Spring

The Spring Framework for J2EE

Dan Hayes

Chariot Solutions

March 22, 2005

CHARIOT SOLUTIONS

Agenda

- Introduction to the Spring Framework
- Inversion of Control and AOP* Concepts
- The Spring Bean Container
- Spring in the Business Tier
- Spring in the Web Tier
- Spring Related Projects

* Aspect Oriented Programming

INTRODUCTION TO THE SPRING FRAMEWORK

Introduction to Spring

Spring is a lightweight inversion of control and aspect oriented container framework

Spring makes developing J2EE application easier and more fun!!!

Introduced by Rod Johnson in “J2EE Design & Development”

Introduction to Spring

Spring helps J2EE developers by:

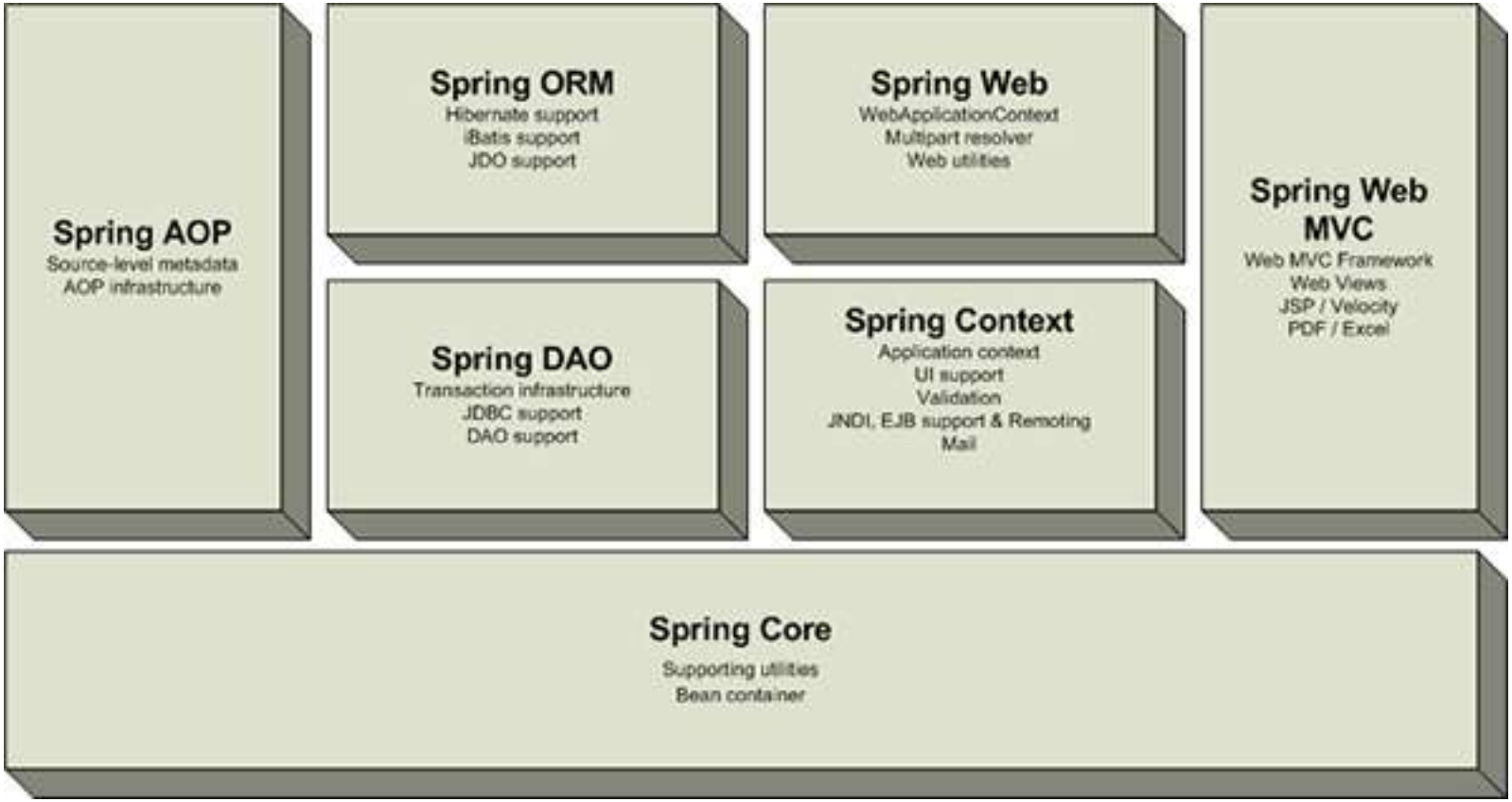
- Offering a lightweight JavaBean *container* that eliminates the need to write repetitive plumbing code such as lookups
- Providing an inversion of control framework that allows bean dependencies to be automatically resolved upon object instantiation
- Allowing cross cutting concerns such as transaction management to be woven into beans as “aspects” rather than becoming the concern of the business object
- Offering layers of abstraction on top of popular existing technologies such as JDBC and Hibernate that ease their use and organize configuration management

Introduction to Spring

…while adhering to certain principals:

- Your application code should not depend on Spring API's
- Spring should not compete with good existing solutions, but should foster integration
- Writing *testable* code is critical and the container should help - not interfere - with this objective
- Spring should be a pleasure to use

Introduction to Spring



Introduction to Spring

Architecturally speaking:

- Spring can be employed in a number of situations
  - Web app only applications
  - Applications that involve EJB's or other Services
  - Applications that access one or more resources (databases)
- However, Spring is emerging as the leading framework for "lightweight" J2EE application development
  - Do we really need heavyweight services, such as EJB, provided by traditional J2EE application servers?
  - Clustering of co-located (single JVM) applications

INVERSION OF CONTROL AND AOP CONCEPTS

Inversion of Control

- What is it?
  - A way of sorting out dependencies between objects and automatically “injecting” references to collaborating objects on demand
- Who determines how the objects should be injected?
  - The IoC framework, usually via XML configuration files
- Benefits
  - Removes the responsibility of finding or creating dependent objects and moves it into configuration
  - Reduces coupling between implementation objects and encourages interface based design
  - Allows an application to be re-configured outside of code
  - Can encourage writing testable components

Inversion of Control

Traditional way of obtaining references….

```
…
private AccountService accountService = null;

public void execute(HttpServletRequest req, ….) throws Exception {
  Account account = createAccount(req);
  AccountService service = getAccountService();
  service.updateAccount(account);
}

private AccountService getAccountService() throws … {
  if (accountService == null) {
    Context ctx = new InitialContext();
    Context env = (Context) ctx.lookup("java:comp/env");
    Object obj = env.lookup("ejb/AccountServiceHome");
    AccountServiceHome home = (AccountServiceHome)
        PortableRemoteObject.narrow(env, AccountService.class);
    accountService = home.create();
  }
  return accountService;
}
…
```

Inversion of Control

With Spring IoC the container will handle the "injection" of an appropriate implementation ….

```
private AccountService accountService = null;

public void setAccountService(AccountService accountService) {
  this.accountService = accountService;
}

public void execute(HttpServletRequest req, ….) throws Exception {
  Account account = createAccount(req);
  accountService.updateAccount(account);
}

…
```

We will see how Spring does this in the next section

AOP Concepts

- Applications must be concerned with things like:
  - Transaction management
  - Logging
  - Security
- Do these responsibilities belong to the implementation classes?
  - Should our AccountService class be responsible for transaction management, logging, or security?
- These concerns are often referred to as “cross-cutting” concerns

AOP Concepts

Problems with traditional approach ….

- Code that implements system-wide concerns is duplicated across multiple components
- The business components become littered with code that isn't central to it's responsibilities

AOP Concepts

Spring AOP allows us to:

- Separate these concerns and define them as an “advice”
  - Before Advice
  - After Advice
  - Around Advice
  - Throws Advice
- Declaratively “weave” them into the application based on certain criteria (“pointcuts”).
- Proxies then *intercept* relevant methods and silently introduce the advice

AOP Concepts

Types of AOP:

- Static AOP
  - Aspects are typically introduced in the byte code during the build process or via a custom class loader at runtime
  - AspectJ (byte code manipulation at compile time)
  - JBoss 4, AspectWerkz (Class loader approach)
- Dynamic AOP
  - Create “proxies” for all advised objects
  - Slightly poorer performance
  - Easier to configure
  - **Spring**

THE SPRING BEAN CONTAINER

Spring Bean Container

Bean container is core to the Spring framework:

- Uses IoC to manage components that make up an application
- Components are expressed as regular Java Beans
- Container manages the relationships between beans and is responsible for configuring them
- Manages the lifecycle of the beans

Spring Bean Container

Types of Bean Containers:

- Bean Factory
  - Provides basic support for dependency injection
  - Configuration and lifecycle management
- Application Context
  - Builds on Bean Factory and adds services for:
    - Resolving messages from properties files for internationalization,
    - Loading generic resources
    - Publishing events

Spring Bean Container

Loading a Bean Factory:

```
public interface Greeting {
  String getGreeting();
}

public class WelcomeGreeting implements Greeting {
  private String message;

  public void setMessage(String message) {
    this.message = message;
  }

  public String getGreeting() {
    return message;
  }
}
```

Spring Bean Container

Loading a Bean Factory (cont):

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
        "http://www.springframework.org/dtd/spring-beans.dtd">

<beans>
  <bean id="greeting" class="WelcomeGreeting">
    <property name="message">
      <value>Welcome</value>
    </property>
  </bean>
</beans>
```

… saved in a file called foo.xml:

Spring Bean Container

Loading a Bean Factory (cont):

```
public static void main(String[] args) {

  BeanFactory factory = new XmlBeanFactory(
                                  new FileInputStream("foo.xml"));

  Greeting greeting = (Greeting) factory.getBean("greeting");

  System.out.println(greeting.getGreeting());

}
```

Output:
Welcome

Spring Bean Container

Features of Managed Beans:

- Default to Singleton's
- Properties can be set via dependency injection
  - References to other managed beans
  - Strings
  - Primitive types
  - Collections (lists, set, map, props)
  - "Inner Beans"
- "Auto-wiring"
- Parameters may be externalized to property files

SPRING IN THE BUSINESS TIER

Spring in the Business Tier

Spring supports the business tier by providing:

- A layer of abstraction for persisting data via one or more of the following technologies
  - JDBC
  - Hibernate, OJB
  - JDO
  - iBatis
- A robust infrastructure for declarative transaction management that supports local transactions as well as global transactions via JTA
- A simplifying layer for “remoting” technologies including EJB, RMI, Hession/Burlap, and Web Services
- Useful support for JNDI, JMS, email, task scheduling

Spring Persistence Support

The Spring DAO module includes:

- A technology agnostic data access API that helps isolate and streamline the way data is served by the business tier
- A consistent and rich exception hierarchy that is smart enough to map technology specific exceptions to generalized exceptions
- A series of template and wrapper classes for working with JDBC, Hibernate, JDO, etc.

Spring Persistence Support

Traditional JDBC coding techniques:

```
public void updateCustomer(Customer customer) {
  Connection conn = null;
  PreparedStatement ps = null;
  try {
    conn = getConnection();
    ps = conn.prepareStatement("update customer set " +
      "firstName = ?, lastName = ?, …);
    ps.setString(1, customer.getFirstName());
    ps.setString(2, customer.getLastName());
    ps.executeUpdate();
  } catch SQLException e) {
    log.error(e);
  } finally {
    try { if (ps != null) ps.close(); }
    catch (SQLException e) {log.error(e);}
    try {if (conn != null) conn.close();}
    catch (SQLException e) {log.error(e);}
  }
}
private Connection getConnection() {
  … more plumbing code here
}
```

Spring Persistence Support

Using Spring JDBC template:

```
public void updateCustomer(Customer customer) {
  String sql = "update customer set " +
    "firstName = ?, lastName = ?, …);
  Object[] params = new Object[] {
                          customer.getFirstName(),
                          customer.getLastName(),
                          … };
  int[] types = new int[] {
                          Types.VARCHAR,
                          Types.VARCHAR,
                          … };
  jdbcTemplate.update(sql, params, types);
}
```

jdbcTemplate can be injected by container…

Spring Persistence Support

Operations can also be modeled as objects:

```
public class UpdateCustomer extends SqlUpdate {
  public UpdateCustomer(DataSource ds) {
    setDataSource(ds);
    setSql("update customer set… values (?,?..)");
    declareParameter(new SqlParameter(Types.VARCHAR));
    declareParameter(new SqlParameter(Types.VARCHAR));
    compile();
  }

  public int update(Customer customer) {
    Object[] params = new Object[] {
                  customer.getFirstName(),
                  customer.getLastName()
                  };
    return update(params);
  }
}
```

Spring Persistence Support

Using the UpdateCustomer object :

```
public class JdbcCustomerDao extends JdbcDaoSupport
                                        implements CustomerDao {
  private UpdateCustomer updateCustomer;

  protected void initDao() throws Exception {
    super.initDao();
    updateCustomer = new UpdateCustomer(getDataSource());
  }

  public void updateCustomer(Customer customer) {
    updateCustomer.update(customer);
  }

}
```

We may even choose to make the UpdateCustomer object an inner class!

Spring Persistence Support

Hibernate Support :

- Spring provides a SessionFactory bean that simplifies configuration and session management for business objects
- A HibernateTemplate class is provided to reduce need for boilerplate code
- HibernateDaoSupport class that can be extended for further abstraction
- HibernateException management and mapping
- Seamlessly plugs into Spring transaction framework

Spring Persistence Support

Hibernate configuration example :

```
…
<beans>
  <bean id="sessionFactory" class="org.springframework.orm.
                                hibernate.LocalSessionFactoryBean">
    <property name="dataSource">
      <ref bean="dataSource"/>
    </property>
    <property name="hibernateProperties">
      <props>
        <prop key="hibernate.dialect">
          net.sf.hibernate.dialect.MySQLDialect
        </prop>
      </props>
    </property>
    <property name="mappingResources">
      <list><value>Customer.hbm.xml</value></list>
    </property>
  </bean>
continued…
```

Spring Persistence Support

Hibernate configuration example (cont) :

```
…
  <bean id="customerDao" class="HibernateCustomerDao">
    <property name="sessionFactory">
      <ref bean="sessionFactory"/>
    </property>
  </bean>

  <bean id="dataSource" class="org.springframework.jndi.
                          JndiObjectFactoryBean">
    <property name="jndiName">
      <value>java:comp/env/jdbc/myDataSource</value>
    </property>
  </bean>

</beans>
```

Spring Persistence Support

The CustomerDao is simply:

```
public class HibernateCustomerDao extends HibernateDaoSupport
                                  implements CustomerDao {
  public void updateCustomer(Customer customer) {
    getHibernateTemplate().update(customer);
  }
}
```

Similar integration exists for iBatis, JDO, and OJB

Spring Transaction Support

Spring support for transactions :

- Support for both programmatic and declarative transaction management
- Local transactions delegated by Spring to the underlying data source transaction manager
- When multiple resources are involved, Spring delegates to the JTA transaction manager obtained via JNDI
- Only minor configuration changes are required when switching between local and JTA

Spring Transaction Support

Programmatic Transaction Management :

- Not as popular as declarative approach
- When fine-grained control is required
  - Similar to manually controlling JTA UserTransactions
- Available via the Spring TransactionTemplate class
  - Reference typically passed to the business object via DI
  - Transaction template holds a property of type PlatformTransactionManager
  - PlatformTransactionManager can be an implementation of Hibernate, JDO, JDBC, or JTA transaction manager

Spring Transaction Support

Declarative Transaction Management :

- Uses AOP to wrap calls to transactional objects with code to begin and commit transactions
- Transaction propagation behavior
  - Mandatory, Never, Not Supported, Required, Requires New, Support, Nested
  - Similar to EJB style behaviors
- Also supports isolation levels
  - Default, Read Uncommitted, Read Committed, Repeatable Read, Serializable

Spring Transaction Support

Declaring transaction attributes:

Propagation Behavior

Is transaction Read Only?
[optional]

```
PROPAGATION, ISOLATION, readOnly, -Exceptions, +Exception
```

Isolation Level
[optional]

Rollback Rules
[optional]

Attributes are declared in bean definition file

Spring Transaction Support

Declarative Transaction Example:

```
…
<beans>
  <bean id="customerService" class="org.springframework.transaction.
                                interceptor.TransactionProxyFactoryBean">
    <property name="transactionManager">
      <ref bean="transactionManager"/>
    </property>
    <property name="target">
      <ref bean="customerServiceTarget"/>
    </property>
    <property name="transactionAttributes">
      <props>
        <prop key="get*">PROPAGATION_REQUIRED, readOnly</prop>
        <prop key="store*">PROPAGATION_REQUIRED</prop>
      </props>
    </property>
  <bean>
```

continued

Spring Transaction Support

Declarative Transaction Example (cont):

```
…
  <bean id="customerServiceTarget" class="CustomerServiceImpl">
    <property name="customerDao">
      <ref bean="customerDao"/>
    </property>
  </bean>

  <bean id="transactionManager" class="org.springframework.orm.
                                hibernate.HibernateTransactionManager">
    <property name="sessionFactory">
      <ref bean="sessionFactory"/>
    </property>
  </bean>

  <bean id="customerDao" class="HibernateCustomerDao">
    <property name="sessionFactory">
      <ref bean="sessionFactory"/>
    </property>
  </bean>
  …
<beans>
```

Spring Remoting

Spring provides support for remoting:

- RMI
  - Provides a RmiProxyFactoryBean that can be configured to connect to an RMI service
  - Provides a RmiServiceExporter that can be configured to expose any Spring managed bean as an RMI service
- Caucho (Hessian/Burlap)
  - Provides a Hession(Burlap)ProxyFactoryBean and ServiceExporter to connect to and expose Caucho remote services respectively
  - Also requires configuration of web application context and mapping of appropriate servlet url
- Spring Http Invoker
  - Similar to Caucho except using non-proprietary object serialization techniques

Spring Remoting

Spring Remoting (cont):

- EJB
  - Spring allows EJB's to be declared as beans within the Spring configuration file making it possible to inject references to them in other beans
  - Spring provides the ability to write EJB's that wrap Spring configured Java Beans
- Web Services via JAX-RPC
  - Client side support via the JaxRpcPortProxyFactoryBean
  - ServletEndpointSupport class allows services that sit behind a servlet to access the Spring Application Context

Spring Remoting

Accessing EJB's in Spring Apps:

- LocalStatelessSessionProxyFactoryBean
  - Proxy for local SLSB
    - Finds and caches the EJB Home upon first access
    - Intercepts method calls and invokes corresponding method on ejb
- SimpleRemoteStatelessSessionProxyFactoryBean
  - Proxy for remote SLSB
    - Will also catch Remote Exceptions and re-throw as unchecked exceptions
- Stateful session beans are NOT proxied
  - Simplified lookup via Spring JndiObjectFactoryBean

Spring Remoting

Looking up remote SLSB using Spring:

```
…
<bean id="customerService" class="org.springframework.ejb.
        access.SimpleRemoteStatelessSessionProxyFactoryBean"
        lazy-init="true">
  <property name="jndiName">
    <value>customerService</value>
  </property>
  <property name="businessInterface">
    <!-- fully qualified name of interface class -->
    <value>CustomerService</value>
  </property>
</bean>
…
<!-- in some web tier bean definition file… -->
<bean id="someWebTierController" class="…">
  <property name="customerService">
    <ref bean="customerService"/>
  </property>
</bean>
…
```

Spring Remoting

Building EJB's with help from Spring:

- Strategy based on "Business Interface" design pattern
  - Create a super interface that defines business methods
  - Let local/remote interface extends this interface
  - Bean class implements the super (business) interface
- Spring provides a set of abstract base classes
  - For SLSB, SFSB, and MDB
  - *Provides access to a Spring Bean Factory from within EJB!*
    - Must place EJB specific bean definition xml file(s) in ejb-jar
    - Declare name as environment entry in ejb-jar

Spring Remoting

CustomerService SLSB using Spring:

```
public class CustomerServiceEJB extends AbstractStatelessSessionBean
                                implements CustomerService {

  private static final String BEAN_NAME = “customerService”;
  private CustomerService customerService;

  public void storeCustomer(Customer customer) {
    customerService.storeCustomer(customer);
  }

  protected void onEjbCreate() throws CreateException {
    customerService = (CustomerService)
                                getBeanFactory().getBean(BEAN_NAME);
  }

}
```

Spring J2EE

Other Enterprise Features:

- JNDI Lookup
  - Moves verbose lookup code into the framework
- Sending Email
  - Template classes that support COS mail (Hunter/O'Reilly) and JavaMail
- Event Scheduling
  - Support JDK Timer
  - OpenSymphony Quartz events

SPRING IN THE WEB TIER

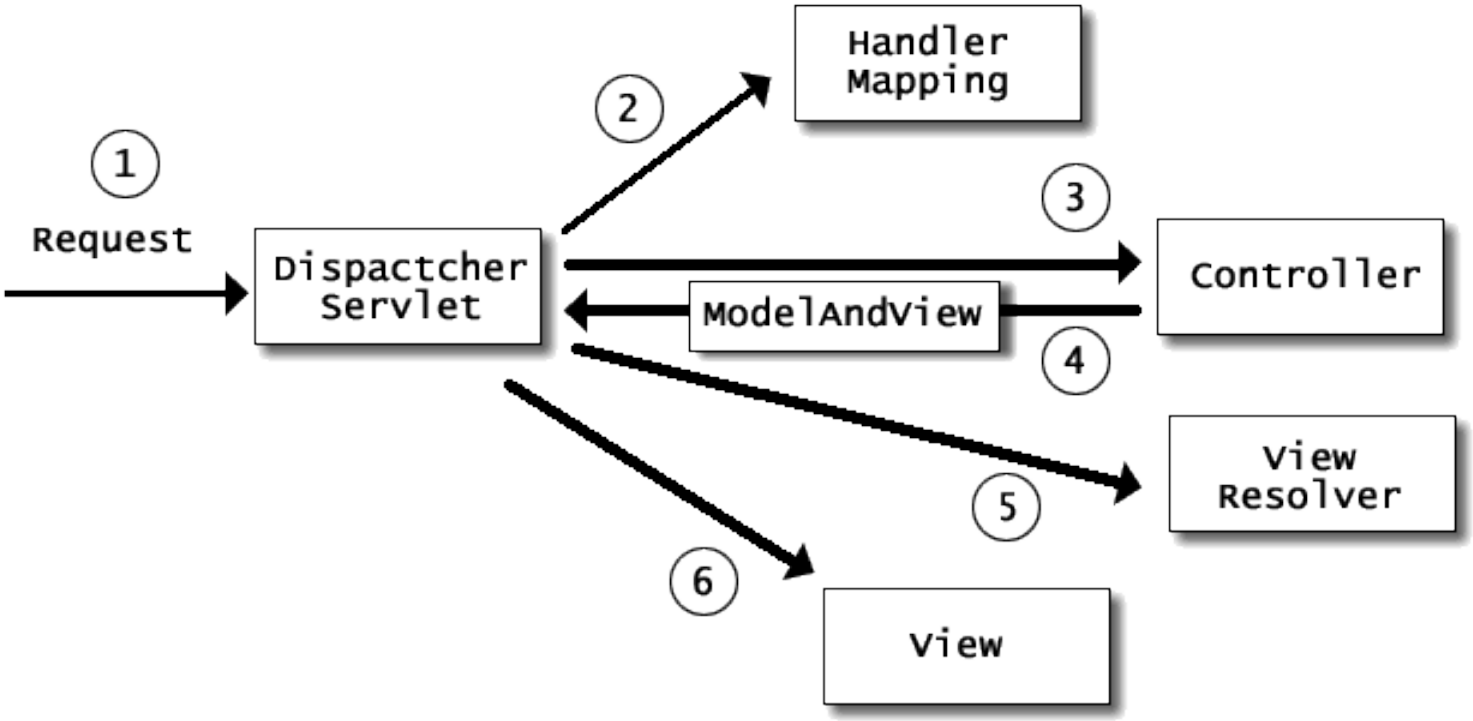
Spring in the Web Tier

Spring offers rich support in the Web Tier:

- Spring MVC
  - Spring's own implementation of the Model 2 MVC architecture
  - Flexible request mapping and handling
  - Full forms support
  - Supports multiple view technologies
  - JSP Tag Library
- Support for Existing Web Frameworks
  - Struts
  - Tapestry
  - JavaServerFaces
  - WebWork

Spring in the Web Tier

Spring MVC Lifecycle:



Spring in the Web Tier

Setting up a typical Spring MVC:

- Configure web.xml to map requests to Spring Front Controller
  - Similar to Struts setup
- Configure web.xml to have Spring read additional bean definition files, if necessary (*[servlet name]*-servlet.xml loaded automatically)
  - Spring supports either 2.3 (listeners) or 2.2 (startup servlet)
  - Allows multiple bean configuration files to be loaded into single Application Context
- Write Controller classes and configure them in xml file
  - Must map the controller to one or more URLs via an implementation of a HandlerMapping object
- Configure one or more view resolvers to tie controllers to views (JSP, Velocity template, etc.)
- Write the JSPs or other view artifacts to render the user interface

Spring in the Web Tier

Controllers in Spring MVC:

```
…
public class ListCustomersController implements Controller {
  private CustomerService customerService;

  public void setCustomerService(CustomerService customerService) {
    this.customerService = customerService;
  }

  public ModelAndView handleRequest(HttpServletRequest req,
                                    HttpServletResponse res) throws Exception
{
    return new ModelAndView("customerList", "customers",
                            customerService.getCustomers());
  }

}
```

ModelAndView object is simply a combination of a named view and a Map of objects that are introduced into the request by the dispatcher

Spring in the Web Tier

Controllers in Spring:

- Interface based
  - Do not have to extend any base classes (as in Struts)
- Have option of extending helpful base classes
  - Multi-Action Controllers
  - Command Controllers
    - Dynamic binding of request parameters to POJO (no ActionForms!)
    - Use PropertyEditor features of JavaBeans to map complex types
  - Form Controllers
    - Hooks into cycle for overriding binding, validation, and inserting reference data
    - Validation (including support for Commons Validation)
    - Wizard style controller

Spring in the Web Tier

Resolving the Url via a HandlerMapping :

```
…
<beans>
  <bean id="simpleUrlMapping" class="org.springframework.web.servlet.
                                 handler.SimpleUrlHandlerMapping">
    <property name="mappings">
      <props>
        <prop key="/admin/listCustomers.htm">
                 listCustomersController</prop>
      </props>
    </property>
  </bean>

  <bean id="listCustomersController" class="ListCustomersController">
    <property name="customerService">
      <ref bean="customerService"/>
    </property>
  </bean>
  …
</beans>
```

Spring in the Web Tier

Resolving the view names in Spring:

```
…
<beans>
  <bean id="viewResolver"
class="org.springframework.web.servlet.
                          view.ResourceBundleViewResolver">
    <property name="basename">
      <value>views</value>
    </property>
  </bean>
  …
</beans>
```

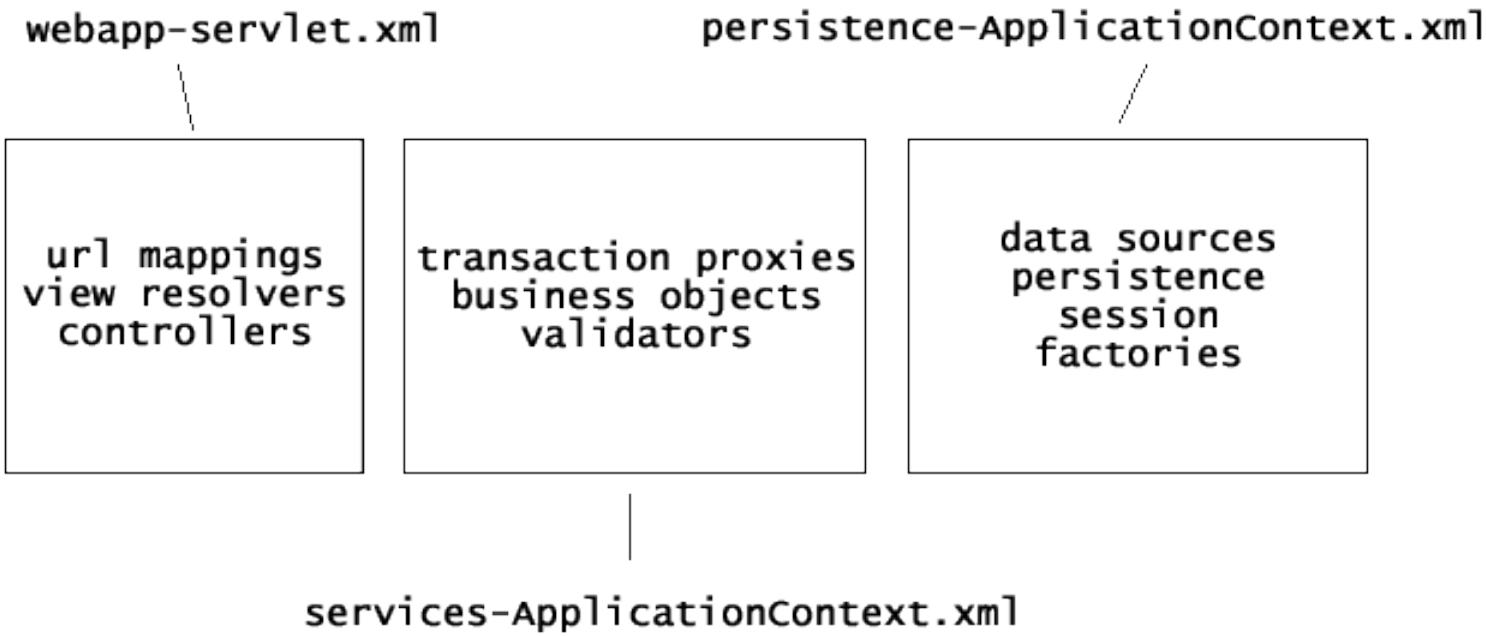
```
… in views.properties

customerList.class=com.springframework.web.servlet.view.JstlVie
w
customerList.url=/WEB-INF/jsp/customerList.jsp
```

The view class can be any implementation of a View object!

Spring includes hooks (via abstract base classes) to create your own view classes for rendering PDF and Excel using iText and POI libraries

Spring in the Web Tier



SPRING RELATED PROJECTS

Spring Related Projects

- Spring Rich Client
  - Develop Swing-based application using Spring concepts
- Spring IDE
  - Eclipse Plug-in for managing bean definition files and visualizing bean relationships
- Acegi Security
  - Security framework based upon the Spring framework
- Sandbox
  - Scripting support for bean dependencies (BeanShell, Groovy)
  - J2SE 5.0 annotations meta-data support
  - Spring JMX for exposing beans as JMX managed resources

Questions?

- Dan Hayes
  - Chariot Solutions
  - http://www.chariotsolutions.com
  - dhayes@chariotsolutions.com
- Spring Framework
  - http://www.springframework.org
  - http://forum.springframework.org
  - http://opensource.atlassian.com/confluence/spring