Provide an object-oriented format for defining test data for the Arquillian Persistence Extension
Draft 10

Student:
Martin Skurla

Mentor:
Bartosz Majsak

Content

1. Assignment

DBUnit data sets are low-level, database oriented test fixtures. Hence it's often quite cumbersome to write them by hand, especially when there are relationships between entities/tables. The goal of this project is to introduce an alternative way of describing test data in external, human-readable text files representing object structure (entities), not rows as it's in DBUnit case. Considered formats are YAML or JSON.

- **Knowledge prerequisite:** Java, some familiarity with testing techniques and frameworks such as JUnit or TestNG
- **Skill level:** Low

- **Contact(s):** **Bartosz Majsak**, **Aslak Knutsen**
- **Mentor(s):** **Bartosz Majsak**

- **Associated project(s):** **Arquillian**

2. DBUnit

In order to improve the way how DataSets could be defined (basically to improve the characteristics or even replace the DBUnit), we need to gain deeper understanding of the **DBUnit**[1] concepts and how it works. Let's first start with a brief overview of what DBUnit is, what unique features it has and summarize some of its advantages and disadvantages.

DBUnit:

- is a JUnit extension
- has the ability to export and import database data to and from XML DataSets
- can verify that your database data match an expected set of values
- assures that the test cases clean out the database before starting any tests
- supports various formats of DataSets:
  - flat
  - XML
  - CSV
  - DTD

2.1 Advantages

- Maven integration
- DBUnit can generate DTD for flat XML DataSet
- DBUnit supports both in-container and remote client connection strategy
- the granularity of database operations that could be executed before and after each test is fine-grained

2.2 Disadvantages

- DBUnit can't execute DDL (CREATE, ALTER, DROP, TRUNCATE, COMMENT and RENAME SQL statements)
- even if there are many **DataSets** [2] (implementations of org.dbunit.dataset.IDataSet), some of the XML ones are not very user readable/flexible
- FlatXmlDataSet represents database data as rows in tables; with bigger number of columns, this quickly gets messy
- XmlDataSet represents database data as tables with a very low “database values vs XML markup” ratio
- the XML format for DataSets is just not always good enough / readable / usable

2.3 Summary

It turns out that the most of the “weirdness” is done because the data is represented in tabular form and there is always a lot of “noise” caused by XML markup.

Taking advantage of object-oriented way to describe the DataSets and getting rid of unnecessary XML markup, the process of writing DataSets could become fun again.

3. Arquillian Persistence Extension

Arquillian is an open source framework aimed at making Unit and Integration testing of Enterprise Java applications easy and maintainable. It is also flexible in a way that the testing is application container (Servlet, CDI, EJB) independent. As a proof, there is a bunch of containers supported by Arquillian (including standalone, remote and embedded ones).

Because of its modular and thus also extensible design, it is divided into several modules, each of them providing support for a certain aspect of its test infrastructure. Most of the modules are divided into API, SPI (Service Provider Interface), implementation and testing modules. This is a great example of code separation best practices.

Arquillian Persistence Extension is an Arquillian extension providing you help with writing integration tests dealing with database operations. It provides a useful set of Java annotations which significantly reduce the overall complexity part of every integration test. Arquillian Persistence Extension seamlessly integrates into Arquillian core.

3.1 Features

Arquillian Persistence Extension comes with many useful features:

- wraps each test in the separated transaction (with commit(default) or rollback at the end)
- seeds database using DBUnit with XML, XLS, YAML and JSON supported as DataSet formats using @UsingDataSet annotation
- compares database state at the end of the test using given DataSets defined using @ShouldMatchDataSet annotation
- supports many standalone, remote as well as embedded Servlet, CDI and EJB containers
- is build on top of DBUnit and so is not responsible for neither native SQL code generation nor its execution
- registers many events (hooking into Arquillian core model abstractions) effectively exposing its SPI and thus being extensible itself (yet not SPI but currently part of implementation):
    - AfterPersistenceTest
    - ApplyCleanupStatement
    - AplyInitStatement
    - BeforePersistenceTest
    - CleanupData
    - CleanupDataUsingScript
    - CompareData
    - EndTransaction
    - ExecuteScripts
    - PrepareData
    - StartTransaction

4. Object-Oriented DataSet Proposal

Now, let's focus on the Object-Oriented DataSet proposal. In this chapter, we will look at main design goals, we will build “Mental Model” (changing current Arquillian Persistence Extension pipeline) and look at proposed improvements in more details.

4.1 Design Goals

The ultimate design goal started with the idea: “In the same way as ShrinkWrap is not directly tight to the Arquillian, the Class Generation, DataSet Parsing and Database Population “machinery” should be Arquillian/Maven/JUnit independent at the core level.”

Building on that code design goal, let's define more specific design goals:

- provide an API for possible class generation, DataSet parsing and database population which will be than wrapped by Maven MOJOs, JUnit extensions and the ultimate Arquillian and its Persistence Extension
- everything should be done in a modular and thus extensible manner (multiple Maven modules and the usage of Maven profiles)
- code generation (if approved) should be as transparent as possible, ideally totally invisible to the end user

All the mentioned parts combine different levels of flexibility, transparency and accidental complexity. However, one thing that all of them should have in common is graceful error handling.

4.2 Building “Mental Model”

It appears that in most cases DBUnit is powerful enough to be used directly. However, in some special cases (like we don't like the XML DataSet format) we need to be more flexible.

In such cases, where we don't like every detail of DBUnit but we need similar (ideally better) functionality, we need to mimic the DBUnit functionality at some level. First, let's quickly summarize what is DBUnit doing behind the scenes. There are practically 2 logical steps DBUnit is providing for you:

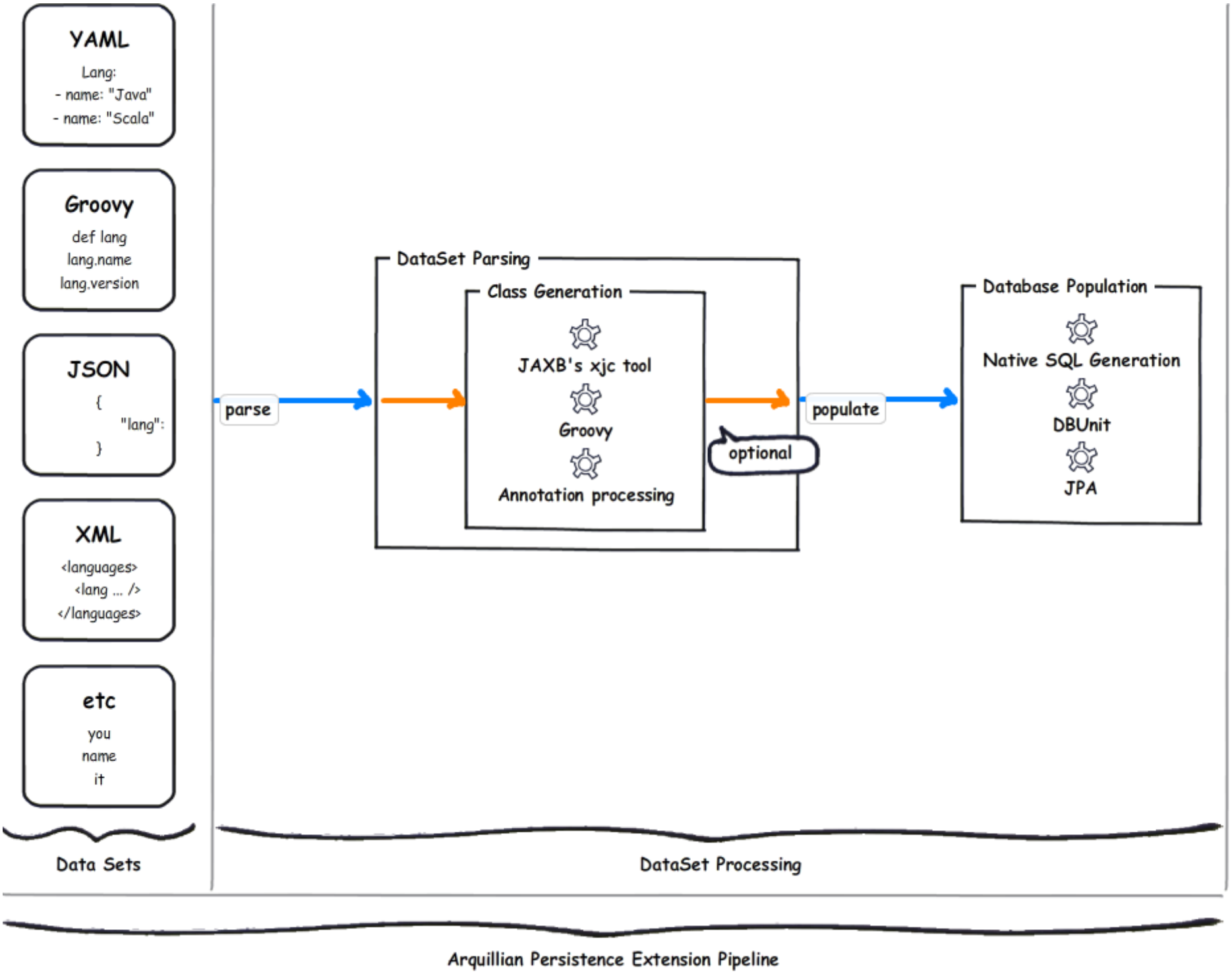
- parsing database data stored in various formats
- database data population

The main idea behind the “Mental Model” is to extend current Arquillian Persistence Extension functionalities by extracting its current functionalities into multiple logical parts and thus providing “bonus features”. As you will see in the following chapters, the whole “Mental Model” was divided into 3 parts:

1. Class Generation
2. DataSet Parsing
3. Database Population

So instead of having all of them tightly integrated together, we should always “Encapsulate what varies”. Practically speaking, 3 logical parts should build 3 parts of the final SPI, where every part should be explicitly represented in the SPI (interfaces, final classes, enumeration types).

Finally, let's have a look at the proposed “Mental Model”:


YAML
Lang:
- name: "Java"
- name: "Scala"
Groovy
def lang
lang.name
lang.version
JSON
{
"lang":
}
XML
<languages>
<lang ... />
</languages>
etc
you
name
it
Data Sets
parse
DataSet Parsing
Class Generation
JAXB's xjc tool
Groovy
Annotation processing
optional
populate
Database Population
Native SQL Generation
DBUnit
JPA
DataSet Processing
Arquillian Persistence Extension Pipeline

I was carefully reading the Forum Thread “Arquillian Persistence Extension Roadmap” and one idea stays in my mind: “the idea to easily navigate from the test class to the fixture definition”. Although the generated object mapping and its runtime representation is not an example of text fixture, the navigation between the test class and its DataSet representation (using generated classes) will be more straightforward.

We learned from compiler theory as well as SRP (Single Responsibility Principle) that multiple steps and single responsibility is usually better than one tightly coupled step. So in comparison to the current Persistence Extension functionality, where parsing and injection are done in one step, this really means separation of logic. And as you can see from the previous picture, everything is more explicit. The process starts with a DataSet file in various formats (and should be extensible). Let's describe the DataSet processing in more details.

First we need to do the DataSet Parsing. This includes optional class generation using various technologies and should be pluggable as well. If the class generation will be a part of DataSet parsing, it has to be done before the actual parsing because the result of the parsing should be an object model using those previously generated classes.

Anyway, the second step after the DataSet parsing is database population. Again, there are multiple technologies that could be used for database population and this should be pluggable as well. The most interesting point from the API design perspective here is how the various object models could be processed by various database population techniques.

In the following sub-chapters we will focus on every step in the pipeline: class generation, DataSet parsing and database population.

4.3 Class Generation

This sub-chapter will be about extending one of the core Arquillian Persistence Extension abstractions. For some use cases it could be very handy to have an object oriented model that could describe any database data (adding ability to intercept, pre-process, post-process, etc).

Adding class generation is definitely not an easy and straightforward task and we have to be careful about any other pieces that will be affected. This includes not only, but also:

- transparency
- performance

4.3.1 Transparency

The class generation as well as seamless integration both need to be transparent. Of course adding a class generation step will create a bunch of new issues we need to address.

As soon as we have new types (from Java programming language point of view), we need to think about **appropriate classpath handling**. Handling classpath in fact contains 2 activities:

- development time classpath (compile, test) – should be handled by Maven
- deployment time classpath (runtime) – should be handled by ShrinkWrap

The class generation should be integrated with Maven in a way that either generated classes will be placed in directory structure Maven understands or some custom solution will need to be found/developed. Unfortunately Maven supports just one source directory per standard Java project (represented by <build><sourceDirectory> XML element in the Maven POM).

On the other side ShrinkWrap will need to add those compiled classes as part of the deployment by default because forcing user to put them in the deployment manually doesn't seem like a good idea.

Another important aspect that needs to be taken into account is **test enrichment**. User should be able to interact with the object model in read-only or read/write mode, because the interactivity with object model was one of the main requirements in the first place. Arquillian should have facilities to respect it. For instance it should be able to do the dependency injection of mentioned object model with whatever representation it has.

4.3.2 Further directions

This chapter is about class generation and the classes should be “POJO styled”. There is however a possibility of having a general class with some kind of HashMap-based structure for storing database data. And so you will always have Object model objects with the same type.

This solution will have the advantage of not having a code generation at all. It could also be much more transparent because as soon as you have the final class for object representation, you are able to build a DataSet implementation on top of that. As a result it will still be possible to do data transformation and the overall integration complexity will be low.

One important question came into my mind. Would it be necessary to make a synchronization between database and Object model ? It will be just weird if the values in Object model do not refer to the corresponding database tables.

Last but not least, we need to think about determining proper field types in generated classes. I see two possible approaches:

- determining field types from DataSet resource – could be tricky
- determining field types from database metadata – better, but how will you gather the data ?

As soon as we start talking about class generation, multiple questions naturally arise:

- Where (what directory) the code should be generated ?
- Who will be responsible for putting classes into classpath ?
- Who will be responsible for compiling classes ?
- What will trigger the code generation? Compile on safe would be great solution for that. But would that be really general enough to be used publicly?
- Will some addition to JBoss Forge / JBoss Drone be necessary / useful?
- Could those generated classes be JPA entities ?

We have made interesting introduction into class generation, now let's examine chosen approaches for class generation.

4.3.3 JAXB's xjc Tool

The first candidate for Class Generation is JAXB and its xjc (XML to Java compiler) tool. Even though DBUnit's XML DataSets could be readable, there are options to generate classes directly from XML files. We could use JAXB's xjc tool to generate Java classes from XML Schema documents and let JAXB to populate the Object tree according to the XML content.

This solution has one serious disadvantage - you will require the users to always embed XML Schema to every XML document you would like to process.

4.3.4 Groovy and its dynamic behavior

Now let's take a completely different approach. What about not generating classes at all? The DataSets could be written in a JVM related dynamic programming language (Groovy, JRuby, Jython). This will on one hand shield you from the need to generate classes, but on the other hand the processing will be dependent on the chosen language. You have to take into account the dynamic behavior of the chosen language as well as the interoperability with Java, possibly Meta Object Protocol.

There is also one serious drawback. For some languages there could be an Object model mismatch between itself and Java. This effectively reduces the set of possible dynamic languages that could be used easily.

As a representation of this group I have chosen Groovy. Groovy has very similar syntax, semantics and Object model in comparison with Java. It also supports JSR 223 (Scripting on the Java Platform) and thus Groovy scripts could be easily run directly from Java.

Still, there are few disadvantages of using Groovy, mostly possible performance degradation (even if Groovy 1.8 already improved performance and JSR 292 could help improve the performance even more).

4.3.5 Annotation processing

Let's be even more crazy :). Let's take a completely different approach one more time. One of the nice things about Arquillian is the usage of annotations. What about taking advantage of annotations even further? What about using annotation processor for class generation? Annotation processing is a standard way to generate sources/resources as part of the compilation process.

In this case annotation processor will not be processing any kind of annotations, but it will be just a tool to generate classes from given DataSets. There are 2 ways to execute Annotation processors:

- specifying "-processor" argument as a part of javac command on command line
- specifying annotation processor implementation using ServiceLoader

To be Maven independent, we can steal the annotation processing integration solution from Hibernate and create a specific module that contains only the annotation processor implementation and its registration using ServiceLoader.

4.3.6 Annotation processing on steroids

If you are thinking that the idea of using annotation processing just cannot work effectively, well there is a very interesting functional prototype called "**LiveDB**" [3] build on top of annotation processors and IDE support for them created by Jaroslav Tulach (Traditional NetBeans Platform architect).

What a noble idea to generate classes representing database tables directly from database metadata automatically using annotation processor! This is a slightly different approach to what almost any other class generation will probably try to do. Instead of deducing data types from its values, types could be easily and safely determined from the database metadata.

We can even integrate **Project Lombok** [4] and no getters and setters will be necessary to be generated any more.

4.4 DataSet Parsing

DataSet parsing is the process of parsing the DataSet into some kind of intermediate representation. It optionally includes the class generation and in that case the intermediate representation is using object model from generated classes.

As part of DataSet parsing, we need to be able to associate a proper parser to a proper DataSet file type. Speaking of which, I think I have found an example of a leaking abstraction introduced as part of org.jboss.arquillian.persistence.data.descriptor.Format enumeration type.

Trying to abstract a type of a file together with its extension is not a very good idea. The problem is the fact that the same file type could be represented by multiple extensions (for instance YAML format have both "*.yml" as well as "*.yaml" extensions). So it will be probably a better idea to do a mapping between parser and the MIME type of processed DataSet and then additional mapping between those MIME types and file extensions. The mapping could be done using ServiceLoader.

As soon as the DataSet parsing will be abstracted and thus could be extended (part of the SPI), few new questions arise:

- Is it required to support just text-based DataSet formats? If not, how will the binary DataSet formats be handled?
- Should parsers use streaming parsing strategy (or should the streaming parsing be recognized as the best practice for parsing DataSets) ?
  Streaming parsing strategy is important when big bunch of data needs to be processed and you can't afford to put the whole model into memory. This changes the question into: Does it make sense/is it common to use big DataSets? Well the answer is probably not. According to **DBUnit Best Practices** [5] , you should "use multiple small DataSets".

4.5 Database Population

We finally arrive to the final step in Arquillian Persistence Extension Pipeline. Obviously the most important goal of the Persistence Extension is to populate database with given data. Because at the end of the day there always have to be SQL INSERT statements when dealing with relational databases, the real question here is who will be responsible for writing/generating them. Will the generation be transparently handled by some support technology, or will we have to write our own generators?

In this sub-chapter we will quickly summarize some of the possible options how could a database be populated. I chose 3 approaches we will focus on:

- native SQL generation
- using DBUnit
- using JPA

4.5.1 Native SQL Generation

Native SQL generation is probably not that great idea because of various SQL dialects. Trying to implement native SQL generation flexibly would be very difficult and practically a waste of time. But is it really our case?

As a starting point, we will need just an INSERT SQL statement (with native database types handling) and a few cleanup SQL commands. The final number of SQL commands depends on the Arquillian Persistence Extension API flexibility (org.jboss.arquillian.persistence.CleanupStrategy). Additional task would be to implement Transactional behavior (in database vendor specific way).

4.5.2 Using DBUnit

Probably a better idea will be to incorporate “the bulldozer approach” and let somebody else (understand some technology) generate the native SQL code. That's in fact what anybody is using and why JPA and ORM in general are so important. That idea can be incorporated using DBUnit and because Arquillian persistence Extension is already using DBUnit, the integration should be the most straightforward from all proposed database population technologies.

4.5.3 Using JPA

In the Enterprise Java world, we usually use JPA, that does all the low-level code and SQL generation and handles almost everything for us.

One of the possible results of class generation together with DataSet parsing could be an object model directly using JPA entities. In such case, JPA will then be used to do the database population and this solution will be one of the most effective.

5. Further Questions & Ideas

1.) Arquillian is a Java EE container-based testing framework, so it is naturally tight to JPA as Java EE standard. But is it really required to be tight to relational databases? What about NoSQL or JDO?

2.) I was thinking about one potential Arquillian API improvement. Do you think it will useful (can you find a use case) to have the ability to define multiple @Cleanup or @CleanupUsingScript annotations on the same method? The idea is to do different cleaning at the beginning and at the end of method execution with (otherwise it does not make much sense).
Now because the Java programming language does not support multiple occurrences of the same annotation on the same language element, we will need to add some kind of "Container annotation":

```
@Retention(RetentionPolicy.RUNTIME)
@Target(/* same as @Cleanup */)
public @interface Cleanups {

    Cleanup before();
    Cleanup after();
}
```

6. Summary

As a summary, this proposal didn't meant to be a final solution, but rather a summary of various ideas I had (maybe even the crazy ones), summarizing advantages as well as disadvantages of possible solutions.

The final questions are:

- “How well would any of proposed approaches fit into Arquillian and ShrinkWrap build and deployment model?”
- "How smart could we really be and how will it affect Arquillian usability?"

7. Links

[1] DBUnit - **http://www.dbunit.org/**
[2] DBUnit Core Components (including DataSets) - **http://www.dbunit.org/components.html**
[3] “LiveDB concept” - **http://wiki.apidesign.org/wiki/LiveDB**
[4] Project Lombok - **http://projectlombok.org/**
[5] DBUnit Best Practices - **http://www.dbunit.org/bestpractices.html**