Testing JPA Code with Maven

Testing code involving databases is one of the more challenging tasks we will encounter when writing tests, raising questions like:

- Which database should to use when we test locally?
- How to ensure that tests run instantly when our code is cloned/pulled regardless if on a local developer machine or on Travis
- Which database should touse when testing on Travis?
- How to ensure that data in the database always is in a "known state" (so that results of running one test, does not interfere with subsequent tests?
- How to ensure, if using a remote database, that tests running simultaneously from different computers will not interfere with each other?
- How to write fast tests? Database access is typically slow, especially if testing on a remote database?
- How to (automatically) switch to another database, when tests runs on Travis, and not on a local developer machine?

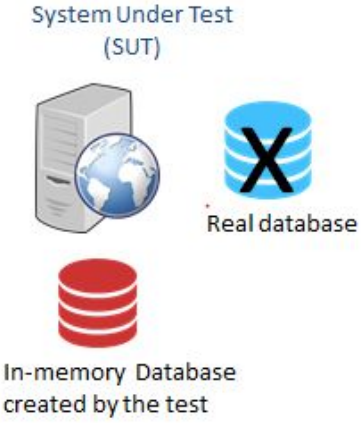
There is no one right answer to all these questions. This semester we will set our major goal for database testing, like this:
Tests must run seamlessly right after cloning, no matter where they are cloned. Two parallel tests may not interfere with each other, and we want a SIMPLE setup.

Using these requirements, we suggest you use an in-memory database for all tests that involves a database. This is not a perfect solution for all tests, but it is far better than not testing database code at all. The advantage of this strategy is, that our tests will run everywhere without the need to set up a specific database.

Mocking Databases, using an in-memory Apache Derby DB.

Mocking database operations "manually" can be very cumbersome and time-consuming.
If we limit our goals to that: Database Unit Test Cases can be executed immediately after a project is cloned, without the need to set up a local/external-test database, and should run notably faster than a traditional database" we can use an in-memory database for testing.
This document targets systems using the Java Persistence API (JPA), but you can use the ideas for other Object Relational Mappers (ORM), or with traditional database systems, written without an ORM

Read this article for a short introduction to InMemory Test databases:



Several databases can be executed in memory. This document uses the Apache Derby Database since it provides this feature, that is; *no need for database installation, setup, start, external database files etc*.
For a Maven Based Project, all it takes to get started is this entry in the Pom-file (check if a newer version is available).

```
<dependency>
   <groupId>org.apache.derby</groupId>
   <artifactId>derby</artifactId>
   <version>10.12.1.1</version>
</dependency>
```

With *JPA* we have at least one Persistence Unit in our *persistence.xml* file with details about the actual database . We need at least two persistence units, one for our test database and one we use while we develop, similar to the database used in production.

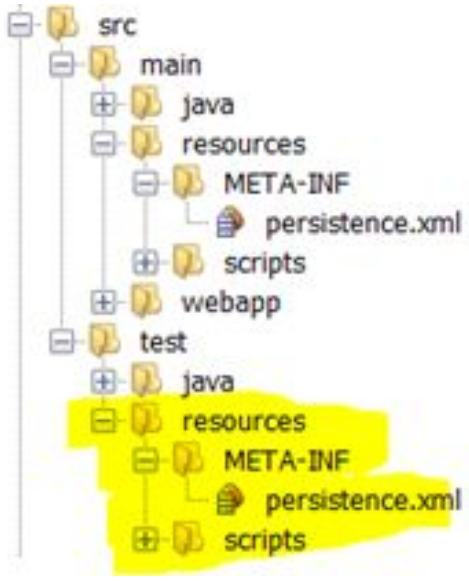
We can solve this problem in two ways.

1. Use a specific persistence.xml file for testing. This can be done with Maven, and its strategy to look in different folders depending on whether it is executing "normal" code or test code. This is our recommended way for Maven projects
2. We can have a single *persistence.xml* file with two or more *persistence units*, as sketched in strategy 2 below.

1) Place a persistence.xml file in a folder maven will use for testing

With a maven/JPA project you will have a folder structure as sketched below in the part under the *main* folder (not the scripts folder, you add that yourself if you have any scripts).

If you generate a *test* class with the NetBeans Wizard, it will generate folders as sketched in the part under the *test* folder (NOT the yellow part).
If you create the folders outlined in yellow and place a new persistence.xml file in this folder (one that uses the in-memory database) this is all that is required.



Maven (and NetBeans with a maven project) will use this persistence file whenever you are running your tests. If you have used the *Drop-and-Create* strategy, you will get a fresh database each time you run your test.

MAKE SURE to rebuild your project whenever you make changes to your persistence.xml file.

2) A single persistence.xml file, with two or more persistence units (one for development, one for test)

This solution requires that your façade classes are written to take in an EntityManagerFactory, so we can provide the right persistence-unit name to the Factory.

You can have more than one persistence-unit in your *persistence.xml*, *as long as you supply each with a unique subpackage name as sketched in yellow below.* You can paste this into an existing persistence.xml (but provide the existing persistence-unit with a package name similar to how it's done below (call it dev)

```
<persistence-unit name="pu_test" transaction-type="RESOURCE_LOCAL">
  <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
  <class>entity.Person</class>  <!-- REPLACE THIS WITH YOUR OWN ENTITY CLASSES →
  <exclude-unlisted-classes>false</exclude-unlisted-classes>
  <properties>
    <property name="javax.persistence.jdbc.url" value="jdbc:derby:memory:myDB;create=true"/>
    <property name="javax.persistence.jdbc.driver" value="org.apache.derby.jdbc.EmbeddedDriver"/>
    <property name="javax.persistence.jdbc.url" value="jdbc:derby:Users;create=true"/>
    <property name="eclipselink.logging.level" value="WARNING"/>
    <property name="eclipselink.canonicalmodel.subpackage" value="test"/>
    <property name="javax.persistence.schema-generation.database.action" value="create"/>
  </properties>
</persistence-unit>
```

Getting Started with JPA, test and DB-mocking with an In-memory database

Fork this project to get a start project set up using strategy-1 from this document and Apache Derby as a Mock-database for Unit-testing.