Internet Application Project: Java Server Side

420-625-DW

“Unit testing” in a container

Today

-A few hints to address “gotchas” with authentication issues

-Arquillian for unit testing

Last Class

How to set up login

-Create a “realm” for the database

-Create roles inside of configuration files

-Map url patterns to certain roles

More about realms

There are many sorts of realms. We are primarily interested in using realms based on a database (since we wish to be able to add users easily)

You can also configure realms based on a hard coded list.

Generally speaking, you can switch between realms based on configuration only within the web.xml file

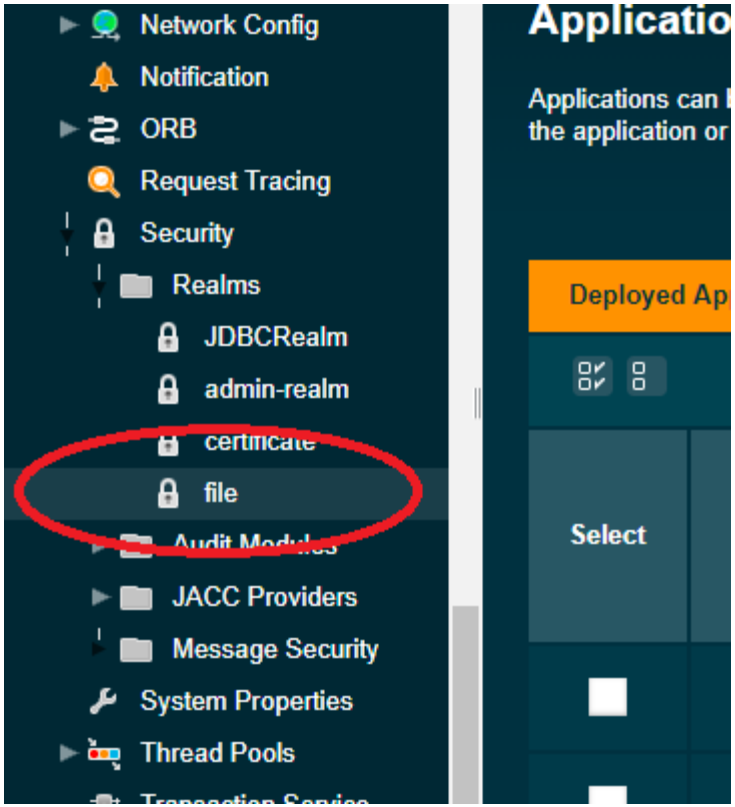
More about realms

The file realm within glassfish is a place you can enter user names and groups there.

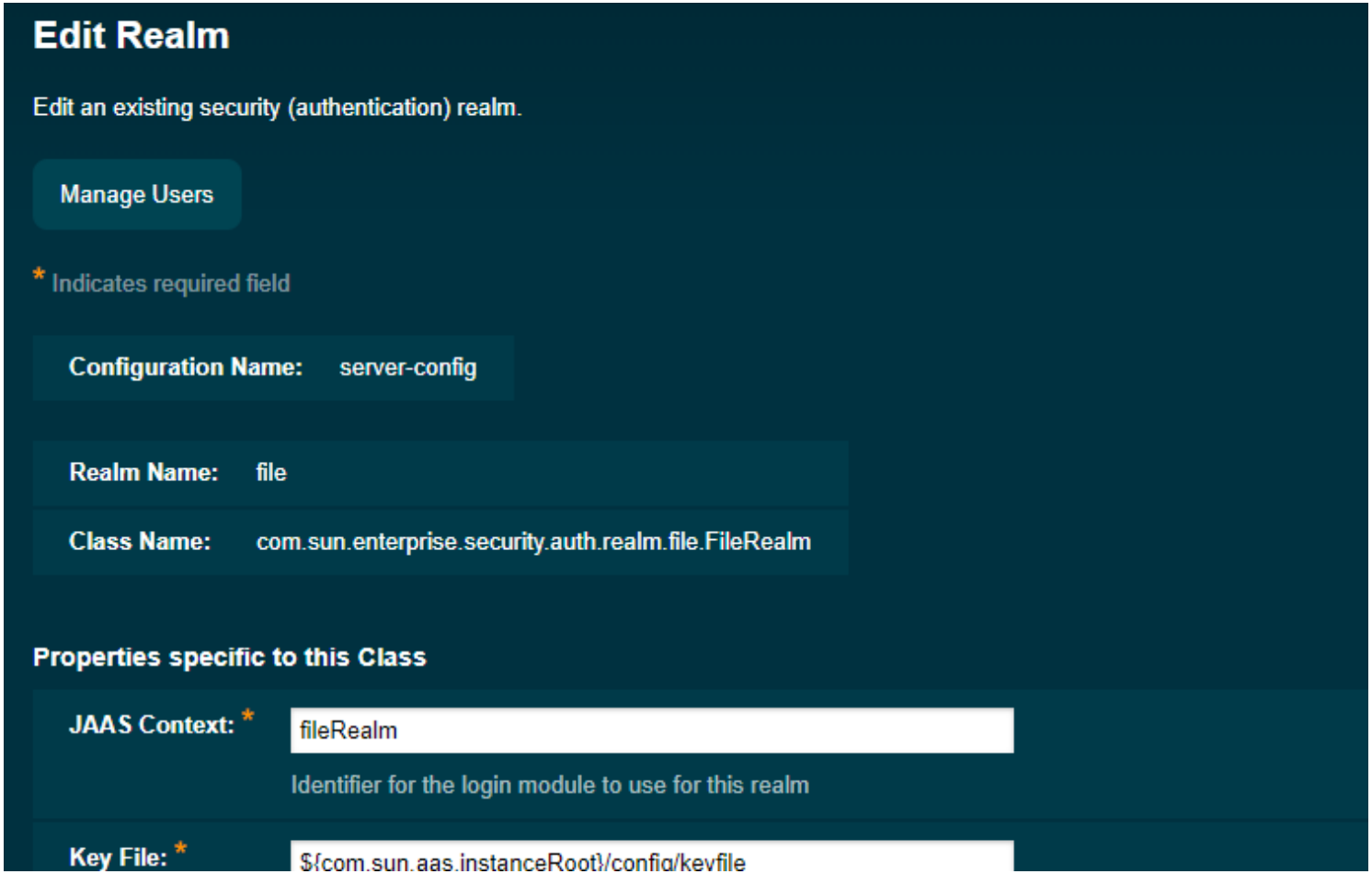
If you wish to use that as your security realm instead, you simply put that into the web.xml

```
 <login-config>
      <auth-method>FORM</auth-method>
      <realm-name>file</realm-name>
...
```

It's useful to use *file* while debugging because you can manually add entries within Glassfish (there are fewer moving parts)



Modifying users on the server



Database realms: Double check!

-JAASContext *must* be jdbcRealm

-The realm name must line up with what you used in your web.xml file (note: we will all share the same *server* so your realms must be distinct names!)

-Digest algorithm does *not* default to no encryption

jdbcRealm

The jdbcRealm is a class defined for you (and installed on glassfish when you add in the sqlconnector)

It looks at the various properties to figure out what tables to use.

Supports sha-256 encoding, but does not support salting. If you enable sha-256 encoding, then that means the passwords on the server side will be hashed automatically.

If you want to perform salting, you need to apply the hash to the salt/password yourself in the code (not recommended for the project because this will make it harder to use the j_security_check)

Database realms: Double check!

JNDI needs to lineup with that name of what you called this resource in the glassfish-resources.xml file:

<jdbc-resource enabled="true" jndi-name="**jdbc/myDatasource**" object-type="user" pool-name="connectionPool">

This is how it knows what DB info to use!

Front end clarifications

Last class we saw:

```
<form method="POST" action="j_security_check">
        Username: <input type="text" name="j_username"
/>
        Password: <input type="password"
name="j_password" />
        <br />
        <input type="submit" value="Login" />
        <input type="reset" value="Reset" />
    </form>
```

Front end clarifications

The previous code is *not* using primefaces. Although you *can* put the code within a primefaces panel, it is not required.

j_security_check is an action that is defined as *part of the container*.

You are using the containers login servlet to login.

After login is successful

After the user logs in successfully, they will be redirected back to the page they were initially requesting.

At this point, you can retrieve the user information in a few ways.

From the backend:

```
FacesContext context = FacesContext.getCurrentInstance();
        HttpServletRequest request = (HttpServletRequest) context.getExternalContext().getRequest();
```

Then either

```
request.getRemoteUser();
```

OR

```
request.getUserPrincipal().getName());
```

From the front end (“quick and dirty”)

request is an implicit object in JSF (just like it was in JSP)

You can refer to it from the jsf as:

#{request.remoteUser}

It is better to use a bean with this property though (for testing and general maintainability purposes)

Other methods

There are a couple other useful methods that you can perform on a request object:

-request.login(username,password)

This method will attempt to login based on the given username/password combination. It will look at the realm that is configured in the web.xml file

-request.logout()

Self explanatory ☺

-request.isUserInRole(rolename)

Checks if the logged in user belongs to a specific role or not

“Unit” Testing with Arquillian

Why the quotes Dan?

First of all, why do we have quotes around “unit” testing?

The tools we will look at are very powerful tools.

They will allow us to *automatically* deploy to our container and run things in our real environment.

This is extremely powerful as often it can take a really long time to manually test things in a real environment.

More than a unit

But they aren't really a single *unit* in the sense that we typically think of unit tests as testing very small.

"UNIT TESTING is a level of software **testing** where individual **units**/ components of a software are tested. The purpose is to validate that each **unit** of the software performs as designed. **A unit is the smallest testable part of any software.** It usually has one or a few inputs and usually a single output."

src: http://softwaretestingfundamentals.com/unit-testing/

More than a unit

Of course our units are often fairly large. If we have objects using objects using objects, we will certainly want to test the composite objects. However, we can still think of a unit as an individual method.

But there need to be some automated tests at the lowest possible level!

Moral

When a class X uses another object/method, you need to check that class X works under all possible values that the method returns.

You should have 1 case for each possible value that the method returned.

You should have a separate test suite for checking that the method returns the correct things at the right time.

Example: “Component test” vs “Unit test”

Component tests can be thought of as a bigger, more “end to end”, test of whether things work properly:

Example: Suppose we have a register user API, we wish to test that the backend api works well.

-Idea 1: Access an API via a post request. Does it return the expected value? Does the database update as we expect?

This way of thinking is very high level and very black box. This is an important step to have and it is what arquillian (and selenium) will help with! This is essentially a *component* test. We should definitely have these. But we need to make sure we are testing the write granularity of code

Unit test : For low granularity

That api probably relies on many different helper methods. These helper methods **should be tested independently from the higher level component**

For example, if there is a method *verifyPhoneNumber(String s)* that checks the phone number satisfies a certain format, we need to check that separately, without calling the API!

Unit test

```
public void testPhoneValidation() {
        PhoneValidator validator = new PhoneValidator();
        Assert.isTrue(validator.validate(“(123)456-7890”));
        Assert.isFalse(validator.validate(“(asddads”));
        …. /*these could even be in separate tests….*/
}
```

These tests will be portable and will work in any code base if you migrate. They do not rely on the API whatsoever. Notice how short it is even if you wanted to add 100 different phone number cases.

The whole test probably runs in under 1 second

What the component test should check

```
public void testApiBadPhone() {
        HttpPostRequest request = new HttpPostRequest();
        request.addParameter(“phone”, “invalid”);
        Assert.isTrue(request.send().getResponseCode(), 200);
        }
```

It is useful to test the api with a bad phone number as well. **This checks that the api properly calls the validator you wrote. It should not be used for validating if the validator itself works.** It is simply establishing that somewhere in the api you wrote validator.validate(what was given as input);

What the component test should NOT check

```
public void testApiBadPhone() {
        HttpPostRequest request = new HttpPostRequest();
        request.addParameter(“phone”, “invalid”);
        Assert.isTrue(request.send().getResponseCode(), 200);
        }
```

What you don’t want to do is check for *every* possible case of the validator whether it works or not. This would be unnecessary overhead and will make it extremely difficult to debug. Your unit tests will be come unmanageable.

In container or not in container?

We want to have some tests running in the container to test the real world environment. But we also need to have tests that do not run in a container for testing our lower level methods/objects.

These low level automated tests *should not,* primarily for performance reasons, be run in the actual container. It adds unnecessary complexity to the code which makes it harder to maintain.

Why do we need to distinguish?

-Remember that we are having our containers do some work for us

-Specifically the injection. Also potentially some of the login information

-We need to do some work to get the test environment to work properly

Option 1:

- One option is to ensure that you do all your injections via *constructor* injection.
  - Recall that this means it will do the same thing in production code, but your unit test can work based on providing the injection

Constructor Injection

```
public class Bean {
        private SomeInjectable val;
        public Bean(@Inject SomeInjectable val) {
                this.val = val;
        }
        public Bean() { } // require to ensure this is also a Bean
}
```

This code behaves identically in the production code to using field injection

Unit test

In the unit test though, we have the ability to pass to the constructor directly an object.

In other words, we still *can* create an object using the new operator:

```
public void test() {
        SomeInjectable toPass = new SomeInjectable();
        Bean b = new Bean(toPass);
        // and now do whatever test you need to do
}
```

Injecting @PersistenceContext

Unfortunately, the @PersistenceContext, although behaving similarly to @Inject, is not the same

In fact you cannot inject the entitymanager via the constructor this way.

What to do for unit tests?

For unit testing purposes:

-You could overload the constructor (one version for production + different version for testing) → not ideal. Testing different code paths!

-You could add a setEntityManager() method → still not quite ideal, but most of the code is the same now

-Or option 3: *use arquillian* to run the unit tests in the container

Arquillian – Definition #1 from MIB

- Arquillians are a small alien species.
- They wear a Human-sized mechanical suit, which is used for both protection and locomotion (also hiding their true form).
- In the movie Men in Black the MiB need to find the galaxy and return it to the Arquillians, who are threatening to annihilate Earth if it is not returned within a certain time period.

Arquillian – Definition #2 from JBoss

- A testing platform for the JVM that enables developers to create automated integration, functional and acceptance tests for Java middleware
  - Manages the lifecycle of the container
  - Bundles the test case, dependent classes and resources into a ShrinkWrap archive
  - Deploys the archive to the container
  - Enriches the test case by providing dependency injection and other declarative services
  - Executes the tests inside (or against) the container

Two Testing Environments – 1: Embedded

- Embedded
  - A special version of the server designed to be embedded in other applications
  - Arquillian uses this version of the container to run your tests
  - Most if not all Java servers have an embedded version for testing
  - The embedded version runs in the same JVM as the development environment

Two Testing Environments – 2: Remote

- Remote
  - Functioning server is used to run the tests
  - If the remote server is started by NetBeans then it runs in the same JVM
  - If the remote server is started separately from NetBeans then it runs in its own JVM

Arquillian and Maven

- two sections:

1) DependencyManagement
2) Regular dependencies

A complete file is posted on Lea along with these notes

<dependencyManagement> (outside of other tags)

```
<dependencyManagement>
    <dependencies>
      <dependency>
        <groupId>org.jboss.arquillian</groupId>
        <artifactId>arquillian-bom</artifactId>
        <version>1.2.0.Final</version>
        <scope>import</scope>
        <type>pom</type>
      </dependency>

      <dependency>
        <groupId>org.jboss.shrinkwrap.resolver</groupId>
        <artifactId>shrinkwrap-resolver-bom</artifactId>
        <version>3.1.2</version>
        <type>pom</type>
        <scope>import</scope>
      </dependency>

    </dependencies>
  </dependencyManagement>
```

Plus several regular dependencies

```
<dependency>
        <groupId>org.jboss.arquillian.container</groupId>
        <artifactId>arquillian-glassfish-remote-3.1</artifactId>
        <version>1.0.2</version>
        <scope>test</scope>
    </dependency>

    <!-- Resolves dependencies from the pom.xml when explicitly referred to
    in the Arquillian deploy method -->
    <dependency>
        <groupId>org.jboss.shrinkwrap.resolver</groupId>
        <artifactId>shrinkwrap-resolver-depchain</artifactId>
        <type>pom</type>
        <scope>test</scope>
    </dependency>

    <!-- Connects Arquillian to JUnit -->
    <dependency>
        <groupId>org.jboss.arquillian.junit</groupId>
        <artifactId>arquillian-junit-container</artifactId>
        <scope>test</scope>
    </dependency>

    <!-- JUnit dependency -->
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.12</version>
        <scope>test</scope>
    </dependency>
```

How to

- Before your unit test class, add the annotation @RunWith(Arquillian.class)
- Arquillian will now search for a *static* method in your code that has the annotation @Deployment before it

- This method should return a JavaArchive (non web) or a WebArchive (web) as in our case. We will use a class called ShrinkWrap to build the archive for us

Shrinkwrap

- Shrinkwrap is a class that creates the war file for the container in which your code will execute for the purpose of unit testing
- It needs to be told of the CDI beans and other configuration features
- Files that must be added to the Shrinkwrap war file are defined using the resolver.

Example deploy (JAR)

```
@Deployment
   public static JavaArchive createDeployment() {
       JavaArchive result =
ShrinkWrap.create(JavaArchive.class)
           .addClasses(Greeter.class, AdditionalClass.class)
           .addAsManifestResource(EmptyAsset.INSTANCE,
"beans.xml");
      System.out.println(result.toString(true));
       return result;
   }
```

Example deploy (WAR)

```
@Deployment
  public static WebArchive createDeployment() {

final File[] dependencies = Maven.resolver()
    .loadPomFromFile("pom.xml")
    .resolve("org.assertj:assertj-core")
    .withoutTransitivity()
    .asFile();

final WebArchive webArchive = ShrinkWrap.create(WebArchive.class, "test.war")
   .setWebXML(new File("src/main/webapp/WEB-INF/web.xml"))
   .addPackage(PatientJpaController.class.getPackage())
   .addPackage(RollbackFailureException.class.getPackage())
   .addPackage(Patient.class.getPackage())
   .addAsWebInfResource(EmptyAsset.INSTANCE, "beans.xml")
   .addAsWebInfResource(
      new File("src/main/webapp/WEB-INF/glassfish-resources.xml"),
      "glassfish-resources.xml")
   .addAsResource(new File("src/main/resources/META-INF/persistence.xml"),
      "META-INF/persistence.xml")
                .addAsResource("createHospitalTables.sql")
                .addAsLibraries(dependencies);
}
```

arquillian.xml

- Arquillian creates a war file that it deploys to the server
- It needs credentials to perform the deploy
- Must be in the root of the classpath

```
<?xml version="1.0"?>
<arquillian xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"
            xmlns="http://jboss.org/schema/arquillian"

xsi:schemaLocation="http://jboss.org/schema/arquillian
    http://jboss.org/schema/arquillian/arquillian_1_0.xsd">

    <container qualifier="glassfish" default="true">
        <configuration>
            <property name="adminUser">admin</property>
            <property name="adminPassword"></property>
        </configuration>
    </container>

</arquillian>
```

Useful tutorial

http://arquillian.org/guides/getting_started/