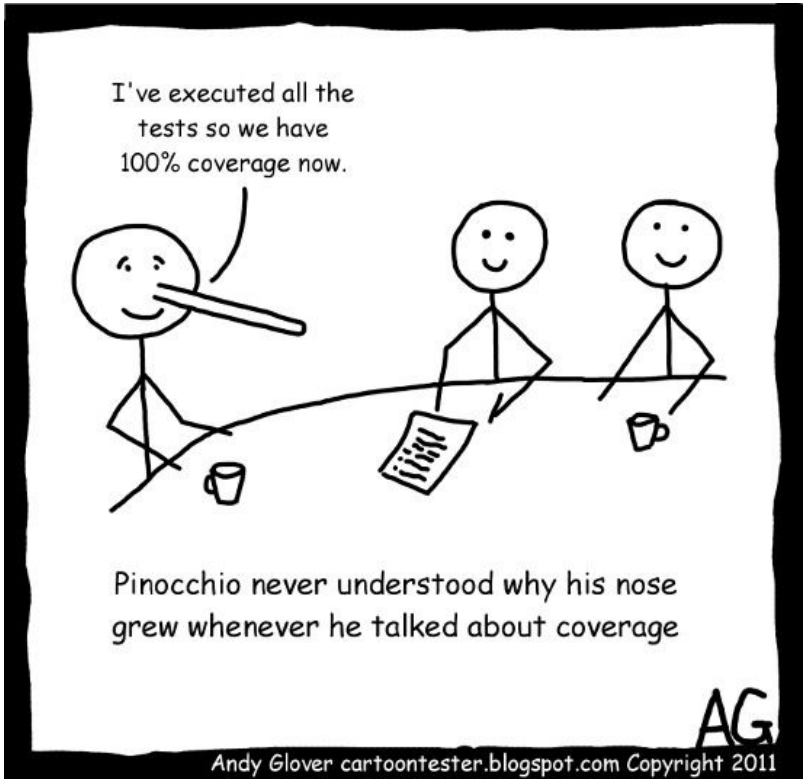
Testing & Java Server Faces

GÖTEBORGS UNIVERSITET

DAT076/DIT126 Web Applications

Adam Waldenberg

Testing EE applications



Testing an EE application is complicated

- How do you handle CDI and all the EE annotations in the model?
- How do you support the database?
- Many EE applications simply don't bother and just do pure model tests, manually filling in any required injected members with reflection
- Is there a problem with this approach?

Testing approaches - Mocking

Mocking is the process of simulating behavior and responses of objects, including encapsulated objects

- **Mockito** *(site.mockito.org)*
- **EasyMock** *(easymock.org)*
- **PowerMock** *(github.com/powermock/powermock)*, extends Mockito and EasyMock, allowing you to mock static methods, private methods and constructors

EASYMOCK

- Reaching anything near to complete code coverage, let alone full logical coverage in an EE application - is quite a hassle because of all the required mocking

When mocking simply doesn't cut it

Two notable solutions exist

- **CDI-Unit** *(bryncooke.github.io/cdi-unit)*
- **Arquillian** *(arquillian.org)* - No more mocks. No more container lifecycle and deployment hassles. Just *real* tests!
- A complete shift in the way we write tests, allowing us to mimic or get the behavior of the application server and the EE framework
- Results in much simpler and cleaner test code

Introducing Arquillian

Allows us to write fully EE-aware JUnit tests and integration test

- From the team behind JBoss and WildFly
- Supported and developed by Red Hat and supported by JBoss Enterrpise
- Supports CDI, JPA, EJB, Java Server Faces and most (if not all) parts of the EE infrastructure
- Can run tests in both client and server mode

Testing with Arquillian

```
@RunWith(Arquillian.class)
public class BasicClientTest {
    @Deployment
    public static WebArchive createDeployment() {
        return ShrinkWrap.create(WebArchive.class)
            .addClasses(MyBean.class)
            .setWebXML("WEB-INF/web.xml");
    }

    @Test
    public void shouldDoSomethingSuccessfully() { … }
}
```

- *Arquillian.class* handles all the heavy lifting
- Requires a shrink wrap for bootstrapping the application server into the test

More testing with Arquillian

```
@RunWith(Arquillian.class)
public class BasicClientTest {
    @Deployment
    public static WebArchive createDeployment() { … }

    @EJB
    private EntityDAO entityDao;

    @Test
    public void shouldReturnThisAndThatOnQuery() { … }
}
```

- Most (maybe all?) types of injections are supported
- Configuration files and resources are processed and looked for in the *test/* directory. This allows us to have a separate collection of resources for tests

Frontend development in EE

Several options are available for developing the frontend code in an EE application

- **JSP with scriptlets** — Combines Java code and markup code into a *.jsp* file. Resembles the coding style of PHP scripts
- **Pure servlets** — Markup code is generally created and outputted by the servlet
- ***Java Server Faces*** — The cleanest and most modern solution. Supports components and clean separation
- **Requestlets** — Planned for EE 9. Convenience layer on top of servlets. Unlike servlets they are also CDI beans

Java Server Faces

Has a long incremental history of improvements over a span of over fifteen years

- JSF 1.0 (2004-03-11)
- JSF 1.1 (2004-05-27)
- JSF 1.2 (2006-05-11)
- JSF 2.0 (2009-07-01)

Where yours truly started to get involved

- JSF 2.1 (2010-11-22)
- JSF 2.2 (2013-05-21)
- JSF 2.3 (2017-03-28)
- JSF 3.0 (Coming in EE 9)

Overview of Java Server Faces

Java Server Faces is a component-based architecture with a design focus on MVC

- Treats view elements as JSF UI components instead of HTML
- Maintains an internal component tree (similar to the DOM)
- Callbacks and events back to Java work with *ActionEvent* calls similar to Swing and JavaFX
- Each tag in a page has a JSF tag handler class and component
- The JSF component class handles translation of of JSF tags to HTML tags, and interpretation of HTTP requests

View Definition Language

Java Server Faces uses an exchangeable View Definition Language (VDL) to define the user interface

- Originally JSP (Java Server Pages via *.jspx* files) was used. Not something we will cover as it is considered deprecated – but for historical reasons it's interesting to mention
- JSP(x) was replaced by Facelets in JSF 2.0
- All major new features from this version on, such as templating, composite components, and more, are only available for Facelets

Java Server Faces XML namespaces

The Standard EE component library consists of various name spaces as shown below

```
http://xmlns.jcp.org/jsf                  jsf:  Pass-through elements
http://xmlns.jcp.org/jsf/core             f:    Core library, not HTML
http://xmlns.jcp.org/jsf/html             h:    HTML library
http://xmlns.jcp.org/jsf/facelets         ui:   Facelet Templating tag library
http://xmlns.jcp.org/jsf/composite        cc:   Composite Component tag library
http://xmlns.jcp.org/jsf/passthrough      pt:   Pass-through attributes

http://xmlns.jcp.org/jsp/jstl/core        c:    JSP Standard Tag Library (JSTL)
http://xmlns.jcp.org/jsp/jstl/functions   fn:   JSTL functions
```

- Can be extended with external libraries
- ***Examples:*** PrimeFaces, BootsFaces, OmniFaces and DeltaSpike

Testing Java Server Faces

```
@RunWith(Arquillian.class)
public class BasicClientTest {
    @Deployment
    public static WebArchive createDeployment() { … }

    @Drone private WebDriver browser;

    @FindByJQuery("input#name")
    private WebElement name;

    @RunAsClient @Test
    public void shouldReturnThisAndThatOnQuery() { … }
}
```

- *@RunAsClient* allows us run the test on the client-side
- Allows us to combine server-side tests and client-side tests
- Uses a combination of Arquillian, Graphene, Drone and Selenium

Expression Language

An integral part of Java Server Faces for adding expression inside the XML documents that define the views

- Almost Java but not quite!

```
#{myBean.myMethod}
#{myBean.myMethod()} // method expressions

#{myBean.myProperty}
#{myBean.myProperty + 10}
#{myBean.myProperty == 5}
#{myBean.invoice.customer["street"]} // value expressions
```

- A very small taste of what you can do !

Expression Language continued

A number of *implicit objects* exist

- application
- component
- cookie
- facesContext
- flash
- header
- initParam
- param
- request
- resource
- session
- view
- … and more ...

An example view descriptor

```
<?xml version="1.0" encoding="UTF-8"?>
<html xmlns="http://www.w3.org/1999/xhtml"
xmlns:h="http://xmlns.jcp.org/jsf/html"
      xmlns:p="http://primefaces.org/ui"
xmlns:f="http://xmlns.jcp.org/jsf/core"
      xmlns:fn="http://xmlns.jcp.org/jsp/jstl/functions">
    <h:head><title>Test</title></h:head>
    <h:body>
        <p>
            The name was... <h:outputText value="#{formBackingBean.name}"
            and is #{fn:length(formBackingBean.name)} /> characters long!
        </p>
        <h:form>
            <p:inputText value="#{formBackingBean.name}" />
            <p:commandButton action="#{formBackingBean.onClicked}"
                             value="Set name" update="@(p)" />
        </h:form>
    </h:body>
</html>
```

The backing bean for the view

```
@Data
@Named
@ViewScoped
public class FormBackingBean implements Serializable {

    private String name;

    @PostConstruct
    private init() {
        name = “John Smith”;
    }

    public void onClicked() {
        /* In this case we don’t need to do anything */
    }
}
```

- So what is *@Named* used for ?

Component libraries

As stated before, Java Server Faces can be easily extended with component libraries

One of the most notable is ***PrimeFaces***

Let’s take a look:
www.primefaces.org/showcase

Lets make an example!

Let’s see if we can take what we learned so far and quickly write a Java Server Faces application