Enterprise Programmering 1

Lesson 04: EJB

Prof. Andrea Arcuri

About these slides

- These slides are just high level overviews of the topics covered in class
- The details are directly in the code comments on the Git repository

Enterprise Java Bean (EJB)

- An EJB is just a Java class annotated with a special tag
  - *@Stateless*, *@Stateful* and *@Singleton*
- When an EJB is run in a JEE container (*WildFly*, *GlassFish*, etc), the container will *enhance* it with special functionalities
- Example: by default, each EJB method is executed inside a *transaction*
  - so, don't need to explicitly call *begin()* and *commit()* on an *EntityManager*
  - EJB reduces boilerplate

EJB Enhancements

- JEE EJB enhancements are based on 2 main properties
- Those are not only for JEE
- *Dependency Injection*: the container will automatically add the dependencies the EJB needs
- *Proxy Class*: container does not return instances of EJBs, but create subclasses with the enhanced functionalities (where method calls are proxied to the actual EJB instances which are inside the proxy)

Dependency Injection by Reflection

```
@Stateless
public class UserBean {

    @PersistenceContext
    private EntityManager em;

    public UserBean(){}
```

- For "*em*", no input for constructor, and no setter
- JEE container will automatically *inject* the current active "*em*"
- EJB just needs to declare the dependency as a field… how it is created and injected is a job for the container…

Java Reflection

- In Java (not just JEE) each object instance keeps information of its declaring class
- Info of the class can be queried at runtime:
  - methods, fields, annotations, etc.
- Fields can be modified with reflection, EVEN IF they are declared *private…*
- … something you should NEVER do, unless you are writing a library that requires it (eg a JEE container, or (un)marshalling of JSON/XML data)

Proxy Class

- The proxy would be automatically generated by the container

```
public class Foo {

    public String someMethod(){
        return "foo";
    }
}
```

```
public class FooProxy extends Foo{
    private final Foo original;

    public FooProxy(Foo original) {
        this.original = original;
    }

    @Override
    public String someMethod(){
        // do something before, eg start a transaction
        String result = original.someMethod();
        //do something after, eg, commit the transaction
        return result;
    }
}
```

Generation of Proxy Classes

- *It is actually quite complex*, as a proxy class would not exist at compilation time
- The proxy class is created at runtime via bytecode manipulation
- The Java SE (not EE) API provides some basic functions to create proxy classes, but they require the existence of interfaces, and not just concrete classes

Lazy Collections

- Collections declared with *@OneToMany* and *@ManyToMany* are not loaded by default
- They are loaded only when accessed (ie, *lazy* loading)
- But you need to access them inside a transaction
- If you try to access them outside, you will get an error
- So, if need such data, need to force loading by accessing them while in a transaction
- Note: we will go into details of transaction boundaries later in the course

Container Deployment

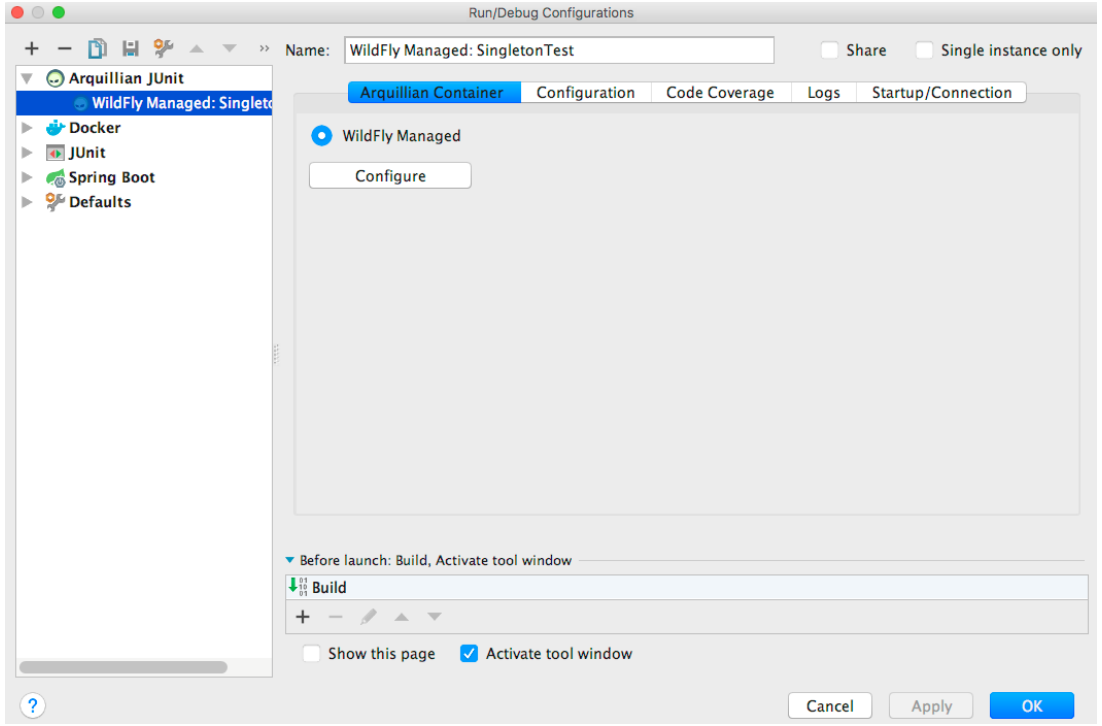
- To use EJBs, we need to run them in a JEE Container
  - *WildFly, GlassFish, Payara*, etc.
- We would need to package the JAR/WAR with our code, install it on a running container
- But before that, we would need to **download**, **install** and **start** a container
- But how to test the methods of EJBs directly from a JUnit test?

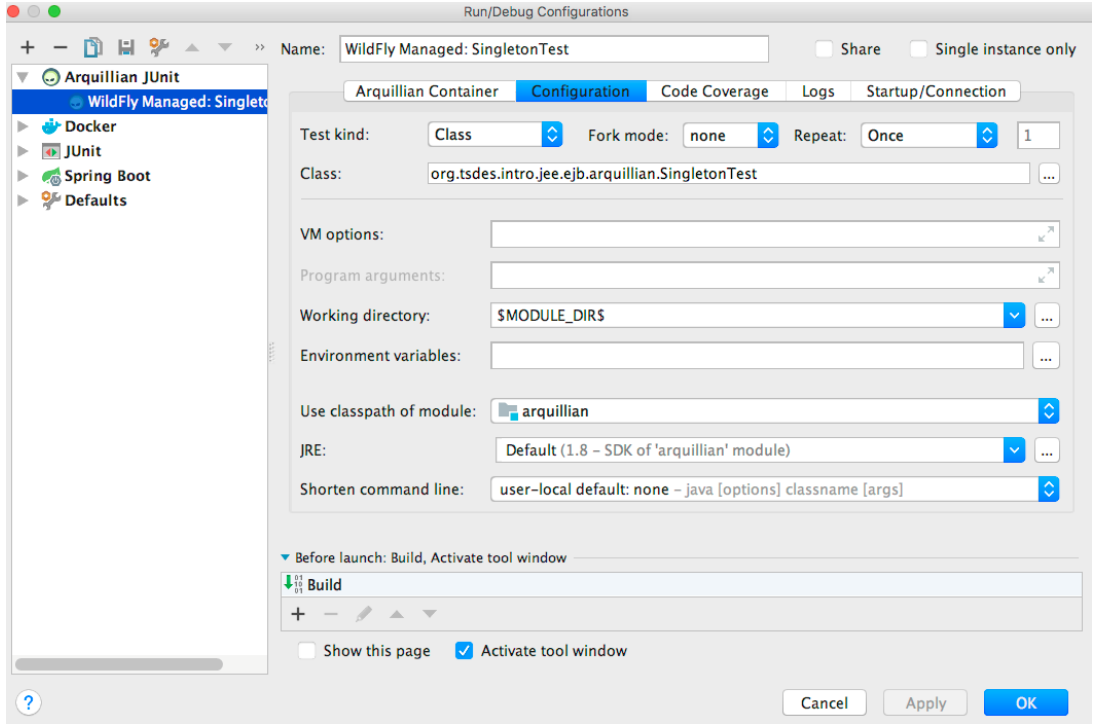
Arquillian

- A library extending JUnit that allows you to package JAR/WAR files directly from tests and deploy them on a container
- The tests themselves are run in the container, so can use dependency injection *@EJB*
- Configuration in special resource file called *arquillian.xml*
- Limitations: cannot just right-click in IDE to run tests, need some manual settings first…
- … plus, you still need to download and install a JEE Container
- Note: life will get easier once we start with *SpringBoot*…

Test Configuration

- Arquillian “*WildFly Managed*”
- “Working directory” -> “$MODULE_DIR$”
  - note: recent versions of IntelliJ might use a different name for such variable, eg “$MODULE_WORKING_DIR$”. Just choose the right one from the drop-down list





Download/Install WildFly

- We do it with a Maven plugin, as part of the build
  - Note: we could use *Docker*… but here we just want to see how Maven plugins can be used to do several different things during the build
- *WildFly* installed under the "*target*" folder
  - So it would be deleted when running "*mvn clean*"
- Need to run "*mvn test*" at least once to download/install *WildFly* **BEFORE** you can run tests in IntelliJ

Multi-Module Projects

- Usually, you would run *Maven* commands like "*mvn test*" directly from the root of your project
- TSDES is a large project: if you build from root, might take a long while…
  - well, "*large*" for students, but not compared to actual enterprise systems…
- If you build a module directly (eg "*mvn test*" in the module folder), it will fail if using other modules as dependency
- You need to run "*mvn install -DskipTests*" at least once from the root of the project
  - so, all JARs of the modules get installed in your *~/.m2* folder, and can be referenced when modules are built in isolation and not from the root of the project

Git Repository Modules

- *NOTE: most of the explanations will be directly in the code as comments, and not here in the slides*
- **intro/jee/ejb/stateless**
- **intro/jee/ejb/query**
- **intro/jee/ejb/lazy**
- **intro/jee/ejb/framework/injection**
- **intro/jee/ejb/framework/proxy**
- Exercises for Lesson 04 (see documentation)