CERN

Software Testing

Basics and integration testing of J2EE applications

Software Testing

Basics and integration testing of J2EE applications

What do we mean by testing?

- Software testing has many flavors: unit testing, integration testing, functional testing, stress testing, black/white box, monkey testing...
- Not all tests are created equal
- The basic purpose of tests is to provide verification and validation of software

Validation versus verification

- Verification: did we build the software right? Does it fulfill its original requirements?
- Validation: did we build the right software? Does the project meet the users’ needs?
- Unit, integration tests, static analysis provide verification
- User acceptance tests (UAT) provide validation

Code coverage

- One of the measures used for systematic software testing
- Tries to answer the question: how much of my code is run in my tests?
- The assumption is:

the more code is executed

$\leftrightarrow$

the smaller the chance is of a bug emerging

Code coverage

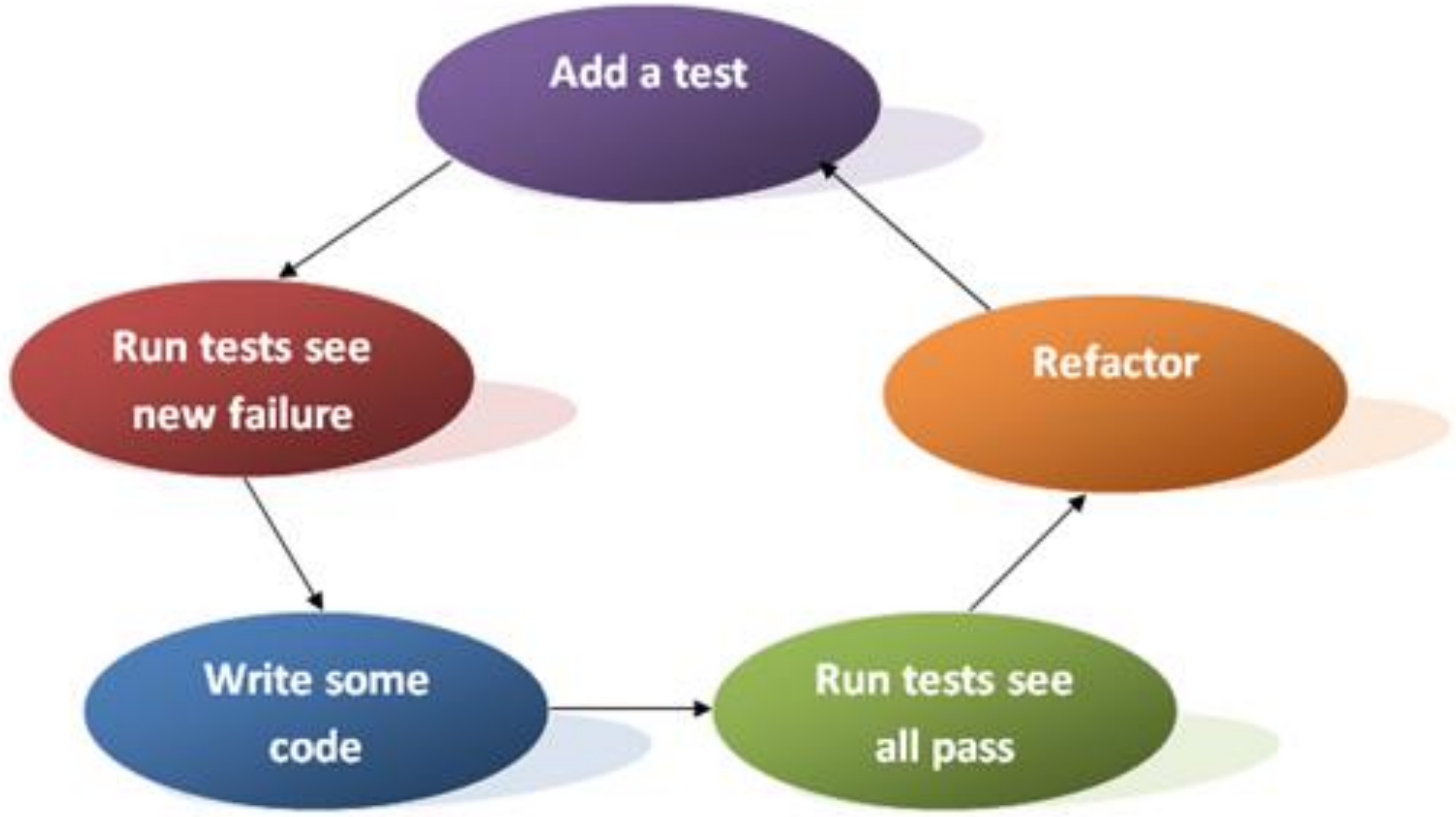
- Of course, a 100% code coverage doesn’t mean 100% working software
- A talented developer can have multiple bugs in fully covered code – the construction of the test cases matters

Test Driven Development (TDD)

- A methodology of developing code which is based on writing tests first
- The TDD mantra is:

red ➔ green ➔ refactor ➔ red...

The TDD Process



Pros and cons

- Forces modularity and testability of the code
- Induces confidence among team members (even if you break something, the tests will let them know)
- Obvious mistakes are harder to commit
- Forces good code coverage

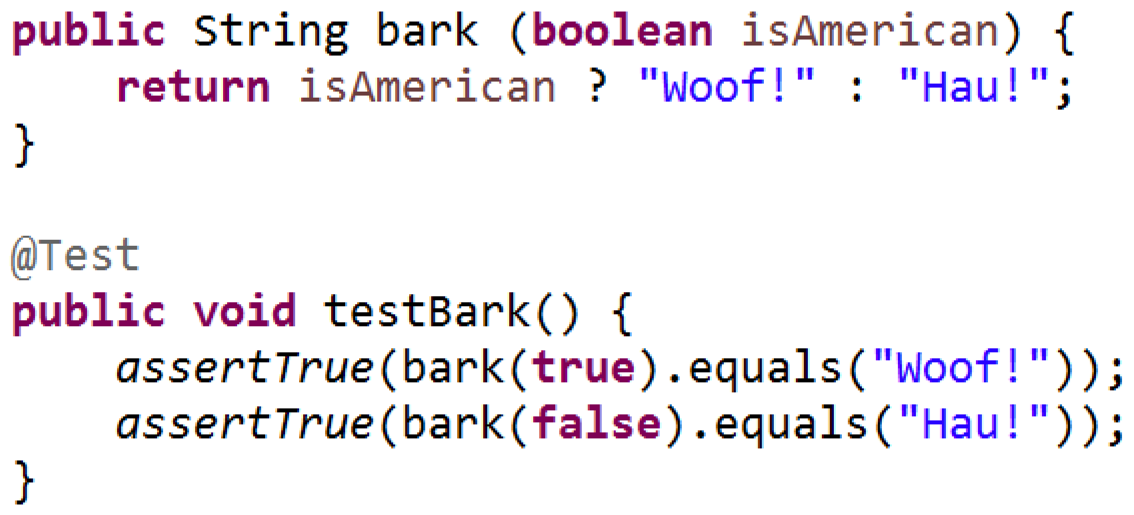
Conclusion: TDD is good

- Tests can be hard to write
- Adds a layer of complexity which needs to be maintained
- Gives false confidence if the test is badly written

Conclusion: TDD is bad

Unit tests

- Low-level
- Usually testing one method/class, ideally limiting dependencies on other methods/classes and components



```java
public String bark (boolean isAmerican) {
    return isAmerican ? "Woof!" : "Hau!";
}

@Test
public void testBark() {
    assertTrue(bark(true).equals("Woof!"));
    assertTrue(bark(false).equals("Hau!"));
}
```

Integration tests

- Take into account the broader context of the application (database, external APIs, filesystem, concurenncy
- Verify that the – ideally unit tested - building blocks (methods, classes, modules etc), work well together.
- Unlike unit tests, which should run at build-time and should be very quick to run, integration tests require a deployment step
- Usually will touch significantly more code than unit tests

Humor

2 UNIT TESTS, 0 INTEGRATION TESTS
via reddit.com/r/programmerhumor

Integration tests in practice

- Java EE or Spring applications can make heavy use of dependency injection or other container services
- Whether an application works well within the container is crucial, since the containter provides the implementation for the services defined e.g. in the JEE standard
- The system configuration is equally as important (have we moved everything from DEV to TEST?)

Arquillian

- *De facto* industry standard for J2EE integration testing; also works with Spring, although it’s not as popular
- Allows to run integration tests within the context of a fully configured application server as a build step
- Arquillian Warp is an extension which allows to use a client-side testing framework and to inspect server state at the same time
- Works well with Junit

Arquillian test

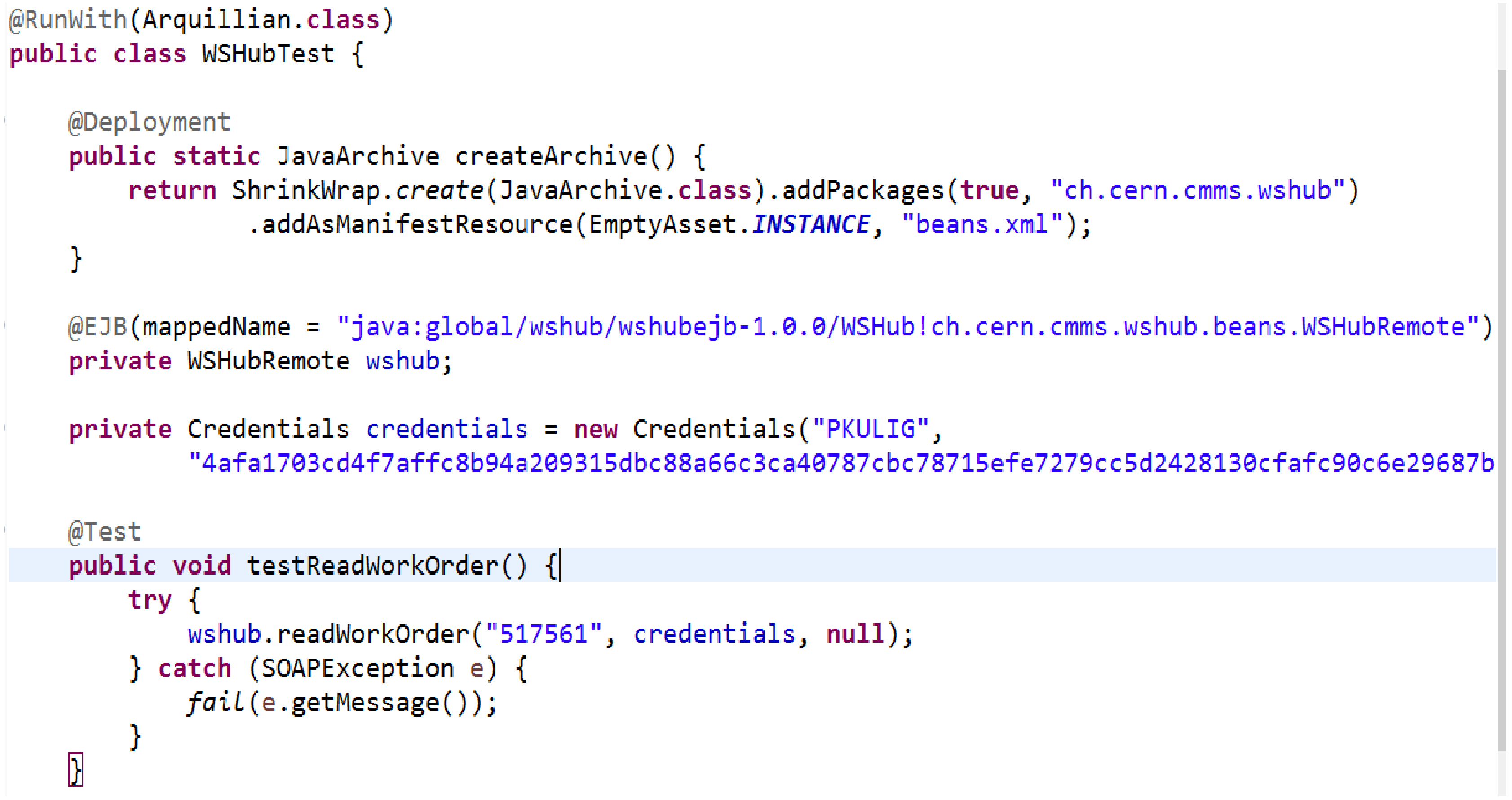
1. Start an instance of the container
2. Deploy an archive to the server.
3. Run the test cases (JUnit/TestNG integration)
4. Undeploy the archive and kill the instance.

The configuration of the application server is preserved. Running the test on a running instance of a server is as simple as

```
mvn clean test
```

including the deployment and undeployment of the test package.

How a test is written

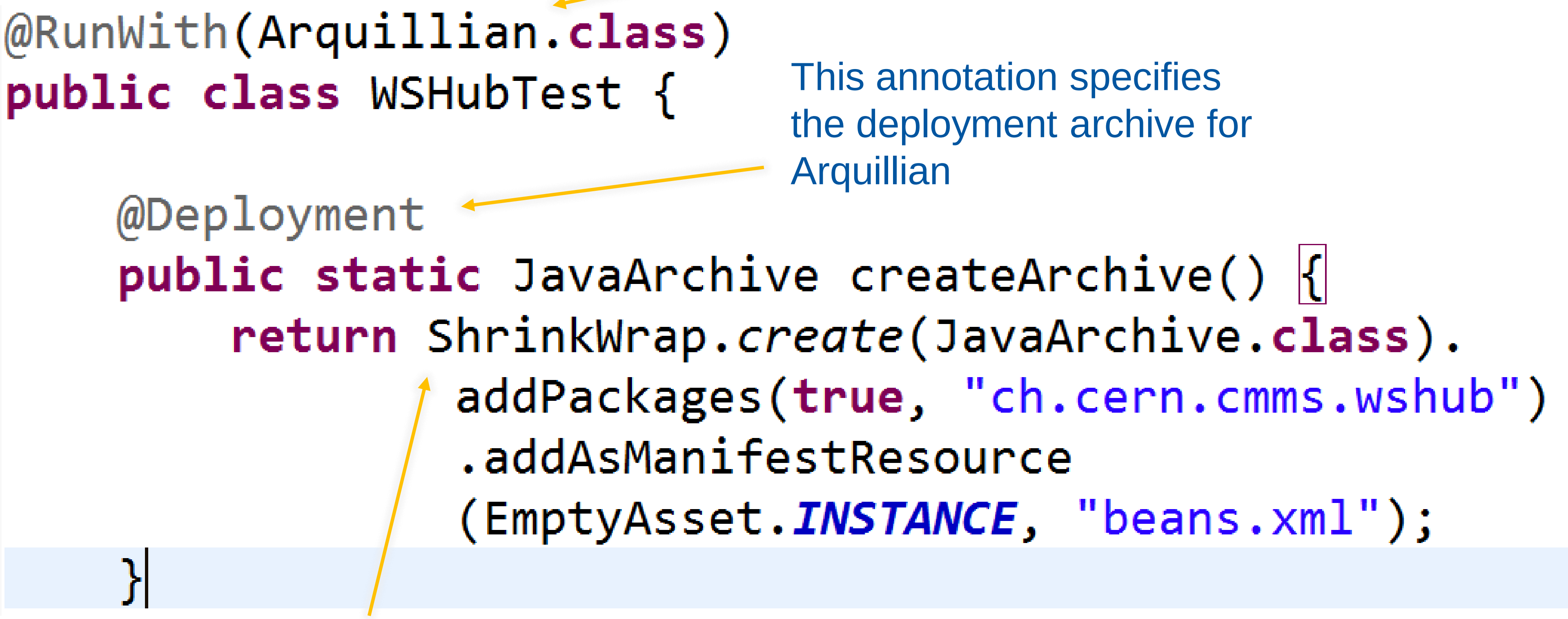


```java
@RunWith(Arquillian.class)
public class WSHubTest {

    @Deployment
    public static JavaArchive createArchive() {
        return ShrinkWrap.create(JavaArchive.class).addPackages(true, "ch.cern.cmms.wshub")
                .addAsManifestResource(EmptyAsset.INSTANCE, "beans.xml");
    }

    @EJB(mappedName = "java:global/wshub/wshubejb-1.0.0/WSHub!ch.cern.cmms.wshub.beans.WSHubRemote")
    private WSHubRemote wshub;

    private Credentials credentials = new Credentials("PKULIG",
            "4afa1703cd4f7affc8b94a209315dbc88a66c3ca40787cbc78715efe7279cc5d2428130cfafc90c6e29687b

    @Test
    public void testReadWorkOrder() {
        try {
            wshub.readWorkOrder("517561", credentials, null);
        } catch (SOAPException e) {
            fail(e.getMessage());
        }
    }
```

Let’s dissect it…

This JUnit annotation tells the engine to use a different runner than the default

This annotation specifies the deployment archive for Arquillian

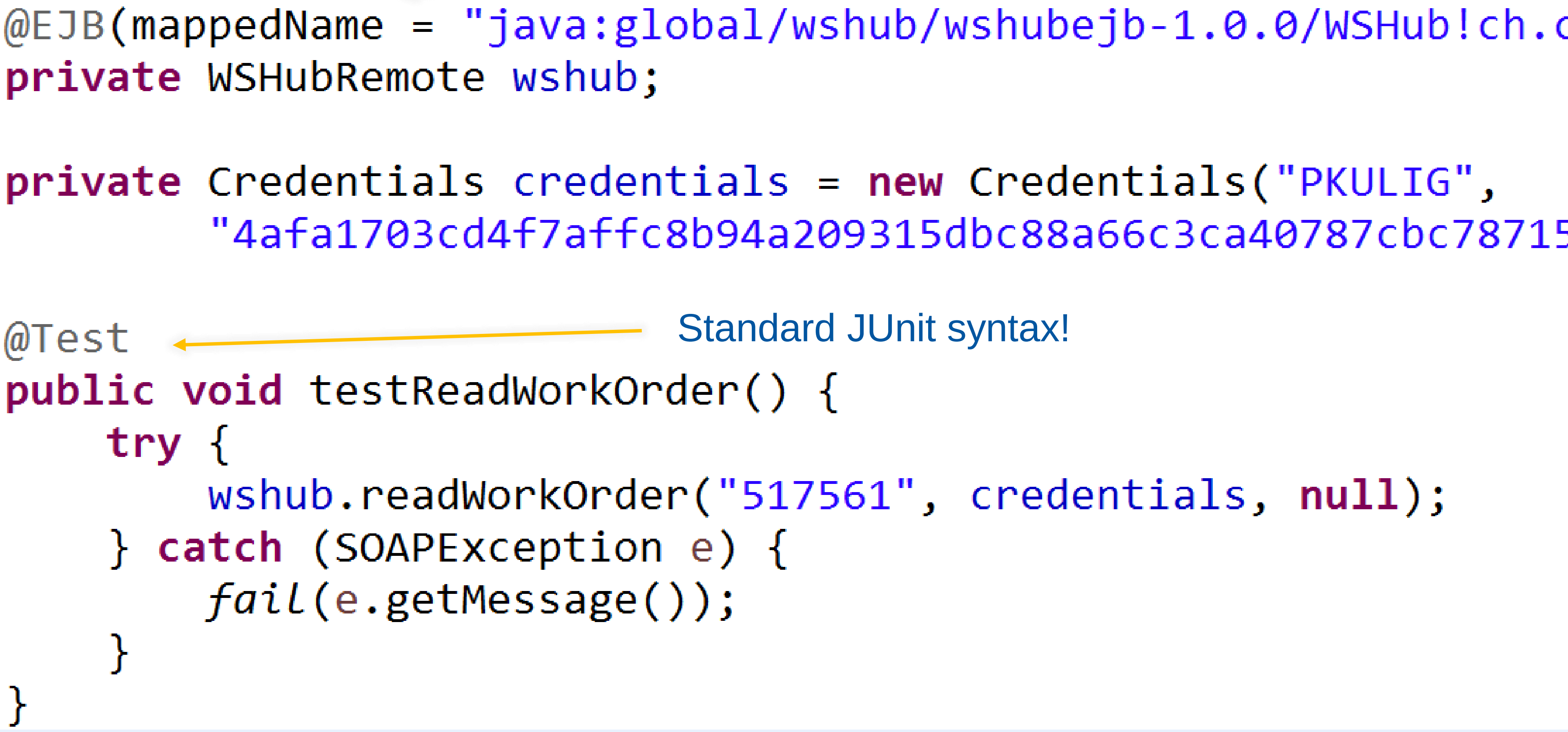


```
@RunWith(Arquillian.class)
public class WSHubTest {

    @Deployment
    public static JavaArchive createArchive() {
        return ShrinkWrap.create(JavaArchive.class).
                addPackages(true, "ch.cern.cmms.wshub")
                .addAsManifestResource
                (EmptyAsset.INSTANCE, "beans.xml");
    }
```

Using ShrinkWrap to package our classes into a web archive
It’s also possible to use a Maven dependency resolver – no need to specify any classes or packages, they are read from the pom.xml

Let's dissect it…

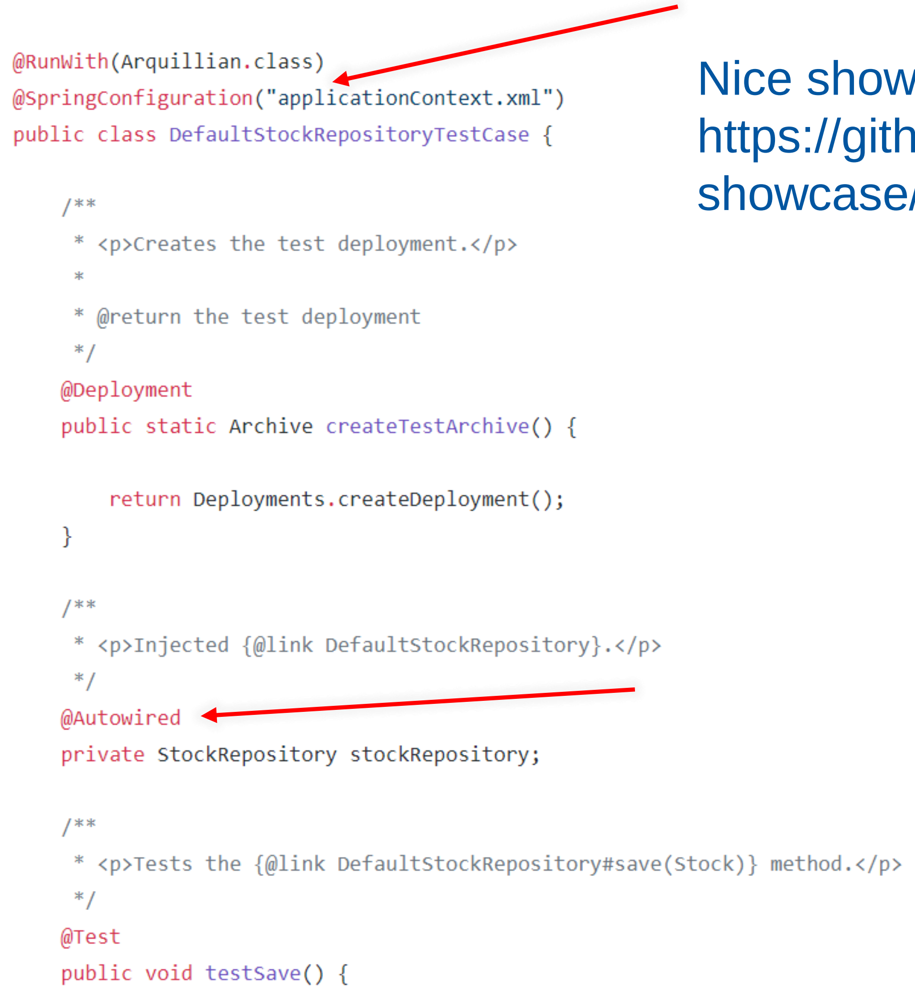
Ordinary EJB annotation to lookup a bean in JNDI



```
@EJB(mappedName = "java:global/wshub/wshubejb-1.0.0/WSHub!ch.c
private WSHubRemote wshub;

private Credentials credentials = new Credentials("PKULIG",
        "4afa1703cd4f7affc8b94a209315dbc88a66c3ca40787cbc78715

@Test
public void testReadWorkOrder() {
    try {
        wshub.readWorkOrder("517561", credentials, null);
    } catch (SOAPException e) {
        fail(e.getMessage());
    }
}
```

Standard JUnit syntax!

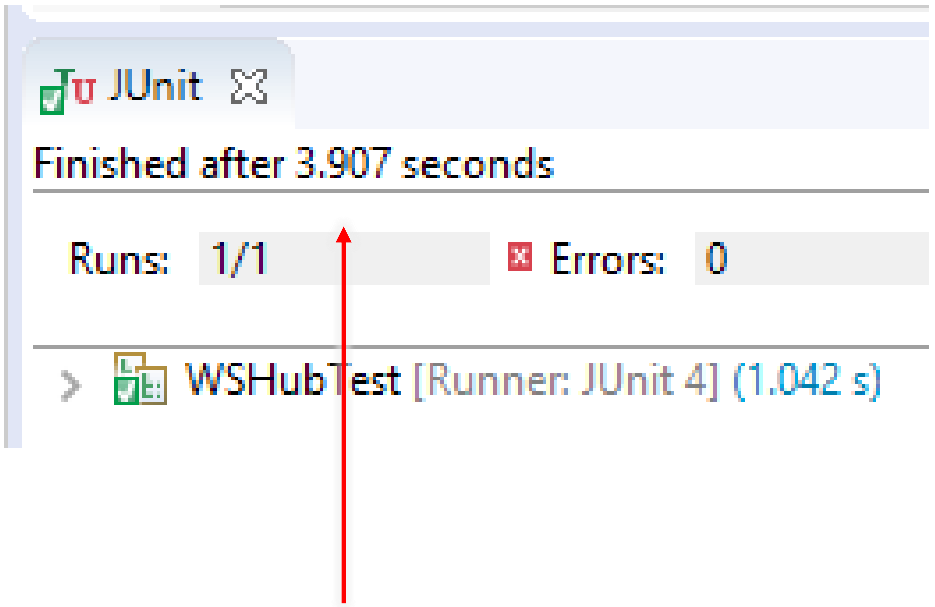
Also possible with Spring...



```
@RunWith(Arquillian.class)
@SpringConfiguration("applicationContext.xml")
public class DefaultStockRepositoryTestCase {

    /**
     * <p>Creates the test deployment.</p>
     *
     * @return the test deployment
     */
    @Deployment
    public static Archive createTestArchive() {

        return Deployments.createDeployment();
    }

    /**
     * <p>Injected {@link DefaultStockRepository}.</p>
     */
    @Autowired
    private StockRepository stockRepository;

    /**
     * <p>Tests the {@link DefaultStockRepository#save(Stock)} method.</p>
     */
    @Test
    public void testSave() {
```

Nice showcase: https://github.com/arquillian/arquillian-showcase/tree/master/spring

Surely it takes long!



JUnit

Finished after 3.907 seconds

Runs: 1/1 Errors: 0

WSHubTest [Runner: JUnit 4] (1.042 s)

This time includes the packaging, deployment, running the aforementioned tests and undeploying the package.

Arquillian Warp

- „Fills the void between client-side and server-side testing”
- Allows to mock client-side actions in order to inspect server state
- Can be used with e.g. Selenium WebDriver
- Example coming….

```
@Test
@InSequence(2)
public final void browserNewMerchant() throws Exception {
    Warp
            .initiate(new Activity() {
                @Override
                public void perform() {
                    WebElement txt = browser.findElement(By.id("form:txt"));
                    txt.sendKeys("sema index 1");

                    WebElement btn = browser.findElement(By.id("form:btn"));
                    guardAjax(btn).click();
                }
            })
            .observe(request().header().containsHeader("Faces-Request"))
            .inspect(new Inspection() {
                private static final long serialVersionUID = 1L;
```

Arquillian

- Configuration is somewhat non-trivial and is done in XML files; however, for J2EE projects there are archetypes which provide Arquillian support „out of the box”
- Sharing the project among team members requires to set an environmental variable
- Can be run with `maven test` - easy to integrate into a Maven build

CERN