Building JavaServer Faces Applications with Spring and Hibernate

Kito Mann
Author of JSF in Action
Virtua, Inc
www.virtua.com

Chris Richardson
Author of POJOs in Action
Chris Richardson Consulting, Inc
www.chrisrichardson.net

TS-7082
What You Will Learn…

Why you should use JavaServer Faces technology, Spring, and Hibernate together and how to do it
About Kito Mann

- Author, *JavaServer Faces in Action*
- Trainer, consultant, architect, mentor
- Internationally recognized speaker
  - JavaOne℠ Conference, JavaZone, TSS Symposium, Javapolis, NFJS, etc.
- Founder, JSF Central
  - http://www.jsfcentral.com
- Java Community Process℠ (JCPSM) Member
  - JavaServer™ Faces 1.2 platform, JavaServer Pages 2.1 (JSP™) software, Design-Time API for JavaBeans™ architecture, Design-Time Metadata for JavaServer Faces Components, WebBeans, etc.
- Experience with Java™ platform since its release in 1995
- Web development since 1993
About Chris Richardson

• Grew up in England
• Live in Oakland, CA
• Developing software for 21 years
  • OO development since 1986
  • Java platform since 1996
  • Java Platform, Enterprise Edition (Java EE) since 1999
• Author of POJOs in Action
• Speaker at JavaOne Conference, JavaPolis, NFJS, JUGs, ….
• Chair of the eBIG Java SIG in Oakland (www.ebig.org)
• Run a consulting and training company that helps organizations build better software faster
Agenda

Using JavaServer Faces technology for the UI
Building a POJO backend
Using Spring in the business tier
Using Hibernate for persistence
Integrating Spring and JavaServer Faces technology
Agenda

Using JavaServer Faces technology for the UI
Building a POJO backend
Using Spring in the business tier
Using Hibernate for persistence
Integrating Spring and JavaServer Faces technology
JavaServer Faces Technology Overview

• Standard web user interface (UI) framework for Java platform
  • JavaServer Faces 1.0 platform: Standardized through Java Community Process (JCP) in 2004 (JSR 127)
  • JavaServer Faces 1.2 platform: Standardized through JCP in 2006 (Java Specification Request (JSR) 252)
    • Part of Java EE 5.0 platform

• Specification consists of:
  • Server side UI component and event model
  • Set of basic UI components
  • Basic MVC-style application infrastructure
JavaServer Faces Technology Overview

• Can automatically synchronize UI components with application objects
• Includes basic Dependency Injection container
• Extensive tool support
  • Sun, Oracle, IBM, BEA, Exadel, Borland, JetBrains, Genuitec, others
• Enables RAD-style approach to Java platform web development
• Built on top of Servlet API
• Works with JSP framework, but does not require it
JavaServer Faces Technology Overview

• Standard UI component model enables a third-party component marketplace
  • Grids, Trees, Menus, Sliders, Panels, Charts, Popup Windows, Calendars, etc.
  • Open source and commercial vendors
  • Often have integrated AJAX support
JavaServer Faces Technology vs. Struts

<table>
<thead>
<tr>
<th>Heavy abstraction</th>
<th>Transparent AJAX support</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Automatic markup generation (i.e., HTML, WML)</td>
</tr>
<tr>
<td></td>
<td>Declarative integration of user interface with business objects (both read and write)</td>
</tr>
<tr>
<td></td>
<td>Stateful user interface component model (running on the server)</td>
</tr>
<tr>
<td></td>
<td>Server-side handling of user interface events</td>
</tr>
<tr>
<td></td>
<td>Type conversion</td>
</tr>
<tr>
<td></td>
<td>Navigation</td>
</tr>
<tr>
<td></td>
<td>Form handling and validation</td>
</tr>
<tr>
<td></td>
<td>Enhanced localization</td>
</tr>
<tr>
<td></td>
<td>Layer separation (Model 2)</td>
</tr>
<tr>
<td></td>
<td>Resource management</td>
</tr>
<tr>
<td></td>
<td>Extensible templating mechanism</td>
</tr>
<tr>
<td></td>
<td>Integration with Java, session management, lifecycle management, basic error handling, security, deployment and packaging, J2EE integration, etc.</td>
</tr>
<tr>
<td></td>
<td>HTTP request and response handling</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Little abstraction</th>
<th>JavaServer Faces</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Struts 1.x</td>
</tr>
<tr>
<td></td>
<td>JavaServer Pages</td>
</tr>
<tr>
<td></td>
<td>Servlet API</td>
</tr>
<tr>
<td></td>
<td>Web server</td>
</tr>
</tbody>
</table>
public String add()
{
    Project project = getVisit().getCurrentProject();
    getProjectCoordinator().add(project);
    return "success";
}

<navigation-rule>
    <from-view-id>create.jsp</from-view-id>
    <navigation-case>
        <from-outcome>failure</from-outcome>
        <to-view-id>create.jsp</to-view-id>
    </navigation-case>
    <navigation-case>
        <from-outcome>success</from-outcome>
        <to-view-id>show_all.jsp</to-view-id>
    </navigation-case>
</navigation-rule>
The JavaServer Faces Technology Expression Language

• Can reference managed bean properties and methods

```java
#{updateProjectBean.comments}

Managed bean property

#{createProjectBean.add}

Managed bean method
```
The JavaServer Faces Technology | Expression Language

• Expression evaluation is pluggable

```java
#{updateProjectBean.comments}
```

VariableResolver PropertyResolver
(ELResolver evaluates both in JSF 1.2)

• Can either replace or decorate the default functionality
• Key integration point
DEMO

JavaServer Faces Platform UI Layer
Agenda

Using JavaServer Faces technology for the UI

**Building a POJO backend**

Using Spring in the business tier

Using Hibernate for persistence

Integrating Spring and JavaServer Faces technology
Avoid the Smart UI Anti-pattern

• Managed beans could do it all:
  • Implement the presentation logic
  • Implement the business rules
  • Access the database

• This might work for tiny application

• For real world applications you need to have a layered architecture
  • Improved modularity and reuse
  • Simplifies development
  • Simplifies testing
Use a Layered Architecture

Presentation Tier
- Managed Bean
- Managed Bean
- ...

Business Tier
- Service
- Service
- ...
- Entity
- Entity
- Entity

Persistence Tier
- DAO
- DAO
- ...

Transaction management
- Security
- Application assembly

Database Access
- SQL: JDBC, iBATIS
- OR/M: Hibernate, JPA

Spring or EJB
Separating Concerns in the Backend

• Layers are essential because they separate some concerns, e.g. presentation and business logic

• But within the business tier there are concerns that are not easily separated
  • Transactions
  • Security
  • Persistence
  • Other: logging, auditing, etc.

• These are cross cutting concerns
  • Span multiple application components
  • Can’t be solved by traditional modularization mechanisms such as layers or base classes
  • You must implement them by sprinkling code throughout the application
Traditional Architecture = Tangled Concerns

Application

Module A  Module B  Module C

Business logic  Persistence
Transactions  Security
**POJO = Plain Old Java Object**

- Java objects that don’t implement any special interfaces or (perhaps) call infrastructure APIs
- Coined by Martin Fowler, Rebecca Parsons, and Josh MacKenzie to make them sound just as exciting as JavaBeans, Enterprise JavaBeans™ technology
- Simple idea with surprising benefits
POJO Application Architecture

- Application
- Module A
- Module B
- Module C

- POJOs
- ORM
- Hibernate
- AOP-based security
- AOP-based transactions
- Spring
Agenda

Using JavaServer Faces technology for the UI
Building a POJO backend

**Using Spring in the business tier**

Using Hibernate for persistence
Integrating Spring and JavaServer Faces technology
Overview of Spring

• What is Spring?
  • Framework for simplifying Java EE platform application development
  • Rich feature set including dependency injection, AOP, ORM support, a web framework, …

• Key Spring features:
  • Dependency injection
  • AOP for transaction management, security and application-specific cross cutting concerns
  • Classes for simplifying data access
Spring Lightweight Container

- Lightweight container = sophisticated factory for creating objects
- Spring bean = object created and managed by Spring
- You write metadata (e.g. XML) or code that specifies how to:
  - Instantiate Spring beans
  - Initialize them using dependency injection
- Separates component instantiation and assembly from the components themselves
public class ProjectCoordinatorImpl ...

public ProjectCoordinatorImpl(
    ProjectRepository projectRepository, ...
)
{
    this.projectRepository = projectRepository;
    ...
}

public class HibernateProjectRepository implements ProjectRepository {
    ...
}

Sprin Code Example

<bean id="projectCoordinator"
    class="ProjectCoordinatorImpl">
    <constructor-arg ref="projectRepository"/>
    ...
</bean>

<bean id="projectRepository"
    class="HibernateProjectRepository">
    ...
</bean>
Spring AOP

- AOP enables the modular implementation of crosscutting concerns
- Spring AOP = simple, effective AOP implementation
- Lightweight container can wrap objects with proxies
- Proxy executes extra code before/after/instead-of original method

Spring uses proxies for:
- transaction management
- security
- tracing
- …
Spring Transaction Management

1. call changeStatus()
2. call changeStatus()
3. begin transaction
4. begin transaction
5. call invoke()
6. changeStatus() returns
7. commit transaction
8. invoke() returns
9. invoke() returns
10. changeStatus() returns

Transaction management API
(JDBC, Hibernate, JDO, JTA, ...)

UpdateProject Bean
AOP Proxy
Transaction Interceptor
Project Coordinator
Transaction Manager
Spring 2 Transaction Management

```xml
<bean id="projectCoordinator"
     class="ProjectCoordinatorImpl">
  ...
</bean>

<beans>
  <aop:config>
    <aop:advisor
      pointcut="execution(* *..*Coordinator..*)"
      advice-ref="txAdvice"/>
  </aop:config>

  <bean id="transactionManager"
       class="HibernateTransactionManager">
    ...
  </bean>

  <tx:advice id="txAdvice">
    <tx:attributes>
      <tx:method name="*"/>
    </tx:attributes>
  </tx:advice>

</beans>
```
Handling Custom Crosscutting Concerns

• Examples of application-specific crosscutting concerns
  • Auditing—recording user actions in a database
  • Automatically retrying failed transactions

• The traditional approach = sprinkle code throughout the application
  • Auditing—logic in every business method
  • Transaction retry—loop/try/catch around every call

• It simple but there are important drawbacks
  • Duplication of code
  • Business logic does several things ⇒ more complex
  • Easy to forget ⇒ insecure/fragile application
Example Transaction Retry Aspect

```java
public class TransactionRetryAspect {
    protected int maxRetries = 3;

    public Object retryTransaction(ProceedingJoinPoint jp)
        throws Throwable {
        int retries = 0;
        while (true)
            try {
                return jp.proceed();
            } catch (ConcurrencyFailureException e) {
                if (retries++ > maxRetries)
                    throw e;
                else continue;
            }
    }
}
```
Bean and Aspect Definitions

```xml
<bean id="transactionRetryPOJOAspect"
     class="net.chrisrichardson.aspects.retry.TransactionRetryPOJOAspect">
    <property name="maxRetries" value="4" />
</bean>

<aop:config>

    <aop:pointcut id="serviceMethod"
        expression="execution(public * net.chrisrichardson..*Coordinator.*(..))" />

    <aop:aspect id="txnRetryAspect" ref="transactionRetryAspect">
        <aop:around method="retryTransaction" pointcut-ref="serviceMethod" />
    </aop:aspect>

</aop:config>
```
DEMO

Spring Service Layer
Agenda

Using JavaServer Faces technology for the UI
Building a POJO backend
Using Spring in the business tier
**Using Hibernate for persistence**
Integrating Spring and JavaServer Faces technology
POJO Persistence

• Using an object/relational framework:
  • Metadata maps domain model to the database schema
  • Application code written in terms of objects
  • ORM framework generates SQL statements

• Java Persistence API (JPA)
  • Standardized OR/M

• Hibernate
  • Very popular open-source project
  • It’s a superset of Java Persistence API
O/RM Example

class Project {
    private int id;
    private String name;
    ...
}

public class HibernateProjectRepository ... {
    public void add(Project project) {
        getHibernateTemplate().save(project);
    }

    public Project get(int projectId) {
        return (Project) getHibernateTemplate().get(Project.class, projectId);
    }
}
Cool OR/M Framework Features

- Provides (mostly) transparent persistence
  - Objects are unaware that they are persistent
  - Minimal constraints on classes
  - They are POJOs
- Supports navigation between objects
  - Application navigates relationships
  - ORM framework loads objects behind the scenes
- Tracks changes to objects
  - Detects which objects have changed
  - Automatically updates the database
- Manages object identity
  - Only one copy of an object per PK
  - Maintains consistency
O/R Mapping Framework Benefits

• Improved productivity:
  • High-level object-oriented API
  • Less Java code to write
  • No SQL to write

• Improved performance
  • Sophisticated caching
  • Lazy loading
  • Eager loading

• Improved maintainability
  • A lot less code to write

• Improved portability
  • ORM framework generates database-specific SQL for you

But Use O/R Mapping Wisely: It’s not a Silver Bullet
DEMO

Hibernate Data Access Layer
Agenda

Using JavaServer Faces technology for the UI
Building a POJO backend
Using Spring in the business tier
Using Hibernate for persistence

**Integrating Spring and JavaServer Faces technology**
Division of Labor: Managed Beans vs. Spring Beans

• Inject service-layer Spring beans into JavaServer Faces technology managed beans
  • Logical separation of UI from Service Layer
  • Integrated support with Spring DelegatingVariableResolver
Spring in a Web Application

```xml
<web>
  <context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>
      classpath:/appCtx/services.xml
      classpath:/appCtx/transactions.xml
      ...
    </param-value>
  </context-param>
  <listener>
    <listener-class>
      org.springframework.web.context.ContextLoaderListener
    </listener-class>
  </listener>
  ...
</web>
```

Stores

Provides access to

ServletContext

WebApplicationContext

WebApplicationContextUtils

defines

getWebApplicationContext(ServletContext) {static}
getRequiredWebApplicationContext(ServletContext) {static}
JavaServer Faces Technology and Spring

- Managed beans = simple dependency injection
- Extend to resolve bean references using Spring
- DelegatingVariableResolver
  - Included with Spring 1.1 and higher
  - First, looks for a JavaServer Faces technology managed bean
  - Then, looks for a Spring bean
JavaServer Faces
Technology Example

```xml
<faces-config>
  <application>
    <variable-resolver>
      org.springframework.web.jsf.DelegatingVariableResolver
    </variable-resolver>
  </application>
...
  <managed-bean>
    <managed-bean-name>
      inboxBean
    </managed-bean-name>
    <managed-property>
      <property-name>
        projectCoordinator
      </property-name>
      <value>
        #{projectCoordinator}
      </value>
    </managed-property>
  </managed-bean>
...
</faces-config>

<beans>
  ...
  <bean id="projectCoordinator"
       class="org...ProjectCoordinatorImpl">
    ...
  </bean>
  ...
</beans>
```
Accessing the WebApplicationContext

• Use the WebApplicationContextVariableResolver
  • Available in Spring 1.25 or later
• Exposes Spring WebApplicationContext under the “webApplicationContext” variable
• Allows you to access the Spring BeanFactory and other services directly from managed beans
• This feature is included in Apache Shale
JavaServer Faces Technology-Spring

- Alternative to DelegatingVariableResolver
- Full bidirectional integration between Spring beans and JavaServer Faces technology managed beans
  - Managed beans can refer to Spring beans
  - Spring beans can refer to managed beans
- Enables integration between Spring MVC and JavaServer Faces technology
- Supports JavaServer Faces 1.1 platform and Spring 2.0
- Open source on SourceForge
  - Sponsored by mindmatters
JBoss Seam Integration

- Spring DelegatingVariableResolver
- Spring integration module
  - Injecting Seam components into Spring beans
  - Injecting Spring beans into Seam components
  - Making a Spring bean into a Seam component
  - Seam-scoped Spring beans
- Some Seam features will be standardized as parts of JavaServer Faces 2.0 platform and the WebBeans JSR
DEMO

JavaServer Faces Technology/
Spring Integration
Summary

- **JavaServer Faces technology, Spring and Hibernate work well together**
  - JavaServer Faces technology implements the presentation tier
  - Spring provides dependency injection and AOP
  - Hibernate transparently persists POJOs
- **JavaServer Faces technology and Spring are seamlessly integrated through dependency injection**
  - Spring 2 integration
  - JavaServer Faces technology-Spring
  - Seam Spring integration
For More Information

- ProjectTrack Sample Code
  - http://code.google.com/p/projecttrack/
- POJOs in Action, Chris Richardson
  - http://www.manning.com/crichardson
- JSF in Action, Kito D. Mann
  - http://www.manning.com/mann
- Official Spring Site
  - http://www.springframework.org
- Official Hibernate Site
  - http://www.hibernate.org
- Official JavaServer Faces Technology Site
  - http://java.sun.com/javaee/javaserverfaces/
For More Information

- **JSF-Spring**
- **JSF Central**
  - [http://www.jsfcentral.com](http://www.jsfcentral.com)
- **Sessions and BOFs**
  - TS-6178—Simplifying JavaServer Faces Component Development
  - TS-4439—Minimalist Testing Techniques for Enterprise Java Technology-Based Applications
  - BOF-4400—Improve and Expand JavaServer Faces Technology with JBoss Seam
  - TS-4514—Three Approaches to Securing Your JavaServer Faces Technology/Spring/Hibernate Applications
Q&A

Kito Mann
Author of JSF in Action
www.virtua.com

Chris Richardson
Author of POJOs in Action
www.chrisrichardson.net
Building JavaServer Faces Applications with Spring and Hibernate

Kito Mann
Author of JSF in Action
Virtua, Inc
www.virtua.com

Chris Richardson
Author of POJOs in Action
Chris Richardson Consulting, Inc
www.chrisrichardson.net

TS-7082