Testing JSF Applications

Stan Silvert
JBoss Core Developer

Overall Presentation Goal

Learn about approaches to teting JSF Applications with emphasis on JBoss JSFUnit and SeamTest.

Pronouncement

"JBoss JSFUnit is the first open source community dedicated to JSF testing."

How can we test JSF?

- Static Analysis
- Performance Testing
- Client-Centric Black Box
- Mock-Centric White Box
- Comprehensive “acrylic” Box

Static Analysis

- Tests performed on an application without running it
- For JSF, this means testing the composite configuration
- Find problems early
- For JSF, only available in JSFUnit
- If you do no other testing, at least do this. It's free.

JSFUnit Static Analysis

- Managed Beans/EL
- Managed Bean methods
- Serializable scopes
- faces-config.xml typos
- Managed Bean Class
- Proper Interface Implementations
- Missing getters/setters
- Duplicate Managed Beans
- JSF Tags

Does it implement java.io.Serializable?

```
<managed-bean>
   <managed-bean-name>
      shoppingCart
   </managed-bean-name>
   <managed-bean-class>
      com.foo.demo.ShoppingCart
   </managed-bean-class>
   <managed-bean-scope>
      session
   </managed-bean-scope>
</managed-bean>
```

junit.framework.AssertionFailedError: Managed bean 'shoppingCart' is in session scope, so it needs to implement interface java.io.Serializable

Duplicate Managed Beans

```
<managed-bean>
  <managed-bean-
 name>shoppingCart</managed-bean-
 name>
  <managed-bean-
 class>com.foo.ShoppingCart</managed-
 bean-class>
  <managed-bean-
 scope>application</managed-bean-
 scope>
</managed-bean>

<managed-bean>
  <managed-bean-
 name>shoppingCart</managed-bean-
 name>
  <managed-bean-
 class>com.foo.dup.ShoppingCart</mana
```

How can we test JSF?

- Static Analysis
- Performance Testing
- Client-Centric Black Box
- Mock-Centric White Box
- Comprehensive “acrylic” Box

Performance Analysis

- A measurement of the time it takes to complete a task or subtask.
- You can do this at many different levels of abstraction and there are many, many tools to help.
- JSFUnit offers a simple tool to measure time spent in the JSF lifecycle.

Performance Analysis with JSFUnit

web.xml:

```
<context-param>
  <param-name>javax.faces.CONFIG_FILES</param-name>
  <param-value>/WEB-INF/timer-config.xml</param-value>
</context-param>
```

timer-config.xml:

```
<faces-config>
  <lifecycle>
    <phase-listener>
org.jboss.jsfunit.framework.JSFTimerPhaseListener
    </phase-listener>
  </lifecycle>
</faces-config>
```

Performance Analysis with JSFUnit

```
public void testJSFPerformance() throws SAXException,
  IOException {
   JSFClientSession client = new JSFClientSession(“/foo”);
   client.setParameter("inputText", "Hello");
   client.submit("submit_button");

   JSFTimer timer = JSFTimer.getTimer();
   assertTrue(timer.getTotalTime() < 3000);

   PhaseId appPhase = PhaseId.INVOKE_APPLICATION;
   assertTrue(timer.getPhaseTime(appPhase) < 2000);
}
```

How can we test JSF?

- Static Analysis
- Performance Testing
- Client-Centric Black Box
- Mock-Centric White Box
- Comprehensive “acrylic” Box

Client-Centric Testing: Black Box

For JSF, black box testing means sending HTTP requests to the server and validating the HTML that is returned.

Client-Centric Testing: Black Box

- Manual Testing: Open a browser and try it out. (Sadly, the most common approach)
- Browser keystroke/mousestroke playback: Selenium Recorder
- Browser emulators: HttpUnit, HTMLUnit, WebTest, etc.

Problems with Black Box JSF Tests

- It's hard to validate plain HTML
- Automated tests can break because of cosmetic HTML changes
- Correct output at the client doesn't always mean correct behavior at the server
- Can be difficult to use in a continuous integration process
- AJAX support is lacking. You'll have a hard time testing RichFaces apps this way.

How can we test JSF?

- Static Analysis
- Performance Testing
- Client-Centric Black Box
- Mock-Centric White Box
- Comprehensive “acrylic” Box

Mock-Centric White Box Tests

- Test the server-side classes without running the whole application
- Create fake domain objects called “mocks”
- Use the mocks to simulate the runtime environment
- Test your code inside the simulated environment using JUnit, TestNG, etc.

Mocks for JSF: Seam Test

- Fine-grained testing of Seam components don't need mocks. Do it the old-fashioned way.
- Combine Seam Test with JBoss Embedded container for a mock environment.
- This gives you a mock JSF environment for testing.

Mocks for JSF: Seam Test

```
public class RegisterTest extends SeamTest {
@Test
public void testRegister() throws Exception {
    new FacesRequest() {
        @Override
        protected void processValidations()throws Exception{
            validateValue("#{user.username}", "1ovthafew");
            validateValue("#{user.name}", "Gavin King");
            validateValue("#{user.password}", "secret");
            assert !isValidationFailure();
        }
// more ...
```

Mocks for JSF: Seam Test

```
@Override
protected void updateModelValues() throws Exception {
   setValue("#{user.username}", "1ovthafew");
   setValue("#{user.name}", "Gavin King");
   setValue("#{user.password}", "secret");
}
@Override
protected void invokeApplication() {
   assert invokeMethod("#{register.register}").equals("success");
}
@Override
protected void renderResponse()
```

Mocks for JSF: Seam Test

```
   @Override
   protected void renderResponse() {
      assert getValue("#{user.username}").equals("1ovthafew");
      assert getValue("#{user.name}").equals("Gavin King");
      assert getValue("#{user.password}").equals("secret");
   }
}.run();  // start FacesRequest anonymous class
```

Problems with Mocks

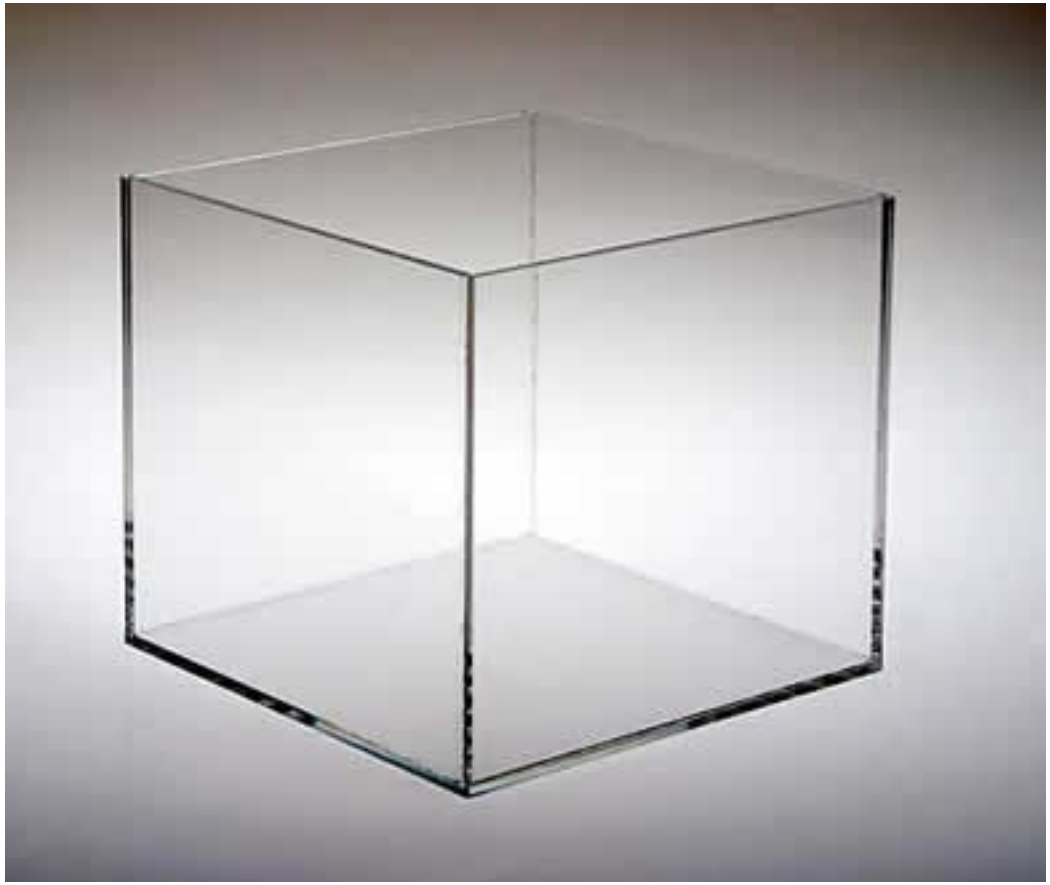
- A simulated environment always falls short. Use cases that pass with mocks can fail in the real container.
- Very fine-grained. Object interactions not tested well.
- Mock libraries can easily get large and out of control.
- Often forces application changes to accommodate mocks. (init's and factories)

How can we test JSF?

- Static Analysis
- Performance Testing
- Client-Centric Black Box
- Mock-Centric White Box
- Comprehensive “acrylic” Box

JSFUnit Acrylic Box Testing

See it all, inside and out.



JSFUnit Comprehensive Testing

- Static Analysis
- JSF Performance Analysis
- Unit and Integration Tests
- Uses ordinary JUnit code
- Runs inside the real container
- Generates real HTTP requests
- No need for mock objects

JSFUnit Comprehensive Testing

- Tests can be as fine or course grained as you like.
- API is simplified for JSF
- Easily integrates into your build
- Test client side HTML and server side state all in the same test!

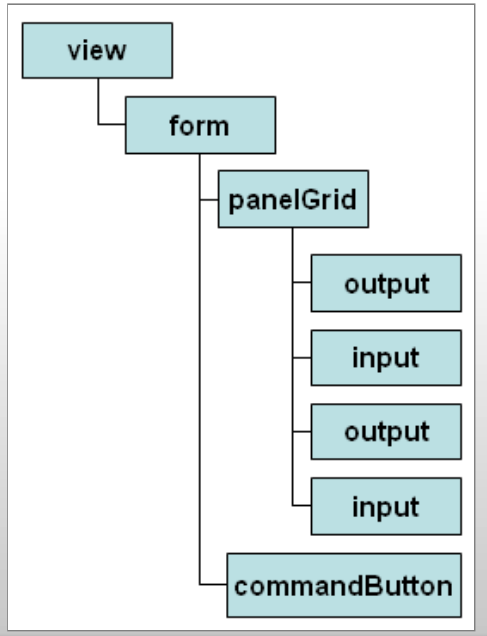
JSF Components: Markup to Browser

1 Development

Facelets/JSP Page

```
<h:panelGrid columns="2">
  <h:outputText value="#{msg.emailLabel}"/>
  <h:inputText value="#{login.email}"/>
  <h:outputText value="#{msg.passwordLabel}"/>
  <h:inputText value="#{login.password}"/>
</h:panelGrid>
<h:commandButton action="#{login.check}"
                 value="#{msg.btnLabel}"/>
</h:form>
```

2 JSF Component Tree

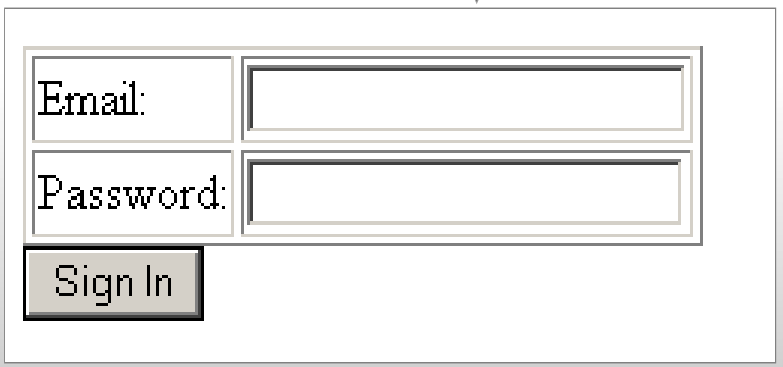


3 Output

HTML

```
<table border="1">
<tbody>
<tr>
<td>Email:</td>
<td><input type="text" name="_id0:_id3" /></td>
</tr>
<tr>
<td>Password::</td>
<td><input type="password" name="_id0:_id5" value="" /></td>
</tr>
</tbody>
</table>
```

Browser



JSFUnit: Testing Hello World

index.jsp:

```
<f:view>
    <h:form id="form1">
      <h:outputText value="Enter your name:"  id="prompt"/>
      <h:inputText value="#{name.text}"
                   id="input_name_text"/>
      <h:commandButton action="/hello.jsp"
                       id="submit_button"/>
    </h:form>
</f:view>
```

JSFUnit: Testing Hello World

hello.jsp:

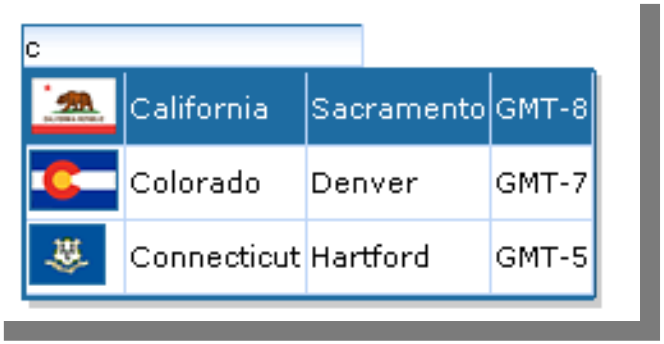
```
<f:view>
     <h:outputText value="Hello, #{name.text}"
                   id="greeting"/>
</f:view>
```

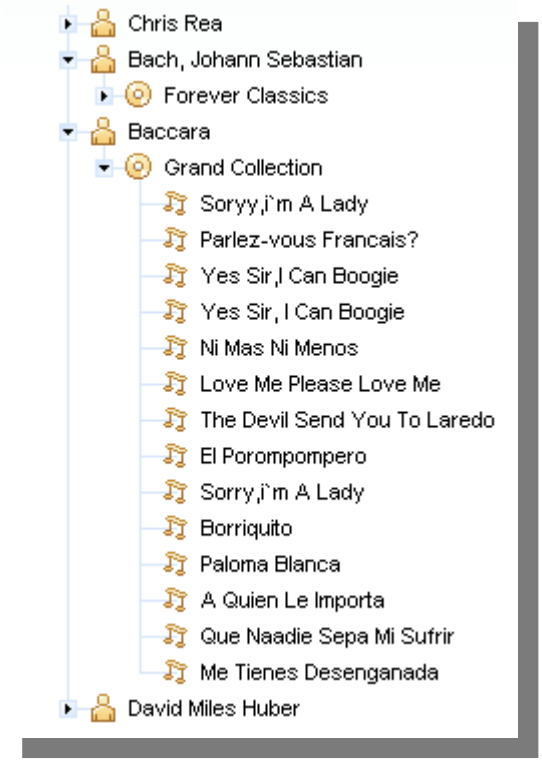
JSFUnit: Testing Hello World

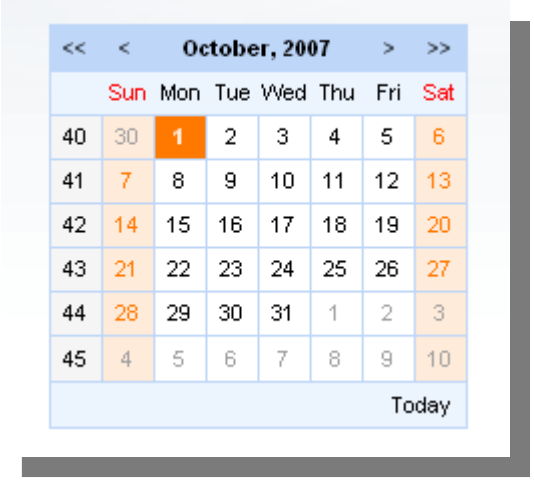
```
public void testHelloWorld() {
   JSFClientSession client =
                      new JSFClientSession("/index.faces");
   JSFServerSession server = new JSFServerSession(client);
   client.setParameter(“prompt”, “Stan”);
   client.submit(“submit_button”);
   assertEquals("/hello.jsp", server.getCurrentViewID());
   assertTrue(client.getWebResponse()
                  .getText()
                  .contains(“Hello, Stan”));
   assertEquals("Stan",
               server.getManagedBeanValue("#{name.text}"));
 }
```

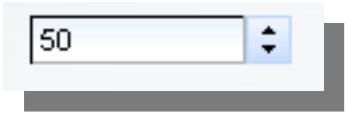
JSFUnit Hello World Demo

How can I test these complex RichFaces components?









Testing RichFaces with JSFUnit

- Plain JSFClientSession won't quite cut it.
- RichFaces uses javascript to send customized requests to the server.
- Javascript is also used in a custom way to handle the response and update the page.
- JSFUnit provides RichFacesClient to handle the special processing of RichFaces components.

Creating a RichFacesClient for JSFUnit

```
JSFClientSession client = new JSFClientSession(“/index.jsp”);
RichFacesClient richClient = new RichFacesClient(client);
```

Notable methods in the RichFacesClient

- ajaxSubmit(componentId)
- clickDataTableScroller(compId, value)
- clickTab(tabPanelId, tabComponentId)
- dragAndDrop(dragCompId, dropCompId)
- setCalendarValue(compId, value)
- setDataFilterSlider(compId, value)
- setInputNumberSlider(compId, value)
- setInputNumberSpinner(compId, value)

JSFUnit RichFaces Demo

Summary

- For JSF, black box style testing is brittle and limited
- White box, mock-style, testing also has limitations
- JSFUnit allows testing in the real container with real requests and real domain objects
- JSFUnit has static & perf analysis
- Testing Ajax components is challenging, but JSFUnit provides an API for RichFaces

For more information

- www.jsfunit.org
- jsfunit.blogspot.com
- JSFUnit Forum linked from jsfunit.org
- stan@jboss.com

Questions?

Headline (Liberation Sans Bold, 28 pt)

- Sub-line (Liberation Sans, 25 pt.)
  - Sub-line (Liberation Sans, 25 pt)