Installing MySQL

This tutorial is going to guide you throughout the setup of your workspace. First, we’re going to see how to install the MySQL RDBMS in your preferred OS. This is going to be the only part that is OS dependent.

Mac OS

For this tutorial we’re going to use the **brew** package manager. The reason behind this is that we want to deal with a system that can be easily updated (we do not want to have different versions of the same db). So, you must first be sure that brew is actually installed in your Mac. Please visit the website brew.sh and follow the instructions.

Now we’re ready to kill all the processes and to completely remove MySQL (some traces could be left behind!).

```
#!/bin/bash
brew services stop mysql
sudo killall mysql
sudo killall mysql
brew remove mysql
brew cleanup
sudo rm /usr/local/mysql
sudo rm -rf /usr/local/var/mysql
sudo rm -rf /usr/local/mysql*
```

```
sudo rm ~/Library/LaunchAgents/homebrew.mxcl.mysql.plist
sudo rm -rf /Library/StartupItems/MySQLCOM
sudo rm -rf /Library/PreferencePanes/My*
launchctl unload -w ~/Library/LaunchAgents/homebrew.mxcl.
mysql.plist
sudo rm -rf ~/Library/PreferencePanes/My*
sudo rm -rf /Library/Receipts/mysql*
sudo rm -rf /Library/Receipts/MySQL*
sudo rm -rf /private/var/db/receipts/*mysql*
```

Now you can install MySQL by just running the following command:

```
brew install mysql
```

First, we must start `mysql` as a service. This is done by the following `brew` command:

```
  brew services start mysql
```

The default user is called `root`. In order to change its password and remove unsafe settings, call the following command:

```
mysql_secure_installation
```

If the root user already has a password, then change the aforementioned command to `mysql_secure_installation -p` and

then type the password. Now, depending on your OS setup, there could be some problems:

- If there are any pw problems, then probably the install setted automatically setted the password that is given in `/Users/<yourusername>/.mysql_secret`. Use that password
- If mysql cannot read the tmp lock file, this probably means that you did not start the server. Check if the start-service command went smoothly.

Now MySQL could be accessed by typing the following command:

```
mysql -u root -p
```

If you ignored the `mysql_secure_installation` setup, then just type `mysql -u root`. Please remember that new users and passwords could be set using some default commands.

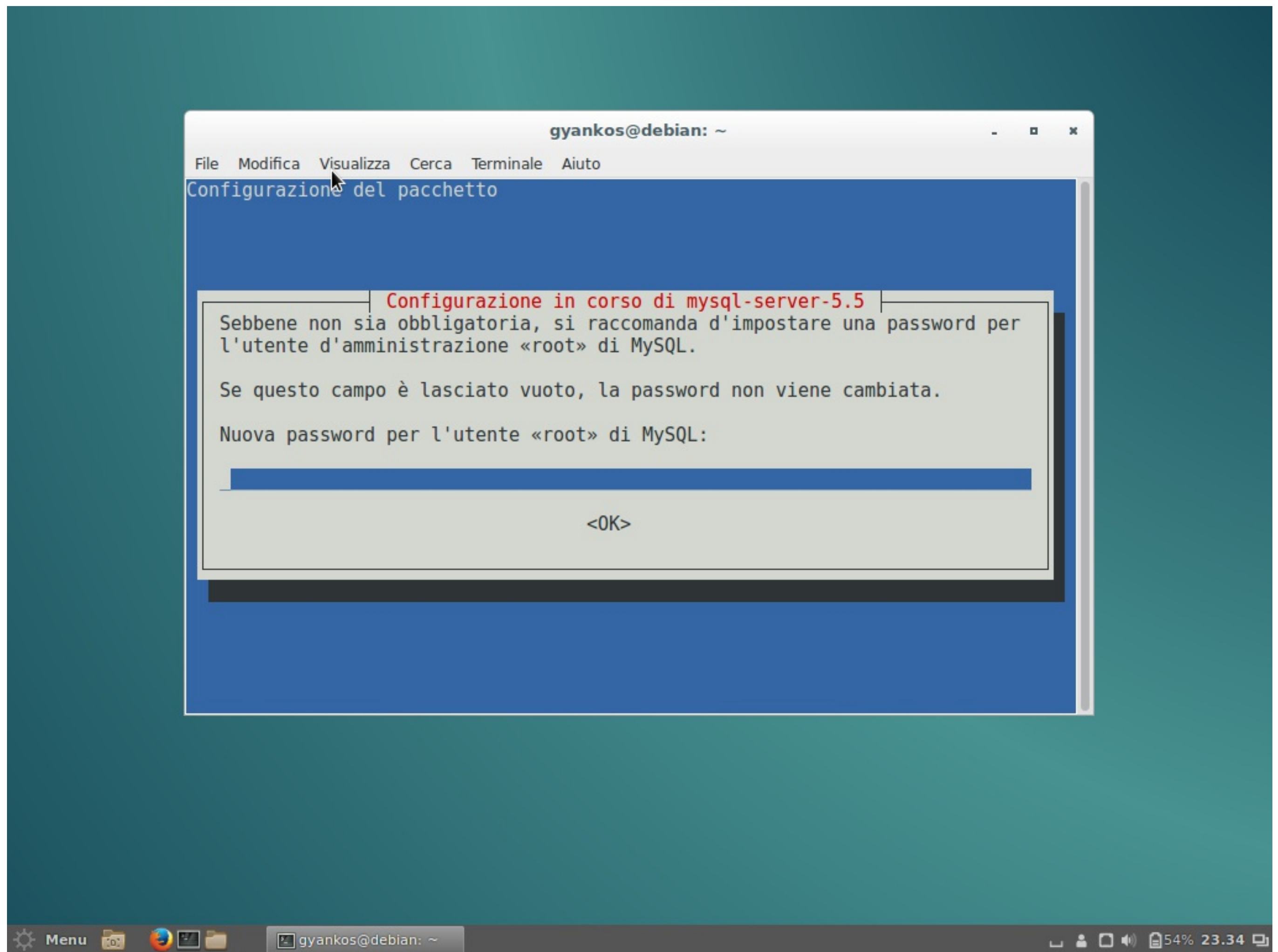
GNU/Linux

For GNU/Linux systems you have to install both the mysql server (the actual database) and the client (the client where to send the SQL queries via terminal).

```
sudo apt-get install mysql-server mysql-client
```

By doing so, the system will automatically ask you the password of

choice.



Set up a default database

In this tutorial we're going to use the mysql test database provided at https://github.com/datacharmer/test_db. Please not that, since now you setted a password, the default command is changed to:

```
mysql -u root -p < employees_partitioned.sql
```

Now we have to check if the procedure went smoothly. In order to do so, enter the MySql client:

```
mysql -u root -p
```

then we can see all the databases currently listed:

```
> show DATABASES;
```

our database is called `employees`. So now you have to type:

```
> use employees;
```

now you can list all the tables inside this database by typing:

```
> show tables;
```

We can see each table schema with the command `describe`.

Accessing RBDMSs through OO Languages

JDBC

Each DB framework uses JDBC as a common interface for accessing to the relational database. This means that you must add the MySQL driver for the database within your `pom.xml` file, alongside with the `jdbc` driver.

```
{% highlight xml %}
<dependency>
    <groupId>org.clojure</groupId>
    <artifactId>java.jdbc</artifactId>
    <version>0.6.1</version>
</dependency>
<dependency>
     <groupId>mysql</groupId>
     <artifactId>mysql-connector-java</artifactId>
     <version>6.0.5</version>
 </dependency>
 {% endhighlight %}
```

By loading the MySQL driver, Maven automatically import the jar in your project and hence there is no need to load the driver by class name, `com.mysql.jdbc.Driver`. This means that we can directly access to the database with the following Syntax:

```
 {% highlight java %}
 String dburl = "jdbc:mysql://localhost/<dbname>";
 String user = "username";
 String pwd = "password";
 /* The transaction starts here. t is an AutoCloseable ob
ject,
 and hence the close semantics automatically closes the c
onnection
 and commits */
```

```
 try (Connection t = DriverManager.getConnection(dburl,us
er,pwd)) {
    /* Put some code in here */
 } catch (SQLException e) {
    /* The transaction aborts. Do something in your code
*/
 }
 {% endhighlight %}
```

Within the next subsections we're going to see how to perform `SELECT` and `INSERT` SQL queries.

SELECT

A SQL statement has to be compiled from a string. The result set is scanned through a pointer which hasn't the standard Iterator java Syntax.

```
 {% highlight java %}
 try (Connection t = DriverManager.getConnection(dburl,us
er,pwd)) {
    Statement stmt = t.createStatement();
    ResultSet rs = stmt.executeQuery("select first_name,
last_name from employees;");
    while (rs.next()) {
        String first = rs.getString("first_name");
        String last = rs.getString("last_name");
```

```
        /* do something */
    }
} /* … */
{% endhighlight %}
```

Moreover, please note that by doing so you have to remember which are the correct types for each attribute and the JDBC methods do not reflect the actual attributes' names. Moreover, the same information (the attribute name) is repeated more than once. All these aspects are quite problematics when queries are the result of join operations involving more than one table. Moreover, any SQL query could be prone to SQL injection.

INSERT

At this point, if we want to insert some values into the database, we have to:

1. Create a POJO class in order to store the values in an ordered way (`JDBCEmployee`)

2. Define a SQL query in order to avoid the SQL Injection (use `PrepareStatement` instead of `ExecuteQuery`)

3. Define batch insertions (`addBatch`): since single transactions do not support limitless object insertions, from time to time you must push all the values to the database (`executeBatch`).

4. We have to unpack the data information in order to populate the query with the default values.

```
 {% highlight java %}
 try (Connection t = DriverManager.getConnection(db
url,user,pwd)) {
     // Defining the SQL Query
     String sql = = "insert into employees ( emp_no
,  birth_date,  " +
             "first_name,  last_name,  gender,  hir
e_date) values " +
             "(?, ?, ?,?,?,?)";
     // Prepare the statement to insert the element
s
     PreparedStatement ps = transaction.prepareStat
ement(sql);

     final int batchSize = 1000;
     int count = 0;

     for (JDBCEmployee employee : employees) {
         ps.setInt(1,employee.get_no);
         ps.setDate(2,Java2SQLData.toSQLData(employ
ee.birth_date));
         ps.setString(3, employee.first_name);
         ps.setString(4, employee.last_name);
         ps.setString(5, employee.isFemale ? "F" :
```

```
"M");
            ps.setDate(6, Java2SQLData.toSQLData(emplo
yee.hire_date));
            ps.addBatch();

            //Sooner or later, the batch has to be emp
tied when too data
            // is sent.
            if(++count % batchSize == 0) {
                //Send some data to the relational dat
abase.
                count = 0;
                ps.executeBatch();
            }
        }
        if (count != 0) ps.executeBatch();

        ps.close(); //close the statement

    } /* … */
    {% endhighlight %}
```

jOOQ

There are two possible ways to access databases from an OO language. We could either define an ORM mapping (Object-Relational Mapping), or extend the language's syntax by allowing the usage of specific SQL

statements. jOOQ implements both those paradigms.

Now create a Maven project with your favourite IDE, and add the following dependencies for your database.

```
{% highlight xml %}
<dependency>
  <groupId>org.jooq</groupId>
  <artifactId>jooq</artifactId>
  <version>3.8.6</version>
</dependency>
<dependency>
  <groupId>org.jooq</groupId>
  <artifactId>jooq-meta</artifactId>
  <version>3.8.6</version>
</dependency>
<dependency>
  <groupId>org.jooq</groupId>
  <artifactId>jooq-codegen</artifactId>
  <version>3.8.6</version>
</dependency>
{% endhighlight %}
```

At this point, we want to automatically generate the POJOs and the DAOs for our class and to automate the SQL query formulation in Java. By doing so, we have to integrate the `pom.xml` file with the `build` command stating:

1. Which database driver we’re going to use, `com.mysql.jdbc.Driver`
2. Which is the datbase URL, `jdbc:mysql://localhost/employees`.
3. Set the database username and password.
4. Choose as an `inputSchema` your MySQL database of choice, `employees`.
5. Select the source folder within your project, `src/main/java`
6. Specify the destination package, `it.giacomobergami.jOOQ.model`

```
{% highlight xml %}
<build>
   <plugins>
       <plugin>
           <groupId>org.jooq</groupId>
           <artifactId>jooq-codegen-maven</artifactId>
           <version>3.8.3</version>

           <!-- The plugin should hook into the generate
goal -->
           <executions>
               <execution>
                   <goals>
                       <goal>generate</goal>
                   </goals>
               </execution>
           </executions>
```

```

            <dependencies/>

            <configuration>
                <jdbc>
                    <driver>com.mysql.jdbc.Driver</driver
>
                    <url>jdbc:mysql://localhost/employees
</url>
                    <user>root</user>
                    <password>password</password>
                </jdbc>

                <generator>
                    <database>
                        <name>org.jooq.util.mysql.MySQLDa
tabase</name>
                        <includes>.*</includes>
                        <excludes></excludes>
                        <inputSchema>employees</inputSche
ma>
                    </database>
                    <target>
                        <packageName>it.giacomobergami.jO
OQ.model</packageName>
                        <directory>src/main/java</directo
ry>
                    </target>
```

```
                        <generate>
                            <relations>true</relations>
                            <deprecated>false</deprecated>
                            <instanceFields>true</instanceFields>
                            <generatedAnnotation>true</generatedAnnotation>
                            <records>true</records>
                            <pojos>true</pojos>
                            <pojosEqualsAndHashCode>false</pojosEqualsAndHashCode>
                            <immutablePojos>false</immutablePojos>
                            <interfaces>false</interfaces>
                            <daos>true</daos>
                            <jpaAnnotations>false</jpaAnnotations>
                            <validationAnnotations>false</validationAnnotations>
                            <globalObjectReferences>true</globalObjectReferences>
                            <fluentSetters>false</fluentSetters>
                        </generate>
                    </generator>
                </configuration>
            </plugin>
        </plugins>
```

```
</build>
{% endhighlight %}
```

All the POJOs and DAOs are generated by the `complile` command. As an example:

```
 mvn clean compile
```

At this point, jOOQ will handle automatically the transaction for you by using the DSL factory:

```
{% highlight java %}
DSLContext create = DSL.using(t, SQLDialect.MYSQL);
{% endhighlight %}
```

"Language Extension" (SELECT)

Suppose that we want to perform the following SQL query as before:

```
{% highlight SQL %}
SELECT first_name, last_name
FROM employees
LIMIT 5;
{% endhighlight %}
```

jOOQ allows a 1-1 mapping with methods of some specific objects

through the DSLContext as follows:

```
{% highlight java %}
create.select(Tables.EMPLOYEES_.FIRST_NAME, Tables.EMPLOYEES_.LAST_NAME).
from(Tables.EMPLOYEES_)
.limit(5). …
{% endhighlight %}
```

At this point we can specify to return such attributes within an `employees` record as follows:

```
{% highlight java %}
….fetchInto(tables.EMPLOYEES_).…
{% endhighlight %}
```

Java 8 could be used to automatically print the result

```
{% highlight java %}
….forEach( er -> System.out.println(er.getFirstName()+" "+er.getLastName()));
{% endhighlight %}
```

We could even decide to perform join queries. If the tables share a Primary Key and a Foreign Key, then the `onKey` statement could be used to carry out the operation.

```
 {% highlight java %}
 create.select(Tables.EMPLOYEES_.GENDER,Tables.EMPLOYEES_
.FIRST_NAME,
               Tables.EMPLOYEES_.LAST_NAME,Tables.TITLES.
TITLE)
                .from(Tables.EMPLOYEES_)
                .join(Tables.TITLES)
                .onKey()
                .fetchInto(ResultClass.class)
                .forEach(System.out::print);
 {% endhighlight %}
```

This solution requires that a specific POJO class, ResultClass, must be defined:

```
 {% highlight java %}
 public static class ResultClass  {
    public final String MrMrs;
    public final String name;
    public final String surname;
    public final String title;

    public ResultClass(EmployeesGender mrMrs, String name
, String surname, String title) {
        MrMrs = mrMrs.getLiteral().equals("M") ? "Mr." :
"Mrs.";
        this.name = name;
```

```
        this.surname = surname;
        this.title = title;
    }

    // Serialization method
    @Override
    public String toString() {
        return MrMrs +" "+name+" "+ surname+","+title+".\n";
    }
 }
 {% endhighlight %}
```

More simply, we could even use the `fetch()` method without specifying a specific POJO class.

ORM Mapping through DAOs: one-table operations (SELECT, INSERT, UPDATE)

DAO (Data Access Objects) is an architectural pattern for handling “persistency” in OO languages. By doing so we separate the `Model` Java object layer from the actual operations required to access the database. This technique allows to map each Database tuple as one object in the OO model. In order to do so, we must first initialize our DAO with our current DSLContext configuration:

```
{% highlight java %}
EmployeesDao dao = new EmployeesDao(create.configuration
());
{% endhighlight %}
```

At this point we can use the POJOs generated by jOOQ to create new rows within the database as follows:

```
{% highlight java %}
Employees e = new Employees();
e.setBirthDate(Java2SQLData.sqlData("02/10/1990"));
e.setEmpNo(555555);
e.setFirstName("Giacomo");
e.setLastName("Bergami");
e.setGender(EmployeesGender.M);
e.setHireDate(Java2SQLData.sqlData("01/11/2014"));
dao.insert(e);
{% endhighlight %}
```

Later on we could even fetch a specific row (e.g.) by ID and then perform some updates:

```
{% highlight java %}
e = dao.findById(555555);
e.setHireDate(Java2SQLData.sqlData("02/11/2014"));
dao.update(e);
```

```
 e = dao.findById(555555);
 //jOOQ POJOs are equipped with default serialization met
hods
 System.out.println(e.toString());
 {% endhighlight %}
```

Using a Persitency Framework in Java.

1. Tutorial
2. Madhusudhan Konda: *Just Hibernate* O'Reilly Media. Online book