III. Spring JDBC

What Does Spring JDBC Offer?

- Spring manages *JDBC Resources*
  - Opening / closing connections
  - Reading JDBC Connection Properties
  - Iterates through ResultSets
  - Manage Transaction Details
  - Handles JDBC Exceptions

81

Notes:

Classic JDBC ...

```
public class CatalogManagerImplementation implements CatalogManager {
    public List getProducts() {
        List products = new ArrayList();

        Connection conn = null;-
        Statement stmt = null;
        ResultSet rs = null;

        Properties properties = new Properties();

        try {
            properties.load(new FileInputStream("jdbc.properties"));
        }
        catch (IOException e) {
            System.out.println("An error occurred while reading the properties file.");
            System.exit(42);
        }

        try {
            Class.forName(properties.getProperty("driverName"));
            conn = DriverManager.getConnection(properties.getProperty("url"),
            properties.getProperty("username"),
            properties.getProperty("password"));

            stmt = conn.createStatement();
            rs = stmt.executeQuery("SELECT * FROM Products");
```

Problems with Classic JDBC

- Lots of exception handling
- Repeated connection mgmt code
- Lots of work processing results

83

Notes:

```
    while (rs.next()) {
            int productId  = rs.getInt("ProductId");
            String name    = rs.getString("Name");
            String partNumber    = rs.getString("PartNumber");
            String description   = rs.getString("Description");
            double price   = rs.getDouble("Price");
            double cost    = rs.getDouble("Cost");
            int vendorId   = rs.getInt("VendorId");

            products.add(new Product(productId, name, description,
                    partNumber, cost, price, vendorId));
    }
}
catch (ClassNotFoundException e) {
        System.out.println("Error loading driver.");
}
catch (SQLException se)  {
        System.out.println("Error processing the SQL.");
}
finally {
        if (conn != null) {
            try {
                conn.close();
            }
            catch (SQLException e) {}
        }
}

return products;
}
}
```

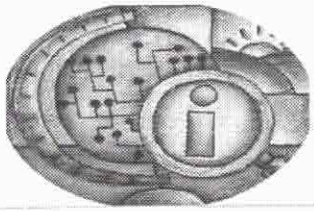
Configuring the DataSource

```
<bean id="dataSource"
    class="org.apache.commons.dbcp.BasicDataSource"
    destroy-method="close" >
        <property name="driverClassName"
                  value="com.ibm.db2.jcc.DB2Driver"/>
        <property name="url" value="jdbc:db2:SAMPLE"/>
        <property name="username" value="db2admin"/>
        <property name="password" value="db2admin"/>
</bean>
```

The Jakarta Commons DataSource has setters that Spring can inject values into it.

The catalog bean is the DAO. Spring injects the dataSource into it.

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



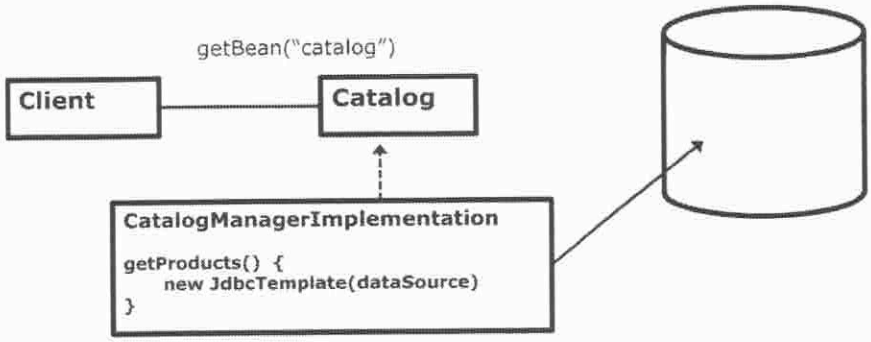
85

Notes:

Notice the bean definition for our CatalogManagerImplementation class. It tells Spring to inject the dataSource bean into it. This means our CatalogManagerImplementation class needs to have a setter called setDataSource(). Upon a casual glance at the code in the provided projects, you will notice that it doesn't! So, how does Spring inject the dataSource into this bean? The answer is coming in 2 slides, but a hint for now: the JdbcDaoSupport class.

Working with JdbcTemplate

- *JdbcTemplate* is a concrete class and can be instantiated directly
- It needs to know which data source will be used



87

Notes:

There are several ways to work with Spring's JdbcTemplate. One way is simply to instantiate it, passing the desired data source object to its constructor or calling its *setDataSource()* before using it. In the following example, the client obtains the Catalog service via a familiar *context.getBean()* call. It invokes the *getProducts()* method which, in turn, then obtains all the products from the database. It does this by using a newly instantiated JdbcTemplate object. See the next slide.

Getting Data from JdbcTemplate ...

```
public class CatalogManagerImplementation
                                    implements CatalogManager{

    private DataSource dataSource;

    public void setDataSource(DataSource dataSource) {
            this.dataSource = dataSource;
    }

    public List getProducts() {
            String sql = "SELECT * FROM springclass.products";
            return new JdbcTemplate(dataSource) .queryForList(sql);
    }
}
```

This is the complete solution, but how does the dataSource get set?

89

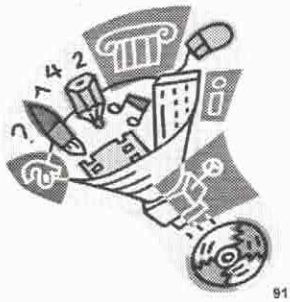
Notes:

The *getProducts()* method in this example returns a List of Map objects. Each row returned from the ResultSet is placed into a Map object. This may not be the easiest way to work with data, so we'll improve this solution in just a moment.

The JdbcTemplate object is passed to the dataSource object, but how does it get set? There is a setter method, but it is not used in this code. Where does it get invoked? See next slide...

Improving the Solution

- In the previous example, we had to add a DataSource property and instantiate the *JdbcTemplate* ourselves.
- By having *CatalogManagerImplementation* inherit from *JdbcDaoSupport*, Spring will take care of:
  - Injecting the *DataSource*
  - Instantiating the *JdbcTemplate*



91

Notes:

The JdbcDaoSupport class provides a nice way for your application to "hook" into the Spring JDBC classes and then take advantage of its IOC features.

Using JdbcDaoSupport

```
public class CatalogManagerImplementation
                    extends JdbcDaoSupport implements CatalogManager {
    public List getProducts () {
        String spl = "SELECT * FROM springclass.products";
        return getJdbcTemplate().queryForList(sql);
    }
}
```

No data source or setter is required now

Instantiating the JdbcTemplate object is not necessary now

93

Notes:

In this revised solution, only the CatalogManagerImplementation has changed. Notice it extends JdbcDaoSupport. This provides a convenient *getJdbcTemplate()* method and the data source is automatically injected by Spring and stored in the CatalogManagerImplementation because the parent class has a setter for it.

A Row Mapper

```
public class ProductMappingQuery extends MappingSqlQuery
                                        implements RowMapper {
    public ProductMappingQuery(DataSource ds) {
        super (ds, "SELECT * FROM Products");
        compile;
    }
    public Object mapRow(ResultSet rs, int rowNumber)
                                              throws SQLException {
        int productId          = rs.getInt("ProductId");
        String name            = rs.getString("Name");
        String partNumber      = rs.getString("PartNumber");
        String description     = rs.getString("Description");
        double price           = rs.getDouble("Price");
        double cost            = rs.getDouble("Cost");
        int vendorId           = rs.getInt("VendorId");

        return new Product(productId, name, description,
                                   partNumber, cost, price, vendorId);
    }
}
```

How to map 1 row

95

Notes:

You can define a class that should inherit from MappingSqlQuery. Override it's *mapRow()* method and define how an object should be mapped from an SQL row. Return one object from the *mapRow( )* method. When implemented, *mapRow()* is called by the Spring Framework for each record in the ResultSet, much like a callback. So, if there are ten records retrieved, *mapRow()* is invoked 10 times. Each time the current row in the ResultSet is advanced by Spring, a rowNumber is also passed to the method for counting purposes, if needed.

The Spring DAO

```
public class CatalogManagerImplementation extends JdbcDaoSupport
implements CatalogManager
{
    public List getProducts() {
        return new ProductMappingQuery(getDataSource()).execute();
    }
}
```

When the DAO extends JdbcDaoSupport, it inherits a set/getDataSource() method

execute() returns a List of objects (one object is returned from each call to mapRow()

97

Notes:

The ProductMappingQuery object overrides the *mapRow()* method and defines how 1 row from the ResultSet should be processed. Typically, this work is to extract the data and populate a newly instantiated business object.

execute() returns a java.util.List of business objects that can now be processed back in the client.

Spring JDBC Simplifies the Work

- No closing of Connections, Statements, ResultSets needed
  - Spring manages resources
    - Closing and releasing connections
- No checked exceptions requiring handling
  - How can you obtain Error messages?

99

Notes:

Spring JDBC Error Handling ...

```
public Product getProduct(int productId)
{
    String sql = "SELECT * FROM Products WHERE productId=?";
    Product p = null;

    try {

        p = (Product) getJdbcTemplate().queryForObject(sql,
                        new Object[]{new Integer(productId)},
                        new ProductMappingQuery());

    }
    catch (DataAccessException e) {
        logger.severe(e.getMessage());
    }

    return p;
}
```

101

Notes:

DataAccessException is a runtime exception. There are a number of subclasses to catch specific error types (see next slide).

Spring JDBC Error Msg Example

```
String sql = "SELECT * FROM Products WHERE prodId=?";
Product p = null;

try
{
    p = (Product) getJdbcTemplate().queryForObject(sql,
                                  new Object[] {new Integer(productId)},
                                  new ProductMappingQuery());
} catch (DataAccessException e)
}
    logger.severe(e.getMessage());
}
```

This should be productId

```
SEVERE: PreparedStatementCallback; bad SQL grammar [SELECT * FROM
Products WHERE prodId=?]; nested exception is
com.ibm.db2.jcc.c.SqlException: DB2 SQL error: SQLCODE: -206,
SQLSTATE: 42703, SQLERRMC: PRODID
```

103

Notes:

1st Technique: Updates using JdbcTemplate

```
public class OrderManagerImplementation extends JdbcDaoSupport
implements OrderManager, OrderUpdater {

    public void saveOrder(Order order) {

        String orderSQL = "INSERT INTO ORDERS (ORDERID,
                          CUSTOMERID, TOTALPRICE) VALUES (?, ?, ?)";
        getJdbcTemplate().update(orderSQL,
                          new Object[] {
                          new Integer(order.getOrderId()),
                          new Integer(order.getCustomerId()),
                          new Double(order.getTotalPrice())} );
    }
```

105

Notes:

This technique requires extending the JdbcDaoSupport class so that its *getJdbcTemplate()* method can be invoked.

Exercise 3: JdbcTemplate

- Using Spring JDBC techniques, create a solution that can *view a single product and all products* from the Products table and *update a product*
- Use the source files provided in the ***SpringLab03*** **Eclipse** project
- Work from the additional tasks provided in ***instructions.html***

107

Notes:

IV. DAOs, ORM Frameworks, and Spring

DAO Patterns

- *Data Access Objects* wrap repetitive database code hiding the ugly details
  - DAOs can be
    - Generic for an entire application
      - Ex: Class might be called **DAO**
    - Specific for each class
      - Ex: Class might be called **EmployeeDAO**
- DAOs provide basic CRUD operations...

111

Notes:

Wrapping Hibernate ...

```
public void delete(Employee e) throws DAOException
{
    try {
       getSession();
       session.delete(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();
       }
    }
}
```

113

Notes:

Wrapping Hibernate ...

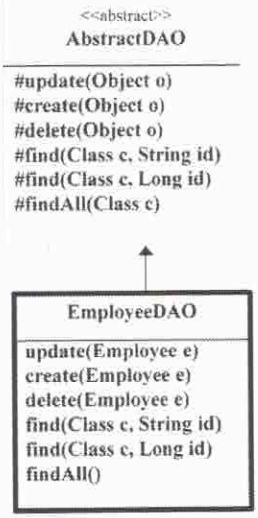
```
public Employee find(String id) throws DAOException
}
    Employee e = null
      try {
          getSession();
          e = (Employee) session.load(Employee.class, id);
          session.getTransaction().commit();
    }
    catch(HibernateException ex) {
      session.getTransaction().rollback();
      ex.printStackTrace();      // for development only
    }
    finally {
      if (session != null && session.isOpen()) {
          session.getTransaction().rollback();
          session.close();
      }
    }
    return e;
}
```

115

Notes:

Improving the DAO

- Reduce the amount of repetitive code by using the a ***Layered Supertype*** *Pattern*
  - *Patterns of Enterprise Application Architecture*
    - Martin Fowler
  - Code is placed in an abstract parent class
  - Concrete child class calls parent methods



117

Notes:

AbstractDAO ...

```
protected List findAll(Class classType) throws DAOException {
    List list = null;

    try {
        getSession();
        list = session.createQuery("FROM" +
                                   classType.getName()).list();
        session.getTransaction().commit();
    }
    catch(HibernateException ex) {
        session.getTransaction().rollback();

    }
    finally {
        close();
    }

    return list;
}
```


Notes:

AbstractDAO ...

```
private void close() {
    if  (session != null && session.isOpen()) {
        session.getTransaction().rollback();
        session.close();
    }
}

private Transaction getSession() {
    session =
          HibernateUtils.getSessionFactory().
          getCurrentSession();

    return session.beginTransaction();
}
```

121

Notes:

Spring *Meets* Hibernate

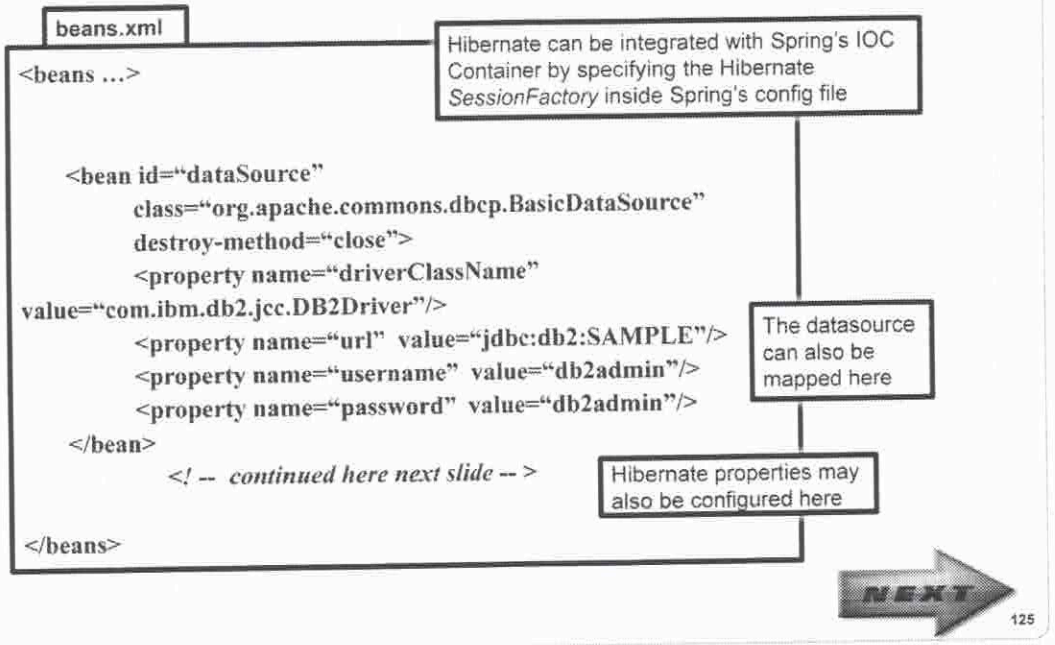
- Spring JDBC provides a *JDBCTemplate* class
  - It took care of the busy work of handling exceptions, opening and closing connections
- Spring provides a similar template for Hibernate called **HibernateTemplate**
- *HibernateTemplate* takes care of:
  - Obtaining the session
  - Handling Exceptions
  - Closing the session
  - Flushing changes
  - Beginning and committing the transaction

123

Notes:

HibernateTemplate is in the **org.springframework.orm.hibernate3** package.

Configuring Hibernate within Spring's IOC Container

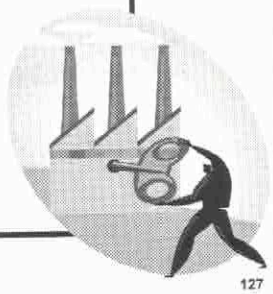


Notes:

Configuring the SessionFactory ...

beans.xml

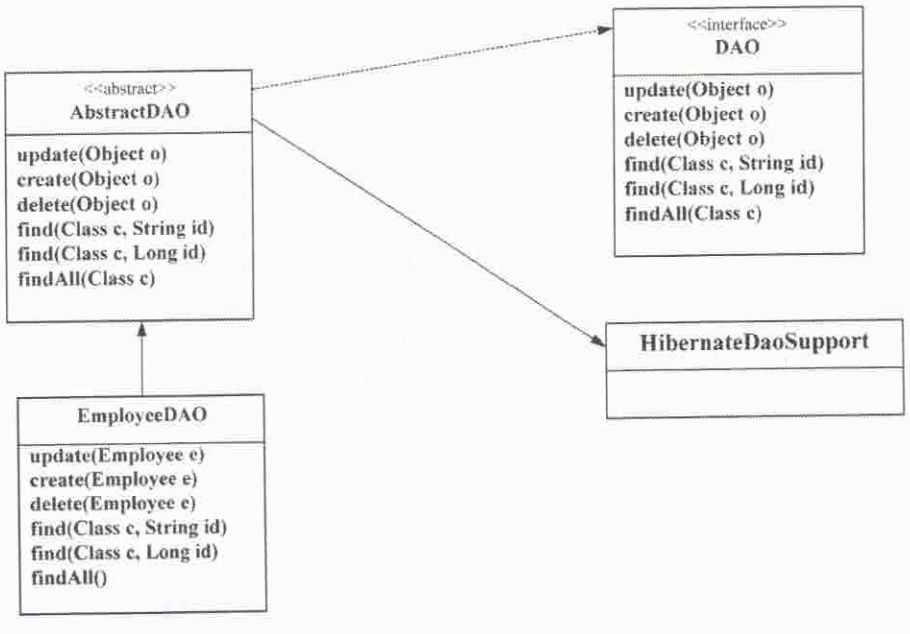
```
<! -- SessionFactory properties -- >

<property name="hibernateProperties">
    <props>
        <prop key="hibernate.dialect">
            org.hibernate.dialect.DB2Dialect
        </prop>
        <prop key="current_session_context_class">thread</prop>
        <prop key="cache.provider_class">
            org.hibernate.cache.NoCacheProvider
        </prop>
        <prop key="show_sql">true</prop>
    </props>
</property>

</bean>
```



127

Notes:

A New and Improved DAO



129

Notes:

The New AbstractDAO

```
public abstract class AbstractDAO extends HibernateDaoSupport
                                                    implements DAO {
    public void create(Object o) throws DAOException {
        update(o);
    }
    public void update(Object o) throws DAOException {
        getHibernateTemplate().saveOrUpdate(o);
    }
    public List findAll(Class classType) throws DAOException {
        return getHibernateTemplate().
                            find("FROM " + classType.getName());
    }
    public Object find(Class classType, String id)
                                        throws DAOException {
        return getHibernateTemplate().load(classType, id);
    }
    // other methods shown in footnotes
}
```

131

Notes:

```
public void delete(Object o) throws DAOException {
    getHibernateTemplate().delete(o);

}
```

Client Code

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

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

EmployeeDAO dao =
                (EmployeeDAO) context.getBean("employeeBean");

dao.create(e);
Employee emp = dao.find(e.getEmpNo());
```

Create a transient object

Obtain the DAO through Spring's IOC Container

Work with the DAO

133

Notes:

JPA Integration

- Much like Hibernate, *JPA* can be integrated into a Spring-based application in several different ways:
  - *Use Spring's LocalEntityManagerFactoryBean*

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

  - *Obtain the EntityManager via JNDI lookup*

```
<jee:jndi-lookup id=" entityMgrFactory "
                jndi-name="persistence/myPersistenceUnit"/>
```

135

Notes:

The advantages to the first technique is that it is easy and fast to configure. However, disadvantages with this approach include the fact that an external data source can't be specified this way. The second technique is appropriate for managed environments such as JavaEE app servers that provide JNDI services.

For most efforts, one of these two options will fit most needs.

JpaTemplate

- As we saw with Spring JDBC and Hibernate, a *JpaTemplate* class provides most of the functionality for interacting with JPA
- Similar methods to the JPA exist within the JpaTemplate:

find()
remove()
persist()
merge()

137

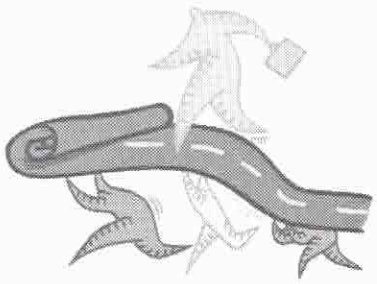
Notes:

The find(), remove(), persist(), and merge() methods found within the JpaTemplate are analogous to the entitymanager's methods with the same name.

Using JpaDaoSupport

- Here's how your DAO might look...

```
public class EmployeeDAO extends JpaDaoSupport {
    public abstract Object find(Object id) {
        return (Employee) getJpaTemplate().
            find(Employee.class, new Integer(id.toString()));
    }
    public void insert(Object o) {
        getJpaTemplate().persist(o);
    }
    public void update(Object o) {
        getJpaTemplate().persist(o);
    }
    public void remove(Object o) {
        getJpaTemplate().remove(o);
    }
}
```



139

Notes:

Other than the different methods invoked, this DAO is very similar to that of the Hibernate DAO that makes use of the HibernateDaoSupport class from Spring.

Summary

- Spring and Hibernate can work together
- SessionFactory, Mapping Resources, DataSources can all be configured through Spring and injected into the Hibernate framework
- Using Spring's Hibernate support classes, the work associated with mapping objects becomes even easier

141

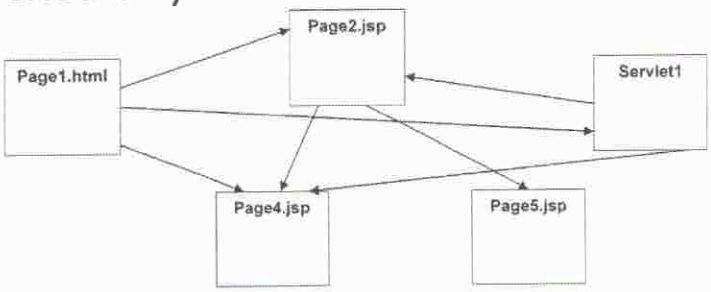
Notes:

V. Spring MVC

143

Traditional Web Architectures

- In the mid-1990s, many web applications lacked any coherent architecture



- Difficult to maintain as solutions become more complex

145

Notes:

Early web architectures consisted primarily of static content and were created largely by graphic designers and HTML designers.

Today, many web-based systems are more complex, attempting to perform many more tasks such as authentication, database interaction, and communication with enterprise components. Most web applications now utilize dynamic components such as JSPs, Servlets, and many other resources.

Model-View-Controller Pattern

- MVC attempts to separate out
  - Presentation-based tasks from
  - Business domain tasks
- The View
  - Consists of the application's presentation components:
    - HTML, JSPs, CSSs, client-side scripts, etc.

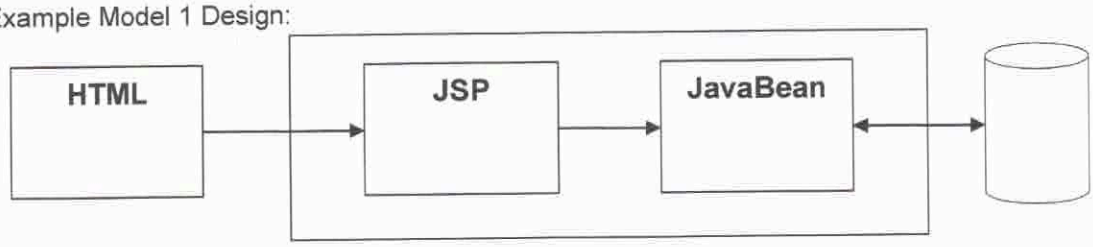
147

Notes:

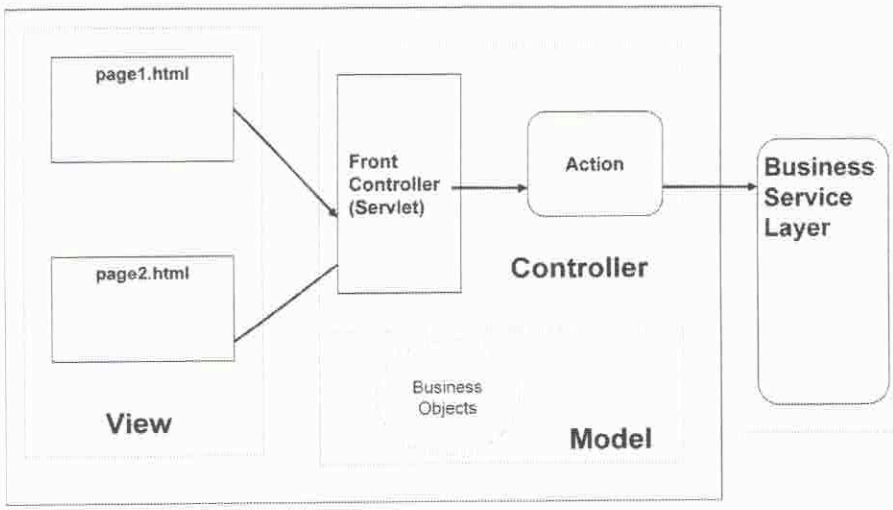
Larger, more dynamic applications become more difficult to maintain without a well-designed structure in place. The demand for better design techniques led the planners of the JSP specification to present two design options:

-Model 1 Architecture (JSP can handle request processing, i.e. a JSP submits to a JSP)
-Model 2 Architecture (Model-View-Controller approach)

Example Model 1 Design:

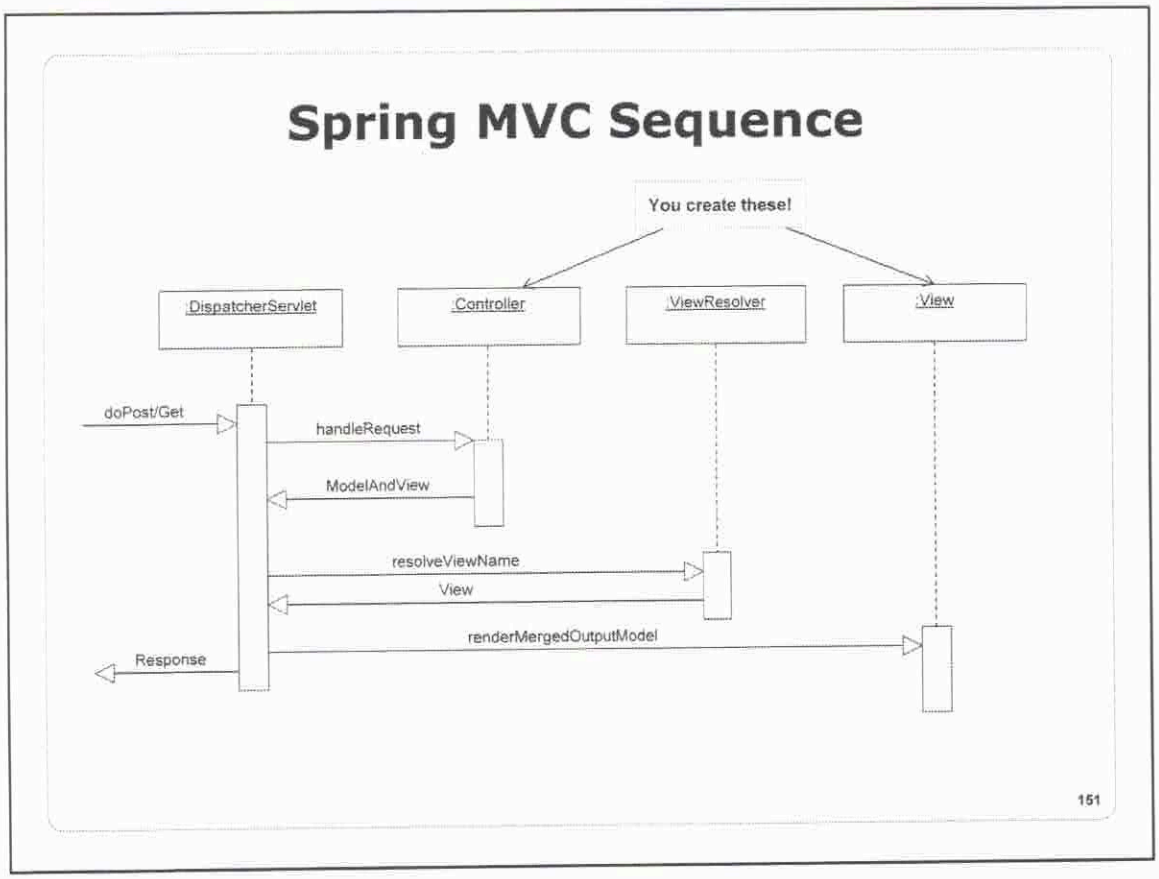


Classic MVC Representation



149

Notes:



Notes:

A Spring MVC application starts and ends a request cycle at the DispatcherServlet. It is configured in the web.xml. Controllers are defined in another config file and managed by Spring. The ViewResolver helps make the decisions on how to traverse from a controller to a view. Finally, the view, usually a JSP, is rendered.

Creating Controllers

- Controllers need to implement the *Controller* interface
  - This interface defines a handlerRequest() method
- Extend *AbstractController* instead to get additional basic HTTP features
  - In this case, provide a handleRequestInternal() method

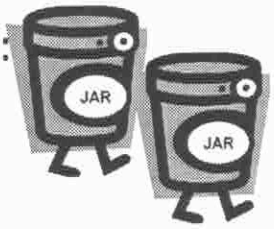
153

Notes:

It is simple to create a Controller by implementing the Controller interface, but if additional features, such as cache-control HTTP headers, access to web objects like request, response, etc. is desired, then it is easiest to extend the AbstractController class instead. Depending on the approach taken will determine which method to provide in your controller. The example uses the AbstractController.

Establishing the Project

- Create a web project
- Add JARs to the lib directory:
  - Add the Spring JARs
  - Jakarta Commons JARs
  - JEE JARs (jstl.jar, standard.jar as needed)



155

Notes:

Mapping URLs to Controllers

- Spring provided multiple ways to map URLs (requests) to Spring MVC Controllers
  - BeanNameUrlHandlerMapping (default)
  - SimpleUrlHandlerMapping
  - ControllerClassNameHandlerMapping
  - Annotations

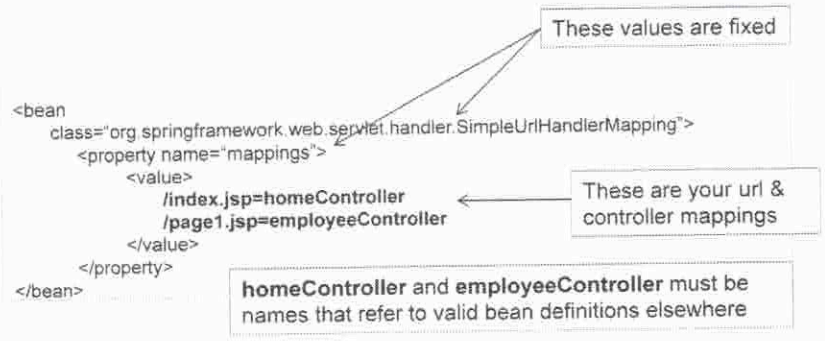
157

Notes:

Each of these techniques are useful and easy to work with. The mapping definitions are placed into the SpringMVC config file, which in our case is called **spring-servlet.xml**.

Mapping URLs to Controllers ...

- SimpleUrlHandlerMapping
  - Uses a map-style approach defining url=controller name pairs



159

Notes:

This handler mapping is great when configuring many beans with a one-to-one mapping. A bean id is not needed when using SimpleUrlHandlerMapping. The disadvantage to using it is that it does not really save you much configuration.

It does have a nice feature, the use of * for wildcard mappings. For example /me* will allow /memon or /medep to map to the same controller.

The default is to map within the same servlet mapping, however if the alwaysUseFullPath property is set, then the url must include the full mapping (ex: /emp/index.jsp).

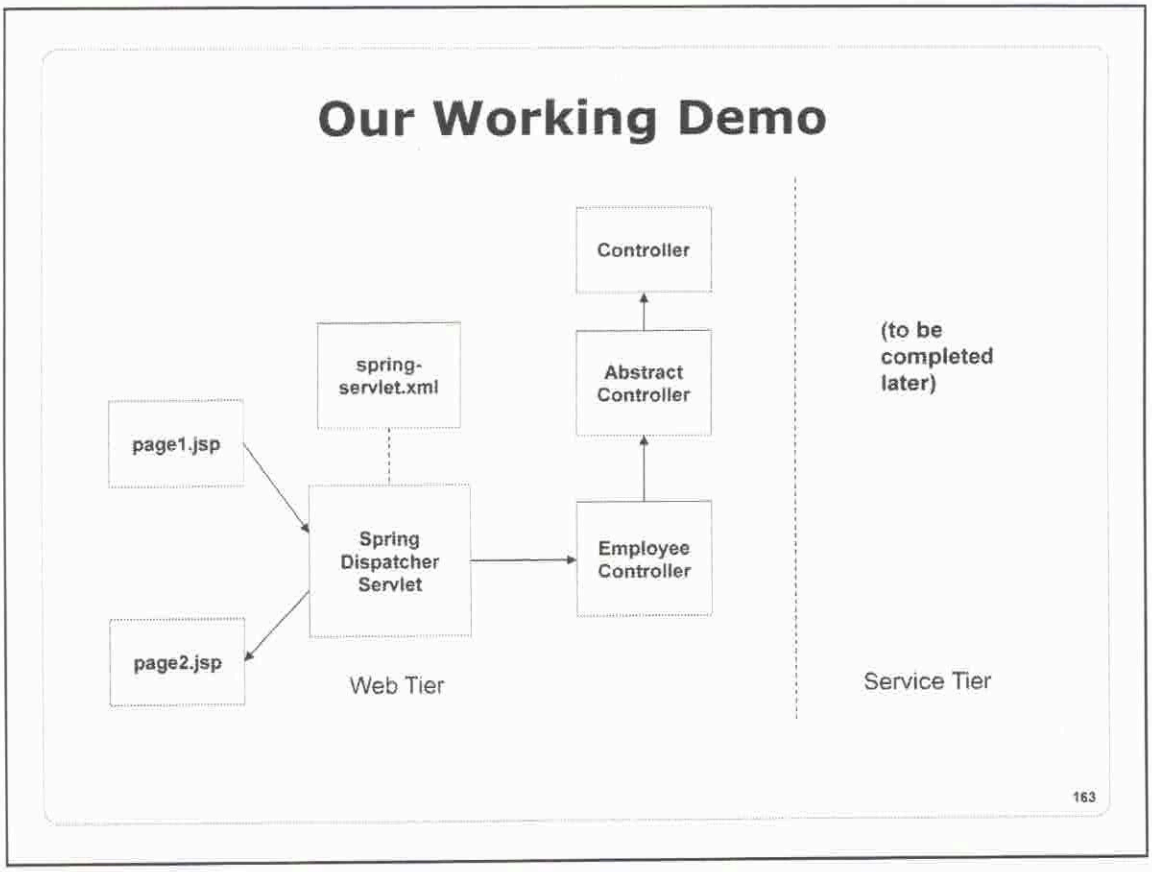
Mapping URLs to Controllers ...

- Annotation-based mapping (Spring 2.5 or later)
  - Uses the @Controller annotation to define controllers
  - Autodetected through classpath scanning
  - Usually used in conjunction with @RequestMapping annotation

161

Notes:

Annotation-based controllers, in conjunction with the use of the RequestMapping annotation is now the preferred technique for handling the mappings between requests and controllers for Spring 3.0.



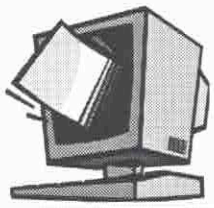
Notes:

Our example starts with a simple request response cycle. A form submits data from page1.jsp to the DispatcherServlet. The DispatcherServlet hands the request to the EmployeeController. The EmployeeController at this stage simply instantiates an Employee object and returns it to page2.jsp for display.

page1.jsp

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

   <body>
      <h1>Employee Retrieval</h1>
      <h4>Enter an employee to find (100 – 103)</h4>
      <form action="/SpringMVC01/emp/findEmployee">
         Employee ID: <input type="text" name="id" /><br/>
         <input type="submit" value="Find" ></input>
      </form>
    </body>
</html>
```

URL mapping will be /findEmployee



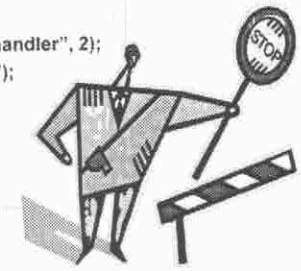
165

Notes:

This page begins the request-response cycle.

EmployeeController

```
public class EmployeeController extends AbstractController {

    protected ModelAndView handleRequestInternal (
                    HttpServletRequest request,
                    HttpServletResponse response) throws Exception  {
        int id = 1;
        try  {
            id = Integer.parseInt(request.getParameter("id"));
        }
        catch (NumberFormatException e)  { }

        Employee empl = new Employee (id, "Steve", "Chandler", 2);
        ModelAndView mv = new ModelAndView("page2");
        mv.addObject ("employee", empl);
        return mv;
    }
}
```



167

Notes:

This controller identifies its parent as AbstractController. Notice the overridden handleRequestInternal() method. In this method, it is our job to access any services, then define what model components to send to the view and which view to go to. The ModelandView object performs this for us.

You can use addObject() to add model objects into the page or use the ModelAndView constructor to do this in one step.

ModelAndView

- ModelAndView is a wrapper for any model components to be setn to a view
  - The view can be defined in the constructor
  - A ViewResolver will determine how to map a logical view (like "page2") to and actual view (like page2.jsp)
  - Objects added to the ModelAndView will be injected by spring into the view page instance

169

Notes: