring MVC, some frameworks use other components.

Why Spring MVC?

- Spring MVC is a specific MVC implementation like (Struts 1.x/2.x, JSF, Ruby on Rails, Zend)
  - Provides nice integration with Spring beans
  - Loosens the rules components must follow within an MVC framework
  - Easy integration with Spring Web Flows 2

150

Notes:

When looking at MVC architectures, one might be led to say, "Why use Spring MVC?". Certainly Struts, JSF, or countless other web frameworks work nicely with Spring, so why use it? With Spring MVC, the bean configuration is integrated nicely into the solution. Also, controllers have more room for flexibility than do actions in a Struts world.

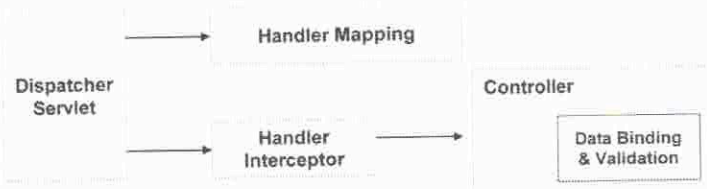
Although there are certainly differences, this course is not going to perform comparisons between the various web frameworks.

Consider this link comparing solutions:

http://static.raibledesigns.com/repository/presentations/ComparingJavaWebFrameworks-ApacheConUS2007.pdf

Spring MVC Controllers

- Controllers are similar to Struts Actions
- Controllers are implemented as singletons
  - This means they are shared by multiple requests
  - Should be created as thread-safe (stateless)



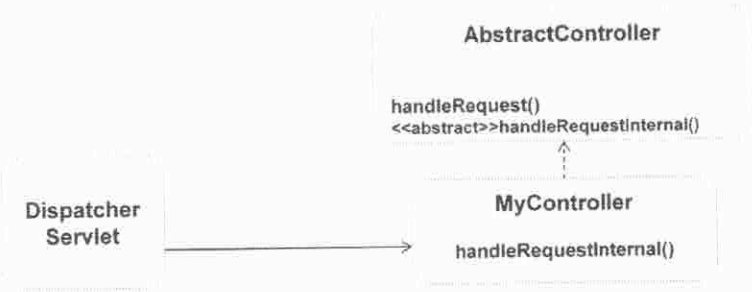
152

Notes:

The DispatcherServlet utilizes a configured HandlerMapping to determine which Controller to invoke. Before the Controller is invoked a HandlerInterceptor can be invoked (optionally if configured). HandlerInterceptors are similar to Filters in the Servlet API and can perform preprocessing work such as data filtering or authentication.

AbstractController

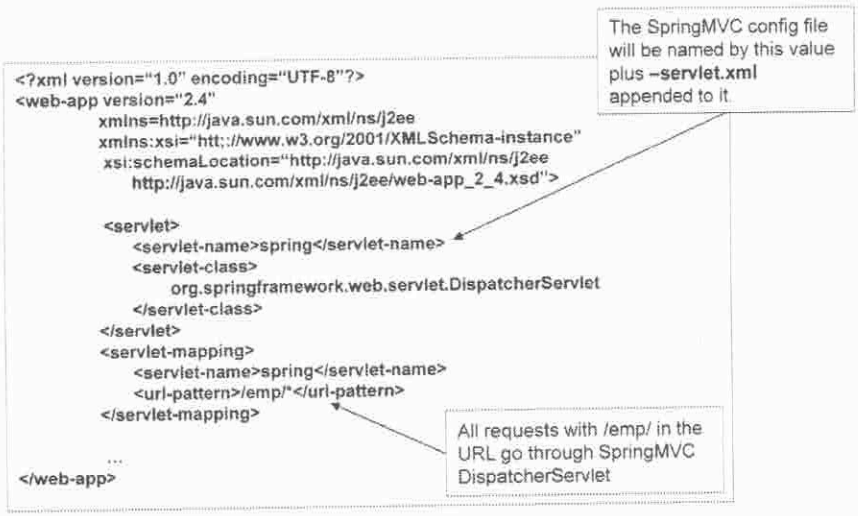
- AbstractController provides a handleRequest() method invoked by the DispatcherServlet
  - AbstractController in turn invokes handleRequestInternal() which is a method you provide (a form of the Template Method pattern)



154

Notes:

Configuring web.xml



156

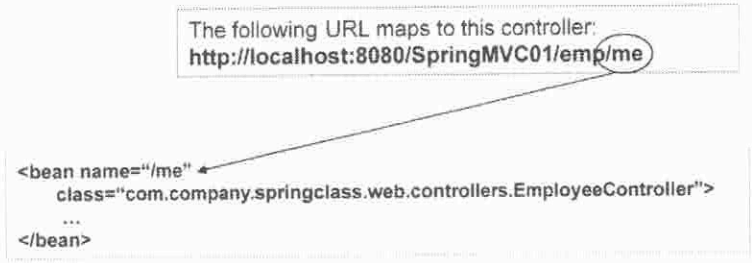
Notes:

Do not confuse the applicationContext.xml file with the SpringMVC config file. applicationContext.xml defines your business services, spring-servlet.xml defines your SpringMVC mappings.

Important: The name chosen for the servlet-name ('spring' above) becomes the name of your SpringMVC config file with –servlet.xml appended onto it. So we should expect to see a spring-servlet.xml file in the WEB-INF directory.

Mapping URLs to Controllers ...

- BeanNameUrlHandlerMapping
  - Uses the URL as a bean mapping name directly



158

Notes:

This technique is the default. Simply map the controller to the URL by providing a bean definition in which the name attribute represents the part of the URL that will uniquely identify which controller to invoke.

Mapping URLs to Controllers ...

- ControllerClassNameHandlerMapping
  - This technique uses the concept of "convention over configuration"
  - A url with /me will invoke MeController in the basePackage

```
<bean id="urlMapping"
    class="org.springframework.web.servlet.mvc.support.
                                        ControllerClassNameHandlerMapping">
        <property name="basePackage">
            <value="com.company.springclass.web.controllers"/>
</bean>
```

If Spring finds a controller called EmployeeController, this will map automatically to /employee

160

Notes:

There are some additional optional properties that may be specified with a ControllerClassNameHandlerMapping. Here are a few of those properties:

<property name="caseSensitive" value="true"/> (url mappings are case sensitive)

<property name="order" value="0"/>

<property name="pathPrefix" value="/"/>

spring-servlet.xml

```
<beans>
   <!– view definition –>
   <bean id="viewResolver"
         class="org.springframework.web.servlet.view.
                                          InternalResourceViewResolver">
      <property name="prefix" value="/"/>
      <property name="suffix" value=".jsp"/>
   </bean>

   <!– controller definitions –>
   <bean name="/findEmployee"
         class="com.company.springclass.web.controllers.
                                                EmployeeController">
   </bean>

</beans>
```

ViewResolvers are explained later

By default, SpringMVC is using a BeanNameUrlHandlerMapping which means it maps a URL to a class. Here, /findEmployee = EmployeeController

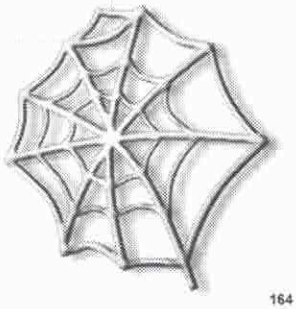
162

Notes:

The default HandlerMapping is a BeanNameUrlHandlerMapping. It matches the URL to the name in a bean definition. If a match is found, the specified Controller class is executed.

web.xml

```
<servlet>
    <servlet-name>spring</servlet-name>
    <servlet-class>
        org.springframework.web.servlet.DispatcherServlet
    </servlet-class>
    <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
    <servlet-name>spring</servlet-name>
    <url-pattern>/emp/*</url-pattern>
</servlet-mapping>
```



164

Notes:

The two important points here:

The spring config filename is determined here by using the format: [servlet-name]-spring.xml

The mapping determines which URLs will go through the Spring MVC framework.

spring-servlet.xml

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

   <!– controller definitions –>
   <bean name="/findEmployee"
         class="com.company.springclass.web.controllers.
                                          EmployeeController">

   </bean>
</beans>
```

URL maps to controller

166

Notes:

This config file is using the default handler mapping which maps URLs to Controllers.

page2.jsp

```
<html>
   <head>
      <title>SpringMVC01 Project Employee Result</title>

      <style type="text/css">ul  { list-style-type: none; }</style>

   </head>
   <body>
      <h1>Employee Info</h1>
      <ul>
      <li> FirstName:  $ { employee.firstName } </li>
      <li> LastName:  $ { employee.lastName } </li>
      <li> Dept Id: $ { employee.deptId } </li>
      </ul>
    </body>
</html>
```

JSP EL Syntax

168

Notes:

Because the ModelAndView added the empl object (under the key of employee) into the page instance, it can easily be retrieved using JSP expression language syntax.

Resolving Views

- A ViewResolver can be used to determine how to get from Controller to JSP (or whichever appropriate view)
- A common approach is to use a InternalResourceViewResolver
  - Configure it by defining a prefix and suffix for mappings
  - JSPs can be easily mapped too

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

The web app's root directory

170

Notes:

A common secure approach is to place JSPs in a directory within the WEB-INF directory. This can be done using a configuration similar to the following:

<property name="prefix" value="/WEB-INF/jsp/"/>

<property name="suffix" value=".jsp"/>