The Shale Framework

Craig R. McClanahan
Senior Staff Engineer
Sun Microsystems, Inc.
Craig.McClanahan@sun.com
Agenda

- Background
- JavaServer Faces and Other Frameworks
- Tour of Shale Features
- Shale and Struts
- Current Status
- Questions and Answers
JavaServer Faces 1.0 released in March 2004:

- Initial focus on getting the component APIs right
- Hidden inside is a *front controller* to handle each request
- Not enough time to address framework aspects well
- So, provided extension points for adding functionality

Extension points can be used by:

- Components – to provide specialized services
- Frameworks – to provide additional functionality
- Applications – to meet specific requirements
JavaServer Faces and Other Frameworks

- JSF came into being in a world filled with frameworks
- Desire to leverage new and old capabilities together
- Two fundamental approaches to framework integration:
  - Treat JSF as a view tier only
  - Treat JSF as a controller and a view tier
- The first approach is available for several frameworks now:
  - Spring
  - Struts
  - Beehive
- And is easily added to others
JavaServer Faces and Other Frameworks

- This first approach has overlapping sets of issues:
  - *Resulting application architecture*:
    - Typically a front controller “in front of” a front controller
    - JSF handles UI events, delegates form submit events
  - *Overall architectural elegance*:
    - Redundant functionality – conversion, validation, page navigation, invoking actions
    - Impedance mismatches – expression language syntax, lifecycle differences
  - Treating JSF as *view tier only* is recommended primarily as a migration strategy, not as an endgame
JavaServer Faces and Other Frameworks

- Building a framework on top of JSF has advantages:
  - Smaller – skip implementing all the redundant functionality
  - Easier to use – only need to learn one approach to each need
  - Enables a focus on *adding features* and *improving ease of use*

- Started work on Shale in Fall 2005, focused on:
  - Adding ease of use APIs inspired by Java Studio Creator
  - Integrate functionality that existing Struts users expect:
    - Client side validation, Tiles layout management
  - Integrate new functionality enabled by JSF
  - (Later) Add a layer that leverages Java SE 5 annotations
JavaServer Faces and Other Frameworks

- To date, I am aware of only one other framework that is taking this approach – JBoss Seam:
  - Focused on tying JSF to JPA and EJB3
  - Also includes features for workflow orchestration
  - Submitted as the basis for JSR-299
- But the extensions capabilities are becoming widely used:
  - Clay / Facelets – Alternative view representations
  - AJAX component libraries – inject phase listeners w/o external configuration
- Treating JSF as a *controller* and a *view* tier is the recommended approach for new projects using JSF
JavaServer Faces Extension Points

- VariableResolver – Customize evaluation of first token in a value binding or method binding expressions
- PropertyResolver – Customize evaluation of the “.” operator in value binding or method binding expressions
- NavigationHandler – Customize navigation decisions
- ViewHandler – Customize creation and restoration of views
- In addition, PhaseListener instances can participate in (and modify) the standard request processing lifecycle
Tour of Shale Features

- **Key Functionality:**
  - View Controller
  - Dialog Handler
  - Clay Plug-In
  - Tiger Extensions
  - Remoting

- **Other Features:**
  - Application Controller
  - JNDI and Spring Integration
  - Unit Testing Framework
  - Struts Feature Integration (Validator, Tiles, Token)
A common pattern in JSF is one backing bean per page.

Must know the JSF request processing lifecycle to understand where to inject some types of application logic.

Example – expensive database query needed to populate a table component.
- Only want to perform the query if it will actually be used.
- Skip it if the user navigated to a different page.

Example – need a transactional resource available through rendering, but then need to clean up.
- Need to regain control after rendering is completed.
View Controller

- Shale provides an optional interface for your backing bean
  - Must also use a naming convention for managed bean names
- Implements the “Hollywood Principle”:
  - Don't call us, we'll call you
- Four application oriented callbacks are provided:
  - init() -- called when view is created or restored
  - preprocess() -- called when about to process a postback
  - prerender() -- called when about to render this view
  - destroy() -- called after rendering, if init() was called
- AbstractViewController – Convenience base class
View Controller – Example Use Case

• Shale MailReader (With JPA) Example Application
  • Typical two-page master-detail CRUD scenario
  • Uses Java Persistence Architecture for database access
    • A Hibernate based application would look very similar
• Usage of view controller callback methods:
  • init() -- Process optional request parameters (bookmarkable URLs)
  • preprocess() -- Restore cached entity instance and mode
  • prerender() – Cache current entity instance and mode
  • destroy() -- No cleanup required
Dialog Handler

- Standard JSF navigation handler decides based on:
  - What view am I currently processing?
  - Which execute action method was invoked?
  - What logical outcome was returned by this action?

- Issue – modelling of a conversation is *ad hoc*

- Issue – how do we deal with conversational state?
  - Pass information in hidden fields
    - Can be unwieldy when numerous fields are required
  - Store information in session
    - Occupies memory if not cleaned up
Dialog Handler

- Dialog Handler abstraction deals with these issues:
  - Models conversations as a state machine
  - Provides storage mechanism for conversational state
  - Heavily inspired by Spring Web Flow, but “JSF-ized”

- States represent processing activities that can occur:
  - *Action* – Call an arbitrary method binding expression
  - *View* – Display a view, plus the subsequent form submit
  - *Subdialog* – Call a named dialog as a black box
  - *Exit* – Exit from the current dialog or subdialog

- State transitions are driven by logical outcome strings
Dialogs can be modelled with UML State Diagrams
Dialog Handler

• Dialog definitions are configured at application startup:
  • Via parsing an XML file
  • Or by programmatic means

• Dialog is entered by returning a special logical outcome:
  • Return `dialog:foo` to enter a dialog named “foo”

• Implemented as a custom `NavigationHandler` extension
Dialog Handler – Example Use Case

- “Use Cases” Demo Application logon dialog:
  - Log on with existing username and password
  - Create user profile and log on
  - Edit existing user profile
  - Optionally support “remember me” cookies
Clay Plug-In

- JavaServer Faces mandates that standard components support JavaServer Pages (JSP) for view representation
  - Optional but recommended for third party components
  - Requires developing tag handler classes and a TLD
- Issue – interoperability problems with template text / tags
  - Mostly resolved with JSF 1.2 and JSP 2.1 (part of Java EE 5)
- Issue – Reuse of portions of page layout is difficult
  - Can be addressed by JSF components focused on this need
- Issue – Some developers prefer a more “pure” HTML representation of the view portion of an application
Clay Plug-In

- Clay enables grafting a component subtree onto an existing component tree
- Sounds simple, but provides compelling features:
  - *HTML Views* – Can separate views into pure HTML pages, with pointers to component definitions
    - Similar capabilities found in Tapestry and Facelets
  - *Metadata Inheritance* – Component definitions can extend previous definitions
    - Similar in spirit to how Tiles can extend other Tiles
    - Create reusable “components” with no Java coding
  - *Symbol Replacement* – Customize managed bean names
Clay Plug-In – Logon Form (JSP)

```html
<h:form>
  <table border="0">
    <tr><td>Username:</td>
        <td><h:inputText id="username"
                         value="#{logon.username}"/></td></tr>
    <tr><td>Password:</td>
        <td><h:inputSecret id="password"
                          value="#{logon.password}"/></td></tr>
    <tr><td><h:commandButton id="logon"
                          action="#{logon.authenticate}"/></td></tr>
  </table>
</h:form>
```
Clay Plug-In – Logon Form (Clay)

<form jsfid="logonForm">
<table border="0">
<tr><td>Username:</td><td><input type="text" name="username" jsfid="username"/></td></tr>
<tr><td>Password:</td><td><input type="password" name="password" jsfid="password"/></td></tr>
<tr><td><input type="submit" value="Log On" jsfid="logon"/></td></tr>
</table>
</form>
Clay Plug-In – Component Definition

```xml
  <component jsfid="username"
    extends="inputText"
    id="username">
    <attributes>
      <set name="required" value="true"/>
      <set name="value" value="#{logon.username}"/>
    </attributes>
  </component>
```
Clay Plug-In

- **So why do I want this?**
  - Pure HTML can be easily built with standard HTML editors
  - Graphic artist can include “sample” data that will be replaced

  ```html
  <table jsfid="addressList">
      ... dummy columns and data values ...
  </table>
  ```

- **Four general strategies are supported:**
  - Strictly XML that uses composite components (addressForm)
  - Tapestry style separate HTML (as illustrated above)
  - Subtree dynamically calculated at runtime (<clay:clay> tag)
  - Pure XML similar to the separate HTML approach
Clay Plug-In – Use Case Examples

- “Clay Use Cases” example application includes four implementations of a simple example (Rolodex)
Tiger Extensions

- JSF and Shale use XML for configuration files:
  - But XML configuration is going out of fashion :-)  
  - Can we reduce or eliminate the need for this stuff?
- Java SE 5 (code name “Tiger”) includes *annotations*:
  - Provide metadata, not functionality
  - Can annotate classes, methods, and fields
  - Can be examined at compile time for code generation
  - Can be processed at runtime
- **NOTE** – Not every config element should be an annotation!
- Tiger Extensions adds annotation support to Shale
Tiger Extensions

- Three categories of annotations are currently supported:
  - Annotated *managed beans*
  - Annotated *view controllers*
  - Annotated *JSF artifact registration*

- All of these annotations are processed at runtime

- Search for annotated classes in a web application:
  - `/WEB-INF/classes`
  - JAR files in `/WEB-INF/lib` that have a `META-INF/faces-config.xml` resource defined
Tiger Extensions – Managed Beans

- Managed beans typically defined in faces-config.xml:

```xml
<managed-bean>
  <managed-bean-name>foo</managed-bean-name>
  <managed-bean-class>...</managed-bean-class>
  <managed-bean-scope>request</managed-bean-scope>
  <!-- Optional property initializations -->
</managed-bean>
```

- Replaced by annotations in Java source code:
  - `@Bean(name="foo", scope=Scope.REQUEST) public class Foo`
  - `@Property("#{bar}") private int bar;`
Tiger Extensions – View Controllers

- Basic Shale requires your backing beans to implement the ViewController interface to receive these services
  - Therefore requires implementing all callback methods
  - Optional base class contains empty definitions
- Tiger Extensions allow you to annotate classes:
  - @View public class Foo { ... }
- And define only callback methods you actually need:
  - @Init public void myInit() { ... }
  - @Preprocess public void setup() { ... }
  - @Prerender public void justBeforeRendering() { ... }
  - @Destroy public void destroy() { ... }
Tiger Extensions – JSF Artifacts

- JSF allows component libraries and applications to register custom artifacts at application startup time:
  - User interface components
  - Converters, renderers, and validators
- Tiger extensions allow annotated “self registration”:
  - @FacesComponent("componentType")
  - @FacesConverter("converterId")
  - @FacesRenderer(renderKitId="x", componentFamily="y", rendererType="z")
  - @FacesValidator("validatorId")
Tiger Extensions – Example Use Case

- Shale SQL Browser – analog to SQL command console:
  - Allow user to perform arbitrary SQL SELECT statements
  - Dynamically reconfigure table columns based on query results
  - In prerender(), execute query and rebuild component tree
  - In destroy(), clean up JDBC resources that were used

`Query.java` class level annotations:

- `@Bean(name=“query”, scope=Scope.REQUEST) @View public class Query { ... }

`Query.java` method level annotations:

- `@Prerender` public void prerender() { ... }
- `@Destroy` public void destroy() { ... }
Remoting

- It is common for applications to respond to programs as well as to humans:
  - Web services
  - AJAX-based asynchronous requests
- Shale provides features to make this easier:
  - For application developers
  - For JSF component authors
- Packaged as a small (40k) JAR, only depends on JSF
  - Zero configuration if you accept the defaults
  - Implemented as a JSF PhaseListener
Remoting

- Primary concept is the *Processor*:
  
  ```java
  public interface Processor {
      public void process(FacesContext context, String resourceId)
      throws IOException;
  }
  ```

- Processor examines resource identifier and constructs the *entire* response

- Processors are registered to a URL pattern like servlets:
  - Path mapping and extension mapping are supported
  - Creates a *FacesContext* so you can use EL expressions and managed beans
Remoting

- Processor architecture is extensible:
  - Each processor mapped to a URL pattern
  - Application specific Processors can be configured

- Standard processor implementations are provided:
  - Serve static resource from the classpath (embedded in JARs)
  - Serve static resource from the web application
  - Map to a dynamically generated method binding:
    - Executes method binding #{foo.bar} to return the response
Remoting

- Helper classes to assist developers:
  - Two-way mapping of resource id \(<---->\) URL
  - Create \textit{ResponseWriter} implementation for dynamic output
- AJAX demonstration components delivered with Sun Java Studio Creator were implemented with Shale Remoting
  - http://developers.sun.com/jscreator/
Other Shale Features

- **Application Controller**
  - Configured as a servlet filter
  - Supports decoration of the request processing lifecycle
    - Uses “chain of responsibility” design pattern (Commons Chain)
    - Similar in spirit to customizing request processor in Struts 1.3.x

- **JNDI and Spring Integration:**
  - Custom JSF variable and property resolvers
  - Transparent access to JNDI contexts and Spring created beans, via EL expressions

- **Unit testing framework:**
  - Mock objects for building unit tests
Other Shale Features

- Struts Functionality Equivalents:
  - Commons Validator for client side validation
    - Implemented as a JSF validator
  - Tiles Support
    - Based on “standalone” version of Tiles being developed
    - No dependency on Struts
    - Can navigate to a view or to a tile
  - Transaction token support
    - Prevent duplicate submits of a form
    - Implemented as a component that fires validation failures on duplicate submits
Shale was originally proposed to the Struts developers as a “revolutionary” basis for Struts 2.0:
- Not accepted in that role
- Was accepted as a subproject of the Struts project

More recently, developers of WebWork 2 expressed interest in merging with Struts project to create Struts 2.0:
- WebWork 2 plus tweaks to become basis for Struts 2.0
- Very elegant evolution of “action” oriented framework design

Shale recently migrated to become a top level Apache project of its own:
- http://shale.apache.org/
Current Status

- Current release is 1.0.2, voted “beta” quality:
  - Depends on unreleased Standalone Tiles library
  - Significant functional issues in dialog functionality
- A 1.0.3 release is imminent:
  - Many bugfixes, some new features
  - Build environment switched from Ant to Maven 2
  - Framework and sample applications each packaged separately
- Most APIs in Shale are stable enough to use today:
  - Pay attention to which APIs are designed for use by applications, versus those extending the framework
The Shale Framework

Craig R. McClanahan
Senior Staff Engineer
Sun Microsystems, Inc.
Craig.McClanahan@sun.com