SeamTest in action

loose notes by Bernard Łabno

version 1

Table of Contents

Introduction

Scattering functionality among several components for better re-use and dependencies among them make project written in Seam a house of cards, at least that is my experience. Very often a small change was bringing the whole structure down. Worst part of it was I do not know which part stopped working unless I test all the use cases. That is why automatic testing is a must.

Thanks to SeamTest we can test quite a lot of things. SeamTest behaves however not to intuitively and I have spent many hours debugging its source to learn what are the right ways to test particular things. Because not every way is good.

As far as now I have managed to test following things :

1. input validation – test if JSF will skip invoking action and updating my data, or not
2. happy path use case – test if a use case can be executed the way it is ment; one of the most important tests in my opinion
3. pages security
4. components security

About the sample application

I love to learn by example so we will learn using a sample application. It is very simple seam-gen generated pice of code. It contains one entity named Tag, one action bean TagHome, TagList bean and a test case of course. Tag consist of id and name. Our application has following use cases :

1. add tag
2. edit tag
3. remove tag
4. view tag's details
5. list tags

All use cases may be performed only by administrator.

To add tag user enters “/admin/tags/tagList.xhtml”, preses “Create” button, fills and submits form.

To view tag details user must enter “/admin/tags/tag.xhtml” and provide valid identifier as a GET request parameter. If there is no tag with given id, id is malformed or user did not provide it user

must be redirected to error page with appropriate communicate.

To edit tag user enters tag's details page, presses “Edit” button, modifies input filed and submits the form.

To remove tag user enters tag's details page and preses “Remove” button.

Note that the same facelet – tag.xhtml – is used to both view, add and edit use cases. It is splited into two parts : edit form and details view. Both fragments are conditionally rendered depending on value of #{tagHome.editMode} expression.

```
        <h:form rendered="#{tagHome.editMode}">
            <rich:panel>
                <f:facet name="header">#{messages['tag']}</f:facet>
                <s:decorate template="/layout/edit.xhtml">
                    <ui:define name="label">#{messages['tag.name']}</ui:define>
                    <h:inputText id="name" required="true" value="#{tagHome.instance.name}"/>
                </s:decorate>
                <div style="clear:both"/>
            </rich:panel>
            <div class="actionButtons">
                <h:commandButton value="#{messages['CRUD.save']}" action="#{tagHome.persist}"
                                 rendered="#{!tagHome.managed}"/>
                <h:commandButton value="#{messages['CRUD.save']}" action="#{tagHome.update}"
                                 rendered="#{tagHome.managed}"/>
                <s:button value="#{messages['CRUD.cancel']}" action="cancel"/>
            </div>
        </h:form>
        <s:fragment rendered="#{not tagHome.editMode}">
            <rich:panel>
                <f:facet name="header">#{messages['tag']}</f:facet>
                <h:panelGrid columns="2">
                    #{messages['tag.name']}
                    <h:outputText value="#{tagHome.instance.name}"/>
                    #{messages['tag.version']}
                    <h:outputText value="#{tagHome.instance.version}"/>
                </h:panelGrid>
            </rich:panel>
            <div class="actionButtons">
                <s:button value="#{messages['CRUD.edit']}" action="#{tagHome.edit}"/>
                <s:button value="#{messages['CRUD.remove']}" action="#{tagHome.remove}"/>
                <s:button value="#{messages['CRUD.done']}" action="done"/>
            </div>
        </s:fragment>
```

tag.xhtml

The editMode property of tagHome is set to true (form will be displayed) by #{tagHome.startCreateProcess} or #{tagHome.edit}. Property is changed to false and form is hidden when #{tagHome.persist} or #{tagHome.update} are invoked. Note that if user presses “Cancel” button than navigation rules defined in tag.page.xml end conversation, so even if user stays on the same page (tag.xhtml) there is new conversation started and new instance of tagHome with editHome set to default of false present.

Take a look at tag.page.xml :

```
<page>

    <param name="id" value="#{tagHome.id}" converterId="javax.faces.Long"/>

    <action execute="#{tagHome.validate}"/>

    <begin-conversation join="true" flush-mode="manual"/>

    <navigation from-action="#{tagHome.persist}">
        <rule if-outcome="persisted">
            <end-conversation before-redirect="true"/>
            <redirect/>
        </rule>
    </navigation>
```

```
    <navigation from-action="#{tagHome.update}">
        <rule if-outcome="updated">
            <end-conversation before-redirect="true"/>
            <redirect/>
        </rule>
    </navigation>

    <navigation from-action="#{tagHome.remove}">
        <rule if-outcome="removed">
            <end-conversation before-redirect="true"/>
            <redirect view-id="/admin/tags/tagList.xhtml"/>
        </rule>
    </navigation>

    <navigation from-action="cancel">
        <rule if="#{tagHome.managed}">
            <end-conversation before-redirect="true"/>
            <redirect view-id="/admin/tags/tag.xhtml"/>
        </rule>
        <rule if="#{not tagHome.managed}">
            <end-conversation before-redirect="true"/>
            <redirect view-id="/admin/tags/tagList.xhtml"/>
        </rule>
    </navigation>

    <navigation from-action="done">
        <end-conversation before-redirect="true"/>
        <redirect view-id="/admin/tags/tagList.xhtml"/>
    </navigation>

</page>
```

Thanks to page parameter, it is enough to supply id parameter to URL and Seam will fetch desired tag from database.
Page action #{tagHome.validate} checks if user is not trying to enter page without providing valid id.
Conversation is started on page entering.
The rest is obvious.

Let's take a look at our action bean.

```
@Name("tagHome")
public class TagHome extends EntityHome<Tag> {

    private boolean editMode;

    @Restrict("#{s:hasRole('ADMIN')}")
    @Override
    public String persist() {
        setEditMode(false);
        return super.persist();
    }

    @Restrict("#{s:hasRole('ADMIN')}")
    @Override
    public String update() {
        setEditMode(false);
        return super.update();
    }

    @Restrict("#{s:hasRole('ADMIN')}")
    @Override
    public String remove() {
        return super.remove();
    }

    public boolean isEditMode() {
        return editMode;
    }

    public void setEditMode(boolean editMode) {
        this.editMode = editMode;
    }

    public void edit() {
        setEditMode(true);
```

```
    }

    /**
     * Invoke this method to switch to editMode.
     */
    public void startCreateProcess() {
        setEditMode(true);
        clearInstance();
    }

    /**
     * Used in tag.page.xml to see if identifier is valid.
     * Identifier may be null only if user wants to add new tag
     * so this component must be in edit mode.
     */
    public void validate() {
        if (getId() == null && !isEditMode()) {
            throw new EntityNotFoundException(getId(), Tag.class);
        }
        getInstance();
    }

}
```

As you can see the core methods are restricted to admin only.

```
        <rich:panel>
            <f:facet name="header">#{messages['tagList']}</f:facet>

            <h:outputText value="No tag exists" rendered="#{empty tagList.resultList}"/>

            <h:dataTable var="tag" value="#{tagList.resultList}"
                         rendered="#{not empty tagList.resultList}">
                <h:column>
                    <f:facet name="header">#{messages['tag.id']}</f:facet>
                    #{tag.id}
                </h:column>
                <h:column>
                    <f:facet name="header">#{messages['tag.name']}</f:facet>
                    <s:link id="tag" value="#{tag.name}" propagation="none"
                            view="/admin/tags/tag.xhtml">
                        <f:param name="id" value="#{tag.id}"/>
                    </s:link>
                </h:column>
            </h:dataTable>

        </rich:panel>

        <div class="actionButtons">
            <s:button value="#{messages['CRUD.create']}" action="#{tagHome.startCreateProcess}"
                      view="/admin/tags/tag.xhtml"/>
        </div>
```

tagList.xhtml

Writing the first test

SeamTest uses TestNG as testing framework.

Lets see a sample test class.

```
public class TagCRUDTest extends SeamTest {

      @Test
      public void myFirstTest() {
             assert true;
      }
       @Test
      public void mySecondTest() {
             assert true;
      }
}
```

Easy, isn't it ? It is just a pure TestNG test. Is extending SeamTest necessary ? Yes, it is very important. It contains methods that start and stop embedded Jboss server and start/stop sessions.

> Note that each @Test annotated method represents separate user session, so do not expect stuff you have put into session context in one method to be present in that context in another test method.

Writing a SeamTest comes down to overriding JSF lifecycle methods. These are :

1. applyRequestValues
2. processValidations
3. updateModelValues
4. invokeApplication
5. renderResponse

```
@Test
public void createTag() throws Exception {
    String conversationId = new RenderedNonFacesRequest(TAG_VIEW) {
        @Override
        protected void beforeRequest() {
            setParameter("actionMethod", TAG_LIST_VIEW + ":tagHome.startCreateProcess");
        }

        @Override
        protected void renderResponse() throws Exception {
            assert (Boolean) getValue("#{tagHome.editMode}");
        }
    }.run();

    new InvokedApplicationFacesRequest(TAG_VIEW, conversationId) {
        @Override
        protected void processValidations() throws Exception {
            validateValue("#{tagHome.instance.name}", "My new legal act");
            assert !isValidationFailure();
        }

        @Override
        protected void updateModelValues() throws Exception {
            setValue("#{tagHome.instance.name}", "My new legal act");
        }

        @Override
        protected void invokeApplication() throws Exception {
            invokeAction("#{tagHome.persist}");
            id = (Long) getValue("#{tagHome.instance.id}");
            assert null != id;
        }
    }.run();
}
```

The applyRequestValues and updateModelValues are used for inserting values into your business objects. I do not know what is the difference between them in SeamTest. In real JSF the first one loads data into JSF components i.e.: UIInput and the latter transfers data from components to business components if validataion passed successfully.

In processValidations method we validate values. If data is invalid then invokeApplication phase is skipped. In invokeApplication we invoke actions. At last in renderResponse we can check viewId or other stuff.

There are also two more methods :

1. beforeRequest
2. afterRequest

First may be used to provide GET request parameters. The latter I use for checking if main lifecycle

methods were invoked.

> Notice that both methods are invoked when no scope is active, so do not use Component.getInstance() nor context browsing.

Why should we need to check if those methods were invoked ? Lets see an example.

```
@Test(
public void invalidInputTest() throws Exception {
    new FacesRequest("/admin/tags/tag.xhtml") {
        @Override
        protected void applyRequestValues() throws Exception {
            setValue("#{tagHome.instance.name}", "some invalid value");
        }
        @Override
        protected void processValidations() throws Exception {
            validateValue("#{tagHome.instance.name}", "some invalid value");
        }
        @Override
        protected void invokeApplication() {
            assert isValidationFailure();
        }
    }.run();
}
```

Above code is wrong because if validation fails in processValidations method then invokeApplication method will be skipped and isValidationFailure() assertion will be lost. It happened to me a lot that first the invokeApplication() (or other) method was being invoked, but after several changes to my project it stopped, i.e.: validations failed or even post restore redirect happened (user is not logged in and he should, or he is not authorized).

So which method is invoked for sure ? If no bug in Seam happenes then it will be afterRequest() method. Correct test should look like this :

```
@Test
public void invalidInputTest() throws Exception {
    new FacesRequest("/admin/tags/tag.xhtml") {
        @Override
        protected void applyRequestValues() throws Exception {
            setValue("#{tagHome.instance.name}", "some invalid value");
        }
        @Override
        protected void processValidations() throws Exception {
            validateValue("#{tagHome.instance.name}", "some invalid value");
        }
        @Override
        protected void afterRequest() {
            assert isValidationFailure();
            assert !isInvokeApplicationBegun();
        }
    }.run();
}
```

Running test

In order to run tests enter you project directory and call ant target "test".

```
bernard@kuna:~/projects/seamTestArticle$ ant test
```

Note that you need to put TestNG descriptor in your src/test directory.

> Descriptor name must match *Test.xml !

Our sample descriptor :

```
<suite name="All tests" verbose="5" parallel="false">
    <test name="Tag tests">
        <classes>
            <class name="pl.labno.bernard.seamTestArticle.test.TagCRUDTest"/>
        </classes>
    </test>
</suite>
```

Which request

There are two request types - POST and GET. SeamTest provides NonFacesRequest and FacesRequest classes. You can override following methods :

- in NonFacesRequest :
  - beforeRequest
  - renderResponse
  - afterRequest
- in FacesRequest :
  - beforeRequest
  - applyRequestValues
  - processValidations
  - updateModelValues
  - invokeApplication
  - renderResponse
  - afterRequest

Use NonFacesRequest if you want to simulate entering page via URL bar. Use FacesRequest if you want to simulate form submit.

Because we make assertions inside lifecycle method we should always check if those methods were actually invoked. It is save some overhead I use additional class that extends SeamTest and provides RenderedNonFacesRequest and InvokedApplicationFacesRequest. Now when I write test I only need to extend ExtendedSeamTest in order to use them. ExtendedSeamTest contains also several other convinience methods and classes. More about them later.

```
public class ExtendedSeamTest extends SeamTest {

    /**
     * Remember to invoke super.afterRequest if you override it.
     */
    public class RenderedNonFacesRequest extends NonFacesRequest {

        public RenderedNonFacesRequest() {
        }

        /**
         * @param viewId the view id of the form that was submitted
         */
        public RenderedNonFacesRequest(String viewId) {
            super(viewId);
        }

        /**
         * @param viewId         the view id of the form that was submitted
         * @param conversationId the conversation id
         */
        public RenderedNonFacesRequest(String viewId, String conversationId) {
```

```
            super(viewId, conversationId);
        }

        @Override
        protected void afterRequest() {
            assert isRenderResponseBegun();
        }
    }

    /**
     * Remember to invoke super.afterRequest if you override it.
     */
    public class InvokedApplicationFacesRequest extends FacesRequest {

        public InvokedApplicationFacesRequest() {
        }

        /**
         * @param viewId the view id of the form that was submitted
         */
        public InvokedApplicationFacesRequest(String viewId) {
            super(viewId);
        }

        /**
         * @param viewId         the view id of the form that was submitted
         * @param conversationId the conversation id
         */
        public InvokedApplicationFacesRequest(String viewId, String conversationId) {
            super(viewId, conversationId);
        }

        @Override
        protected void afterRequest() {
            assert isInvokeApplicationBegun();
        }
    }

    protected EntityManager getEntityManager() {
        return (EntityManager) getInstance(Config.ENTITY_MANAGER_NAME);
    }

    protected void login(final String username, final String password) throws Exception {
        new InvokedApplicationFacesRequest(Config.LOGIN_VIEW) {
            @Override
            protected void updateModelValues() throws Exception {
                setValue("#{credentials.username}", username);
                setValue("#{credentials.password}", password);
            }

            @Override
            protected void invokeApplication() throws Exception {
                invokeAction("#{identity.login}");
                assert (Boolean) invokeAction("#{identity.isLoggedIn}");
            }
        }.run();
    }

    protected void loginAsAdmin() throws Exception {
        login(Config.ADMIN_USERNAME, Config.ADMIN_PASSWORD);
    }

    protected void loginAsMember() throws Exception {
        login(Config.MEMBER_USERNAME, Config.MEMBER_PASSWORD);
    }

    protected void logout() throws Exception {
        new InvokedApplicationFacesRequest(Config.LOGIN_VIEW) {
            @Override
            protected void invokeApplication() throws Exception {
                invokeAction("#{identity.logout}");
                assert !(Boolean) invokeAction("#{identity.isLoggedIn}");
            }
        }.run();
    }

    protected Object getHandledException() {
        return Contexts.lookupInStatefulContexts("org.jboss.seam.handledException");
    }
```

```
    public static class Holder<T> {
        private T value;

        public T getValue() {
            return value;
        }
        public void setValue(T value) {
            this.value = value;
        }
    }
}
```

ExtendedSeamTest.java

Testing input

Take a look at Tag class :

```
@Entity
@Table(name = "TAG")
public class Tag implements Serializable {

    @Id
    @GeneratedValue
    @Column(name = "ID")
    private Long id;

    @Version
    @Column(name = "VERSION")
    private Integer version;

    @NotNull
    @Length(min = 1, max = 20)
    @Column(name = "NAME")
    private String name;


    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    public Integer getVersion() {
        return version;
    }

    private void setVersion(Integer version) {
        this.version = version;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

}
```

Id is autogenerated. Version is managed by Hibernate for optimistic locking. We do not want to test any of these properties. Name property, on the other hand, has some restrictions. It cannot be null, it must have at leas one character, but no more than twenty. This is something worth testing. How do we start ? Let's see how the add tag scenario looks like. First we enter tagList.xhtml page, press “Create” button, fill the form and submit it. Let's see how “Create” button looks like :

```
<s:button value="#{messages['CRUD.create']}" action="#{tagHome.startCreateProcess}"
                    view="/admin/tags/tag.xhtml"/>
```

Well, this is s:button, so it makes GET request. URL will look like this :

```
http://localhost:8080/seamTestArticle/admin/tags/tag.seam?
actionMethod=admin/tags/tagList.xhtml:tagHome.startCreateProcess
```

So we need to make NonFacesRequest and provide actionMethod parameter so that #{tagHome.startCreateProcess} is invoked.

Next step is to simulate form submit with FacesRequest.

```
    @Test
    public void invalidInputTest() throws Exception {
        final String name = "";
        String conversationId = new RenderedNonFacesRequest(TAG_VIEW) {
            @Override
            protected void beforeRequest() {
                setParameter("actionMethod", TAG_LIST_VIEW + ":tagHome.startCreateProcess");
            }

            @Override
            protected void renderResponse() throws Exception {
                assert (Boolean) getValue("#{tagHome.editMode}");
            }
        }.run();
        new FacesRequest(TAG_VIEW, conversationId) {

            @Override
            protected void applyRequestValues() throws Exception {
                setValue("#{tagHome.instance.name}", name);
            }

            @Override
            protected void processValidations() throws Exception {
                validateValue("#{tagHome.instance.name}", name);
            }

            @Override
            protected void afterRequest() {
                assert isValidationFailure();
                assert !isInvokeApplicationBegun();
            }
        }.run();
    }
```

In first request we can test if tagHome entered edit mode. In second request we test if validation failed (it should). One more thing to notice here is conversationId. If we want both requests to happen in one conversation then we have to pass conversationId generated by request that started conversation to following requests.

Of course there is one thing missing here. User is not logged in. Because logging in will be used in several other tests it is worth moving it to other mothod and calling that method from all tests.

```
    protected void loginAsAdmin() throws Exception {
        new InvokedApplicationFacesRequest(Config.LOGIN_VIEW) {
            @Override
            protected void updateModelValues() throws Exception {
                setValue("#{credentials.username}", Config.ADMIN_USERNAME);
                setValue("#{credentials.password}", Config.PASSWORD);
            }

            @Override
            protected void invokeApplication() throws Exception {
                invokeAction("#{identity.login}");
                assert (Boolean) invokeAction("#{identity.isLoggedIn}");
            }
        }.run();
    }
```

Now test looks like this :

```
@Test
public void invalidInputTest() throws Exception {
    final String name = "";
    loginAsAdmin();
    String conversationId = new RenderedNonFacesRequest(TAG_VIEW) {
        @Override
        protected void beforeRequest() {
            setParameter("actionMethod", TAG_LIST_VIEW + ":tagHome.startCreateProcess");
        }

        @Override
        protected void renderResponse() throws Exception {
            assert (Boolean) getValue("#{tagHome.editMode}");
        }
    }.run();
    new FacesRequest(TAG_VIEW, conversationId) {

        @Override
        protected void applyRequestValues() throws Exception {
            setValue("#{tagHome.instance.name}", name);
        }

        @Override
        protected void processValidations() throws Exception {
            validateValue("#{tagHome.instance.name}", name);
        }

        @Override
        protected void afterRequest() {
            assert isValidationFailure();
            assert !isInvokeApplicationBegun();
        }
    }.run();
}
```

Looks good, but what if we want to check more invalid data ? Let's use TestNG fascility called DataProviders[1].

```
@DataProvider(name = INVALID_INPUT_PROVIDER)
public Object[][] createInvalidInput() {
    return new Object[][]{
            new Object[]{""},
            new Object[]{null},
            new Object[]{"asdfghjkl asdfghjkl a"}
    };
}

@Test(dataProvider = INVALID_INPUT_PROVIDER)
public void invalidInputTest(final String name) throws Exception {
    loginAsAdmin();
    String conversationId = new RenderedNonFacesRequest(TAG_VIEW) {
        @Override
        protected void beforeRequest() {
            setParameter("actionMethod", TAG_LIST_VIEW + ":tagHome.startCreateProcess");
        }

        @Override
        protected void renderResponse() throws Exception {
            assert (Boolean) getValue("#{tagHome.editMode}");
        }
    }.run();
    new FacesRequest(TAG_VIEW, conversationId) {

        @Override
        protected void applyRequestValues() throws Exception {
            setValue("#{tagHome.instance.name}", name);
        }

        @Override
        protected void processValidations() throws Exception {
            validateValue("#{tagHome.instance.name}", name);
        }

        @Override
        protected void afterRequest() {
```

1 http://testng.org/doc/documentation-main.html#parameters-dataproviders.

```
            assert isValidationFailure();
            assert !isInvokeApplicationBegun();
        }
    }.run();
}
```

Now invalidInputTest will be run 3 times with empty string name, null name and "asdfghjkl asdfghjkl a" name.

> Converters defined on facelet – problem
>
> Lets assume you have defined converter for String that will trim all data sent by user. It will work fine in your application, but I do not know how to instruct SeamTest to parse facelets and apply appropriate converters.

Testing restrictions

In Seam we can restrict access to both pages and components. Seam throws NotLoggedInException if user is required to be logged in and he is not or if other authorization is required and user is not logged in.
If user is logged in then only AuthorizationException can be thrown. By default seam-gen sets pages.xml up so that redirection to either login.xhtml or error.xhtml takes place.
So how do we test this redirection ? Take a look first at the right way to do that.

```
@Test
public void enterTagViewAsNonAdmin() throws Throwable {
    new NonFacesRequest(TAG_VIEW) {
        @Override
        protected void beforeRequest() {
            setParameter("id", id.toString());
        }

        @Override
        protected void afterRequest() {
            assert !isRenderResponseBegun();
        }
    }.run();
    loginAsMember();
    new RenderedNonFacesRequest(TAG_VIEW) {
        @Override
        protected void beforeRequest() {
            setParameter("id", id.toString());
        }

        @Override
        protected void renderResponse() {
            assert getHandledException() instanceof AuthorizationException;
        }
    }.run();
}
```

Now pay attention cause here comes a tricky part. If user is not logged and login is required then after restore view phase Seam redirects to login view if it has been defined. NotLoggedInException will not be thrown unless no login view is specified (even then you will not be able to see it by looking up handledException, because contexts get cleared as soon as exception is thrown). Summarizing, best you can do is to check if renderResponse phase was invoked.

> Always setup login view in pages.xml. If you don't it will make Seam not to close transaction and get into inconsistent state in case of NotLoggedInException. All following requests, even in other test methods, will fail due to problems with transaction.

Situation gets different when user is already logged in, but he does not meet restriction. Now

AuthorizationException will be thrown and handled (redirect, if specified in pages.xml) but you will be able to verify this by looking up conversation context for "org.jboss.seam.handledException". However there is one important thing to notice.

> If you expect AuthorizationException to happen elsewhere then during invokeApplication phase then do not use FacesRequest ! Again, it will drive Seam into inconsistent state.

This is why we used NonFacesRequest – AuthorizationException will be thrown due to pages.xml setup, so right after restore view phase.

Testing for EntityNotFoundException

This is very easy. Here we test (first request) what will happen if there is no object in database with id given by user in URL.

```
    @Test
    public void enterNonExistingTagView() throws Throwable {
        loginAsAdmin();
        new RenderedNonFacesRequest(TAG_VIEW) {
            @Override
            protected void beforeRequest() {
                setParameter("id", "-1");
            }

            @Override
            protected void renderResponse() {
                assert getHandledException() instanceof EntityNotFoundException;
            }
        }.run();
        new RenderedNonFacesRequest(TAG_VIEW) {
            @Override
            protected void renderResponse() {
                assert getHandledException() instanceof EntityNotFoundException;
            }
        }.run();
    }
```

Second request tests what will happen if user enters page without giving any parameters. I guess the code is self explanatory.

Component testing

Securing pages is one thing. But sometimes at first you plan to use component on one page, you secure the page, but after a while you use the same component on the other page. Now shilding is gone. It is good to secure components themselves. You do it by adding @Restrict annotations to components, or methods. There is easy way to test this disregarding pages.

```
    @Test
    public void componentSecurity() throws Throwable {
        new ComponentTest() {
            protected void testComponents() throws Exception {
                try {
                    invokeMethod("#{tagHome.persist}");
                    assert false;
                } catch (Exception e) {
                    assert e.getCause() instanceof NotLoggedInException;
                }
                try {
                    invokeMethod("#{tagHome.update}");
                    assert false;
                } catch (Exception e) {
                    assert e.getCause() instanceof NotLoggedInException;
                }
            }
        }.run();
```

```
        loginAsMember();
        new ComponentTest() {
            protected void testComponents() throws Exception {
                try {
                    invokeMethod("#{tagHome.persist}");
                    assert false;
                } catch (Exception e) {
                    assert e.getCause() instanceof AuthorizationException;
                }
            }
        }.run();
    }
```

ComponentTest gives you initialized contexts and not much more, but it is usefull anyway.

Tests ordering

Sometimes one tests depend on one another. This is not SeamTest feature, but I will mention it here.

```
    @Test
    public void createTag() throws Exception {…}

    @Test(dependsOnMethods = {"createTag"}, groups = TAG_DEPENDANT_TESTS)
    public void editTag() throws Exception {…}

    @Test(dependsOnMethods = {"createTag"}, groups = TAG_DEPENDANT_TESTS)
    public void cancelEditTag() throws Exception {…}

    @Test(dependsOnGroups = TAG_DEPENDANT_TESTS)
    public void removeTag() throws Exception {…}
```

Here we have four tests. The editTag(), cancelEditTag() and removeTag() depend on createTag(), for they operate on tag created by that test. Additionally removeTag must not be executed before both editTag() and cancelEditTag(). We achieve this by making editTag(), cancelEditTag() to depend on createTag() method. We assign both of them to TAG_DEPENDANT_TESTS group and make removeTag() to depend on that group.

I do not encourage you to use test ordering. I guess it would be more clear to provide appropriate date for each test method in import-test.sql.

IDEA integration

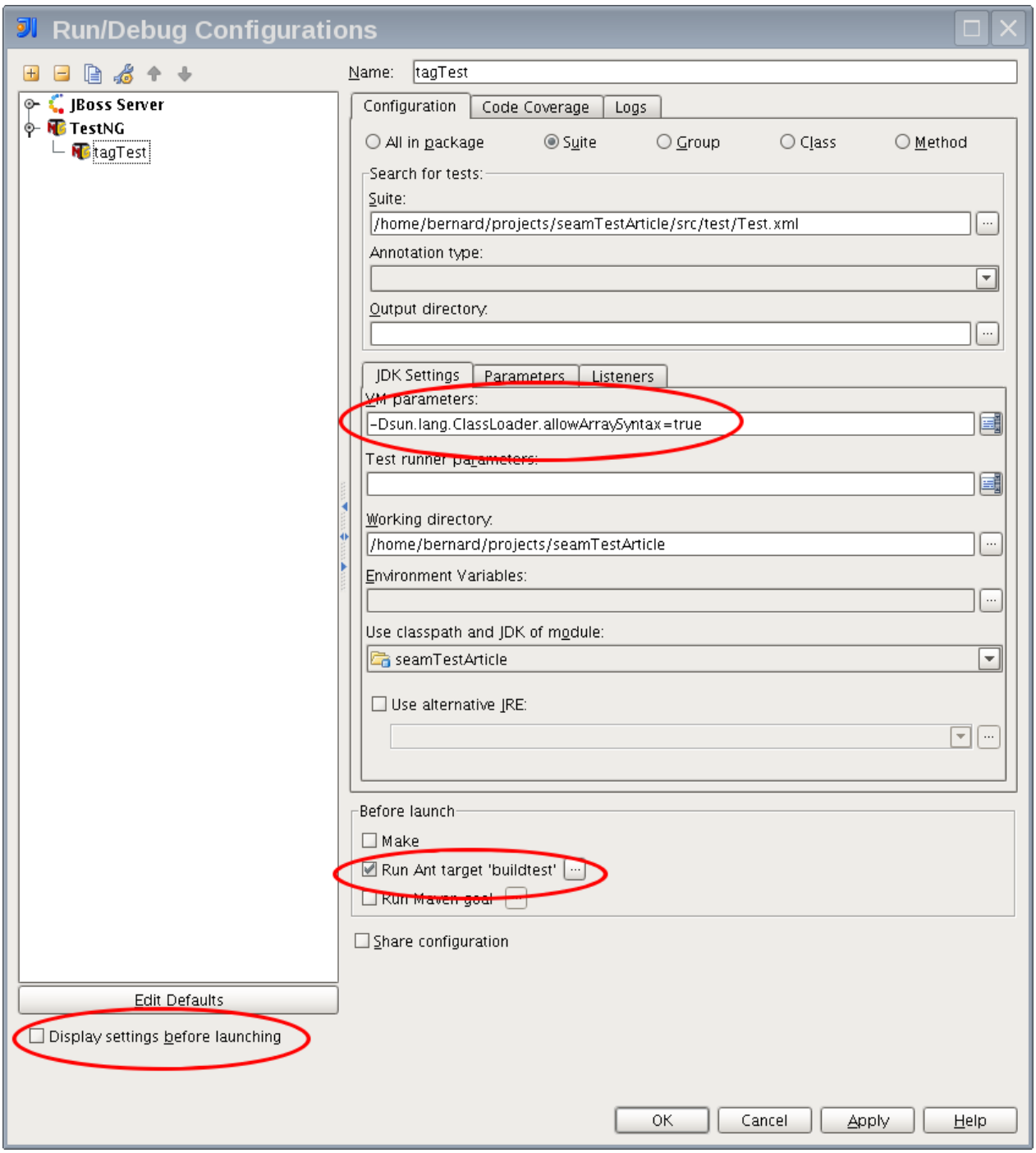
My beloved IDE is IntelliJ IDEA, which has pretty good Seam support actually. That's why I just can't skip this part.

Seam-gen generates project with IDEA config files, but they are not perfect. We need to add libraries from lib/test directory to project dependencies.

Next setup ant build by adding ant build.xml

Next thing is that when you create Run/Debug test configuration you have to provide following VM parameters if you use JVM6 :

```
-Dsun.lang.ClassLoader.allowArraySyntax=true
```


Run/Debug Configurations
JBoss Server
TestNG
tagTest
Name: tagTest
Configuration
Code Coverage
Logs
All in package
Suite
Group
Class
Method
Search for tests:
Suite:
/home/bernard/projects/seamTestArticle/src/test/Test.xml
Annotation type:
Output directory:
JDK Settings
Parameters
Listeners
VM parameters:
-Dsun.lang.ClassLoader.allowArraySyntax=true
Test runner parameters:
Working directory:
/home/bernard/projects/seamTestArticle
Environment Variables:
Use classpath and JDK of module:
seamTestArticle
Use alternative JRE:
Before launch
Make
Run Ant target 'buildtest'
Run Maven goal
Share configuration
Edit Defaults
Display settings before launching
OK
Cancel
Apply
Help