

TP 3 - A Scene Of Information And Flexibility

Java and DataBase Connectivity (JDBC)

TP 3 - A Scene Of Information And Flexibility

Part I : A Graphic Of Table: communication with database

Load Driver

Connect to bdd

Make a request

Exercise 1: Modify the query above to add the location of the department.

Exercise 2: Move Department

Exercise 3: Generic display of Tables:

Exercise 4: Security

Exercise 5: Modify moveDepartement() to use PreparedStatement

Exercise 6: Try with displayTable

Exercise 7 (Bonus to do at home because Long): With J2EE

Part II : DAO.

Exercise 8 : What are cons of JDBC as used above ?

Bean - POJO Creation

Exercise 9 : Write the Bean for Department

Exercise 10 : Write Implementation for method find for departement DAO.

Exercise 11 : Write Implementation for method find for employee DAO.

Exercise 12 : Create DAOFactory and use it in previous example

Part III : Spring Boot & JPA.

Exercise III.1 : Init your Spring project

Exercise III.2 : Activate JPA and connect to your Database

Exercise III.3 : Read Data in your Database

Exercise III.4 : Create an Object via JPA and expose it on a REST endpoint

Exercise III.5 : GET by ID / Update by id / Delete via JPA / REST

Bonus : Follow this very great tutorial of Open Classroom on how to use Spring & JPA:

Part IV : Connections Pool (Bonus)

A Connection Originally Kept

Exercise IV.1 : (Bonus): Manually Create a List which store connection and use it in your code to make request

Exercise IV.2 (Bonus): use BoneCP to create your Connection Pool

Part V : DAO Generator (Bonus)

[Exercise V.1 : List all columns of tables using data dictionary view\(Bonus\)](#)

[Exercise V.2 : Create a new Java class programmatically\(Bonus\)](#)

[Exercise V.3 : Create all POJOs dynamically\(Bonus\)](#)

[Exercise V.4 : Create all DAOs dynamically\(Bonus\)](#)

[Tips](#)

Part I : A Graphic Of Table: communication with database

1. Load Driver

Open your favorite IDE (I recommend IntelliJ, [The Free Community Edition is great](#)). Create a Java Project with a class Main.java.

Copy the connector “**postgresql-42.7.3.jar**” in the project.

Add the jar to the Java Build Path, for that:

- On IntelliJ: File => Project Structure => Libraries => Add Library
- On Eclipse: Project => Properties => Java Build Path => Libraries => Add Jars

You will find it on Moodle (<https://moodle-ovh.isep.fr/moodle/mod/resource/view.php?id=24671>) or here: <https://jdbc.postgresql.org/download/>

Add At the top of the Main.java:

```
/* Load JDBC Driver. */
try {
    Class.forName( "org.postgresql.Driver" );
} catch ( ClassNotFoundException e ) {
    e.printStackTrace();
}
...
```

2. Connect to bdd

Note: If you are on Windows Family and use Local/legs connection or Bequeath, there is a struggle to find the right JDBC URL connection. If you succeed to connect with Windows Family on your local database with JDBC, please indicate tips here in the comment.

First you need to get the url specific to jdbc in order to connect to your database.

With Postgres the URL is like this:

jdbc:postgresql://host/database

<https://jdbc.postgresql.org/documentation/use/>

Connection is made through the [DriverManager](#). You will call its method getConnection() to have an object of type [Connection](#).

It will look like something like this:

```
jdbc:postgresql://localhost/thibautdebroca
```



Don't forget to close the request in the block "Finally". If you keep too many opened connections under a small amount of time, the server will get saturated. If you don't need a connection anymore, close it !

```
String url = "jdbc:postgresql://localhost/postgres";
String user = "postgres";
String pass = "";
Connection connexion = null;
try {
    connexion = DriverManager.getConnection( url, user, pass );

    /* Requests to bdd will be here */
    System.out.println("Bdd Connected");

} catch ( SQLException e ) {
    e.printStackTrace();
} finally {
    if ( connexion != null )
        try {
            connexion.close();
        } catch ( SQLException ignore ) {
            ignore.printStackTrace();
        }
}
```

Don't forget to add imports on the top of your Java Class:

```
import java.sql.Connection;
import java.sql.DriverManager;
```

```
import java.sql.SQLException;
```

3. Make a request

In order to make a request, you need a [Statement](#) object.

How to get it ? You just have to call [createStatement\(\)](#) from the Connection object.

```
Statement statement = connexion.createStatement();
```

Statement will provide you several methods especially:

- [executeQuery\(\)](#) : dedicated to SELECT request. It returns a ResultSet containing all rows asked in the query,
- [executeUpdate\(\)](#) : for queries that have effect on the bdd's state: UPDATE, DELETE, INSERT... It returns an Integer :
 - INSERT returns 0 in case of failure and 1 when success
 - UPDATE or DELETE returns the number of rows affected.
 - CREATE returns 0.

Don't forget to close Statement when you have finished to use it:

```
statement.close();
```

Create a method displayDepartment(Connection connexion) that will display all the departments.

```
public static void displayDepartment(Connection connexion) throws SQLException {
    Statement statement = connexion.createStatement();
    ResultSet resultat = statement.executeQuery( "SELECT deptno, dname FROM dept" );

    while ( resultat.next() ) {
        int deptno = resultat.getInt( "deptno" );
        String dname = resultat.getString( "dname" );

        System.out.println("Department " + deptno + " is for "
            + dname + " and located in ? ");
    }
    resultat.close();
}
```

```
}
```

Exercise 1: Modify the query above to add the location of the department.

Exercise 2: Move Department

Create a method `moveDepartment(int empno, int newDeptno)` that will move an employee from a department to another.

Note: If we want to display all employees with all information about employees, we'll need to extract 10 columns' value inside the `while (resultat.next())` . We would use a method that displays content of a table generically without having to know how many rows and columns there are inside. That leads to exercise 3.

(1) [Tips Exercise 2](#)

Exercise 3: Generic display of Tables:

Create a method

```
public static void displayTable(String tableName)
```

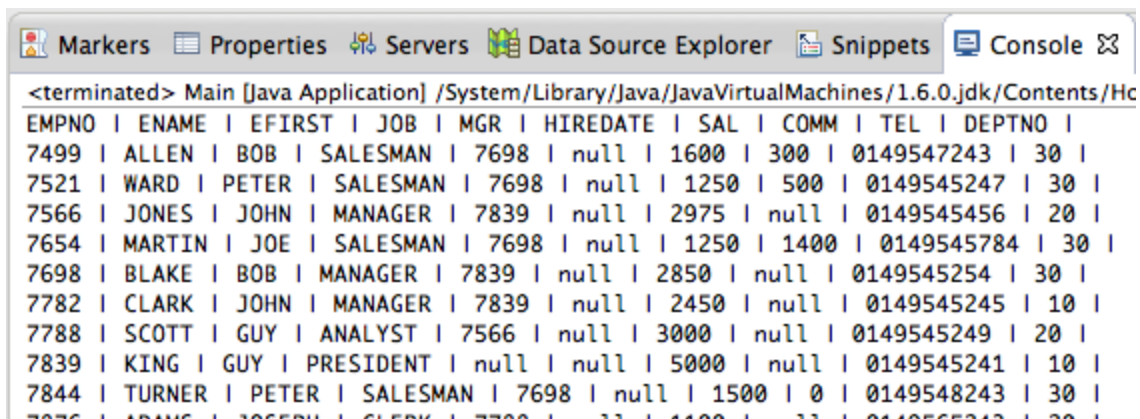
It takes as a parameter the name of a table and it displays all the content of a table, with as a first line the names of the table columns.

Example:

calling:

```
displayTable("emp");
```

will display:



```
<terminated> Main [Java Application] /System/Library/Java/JavaVirtualMachines/1.6.0.jdk/Contents/Hc
EMPNO | ENAME | EFIRST | JOB | MGR | HIREDATE | SAL | COMM | TEL | DEPTNO |
7499 | ALLEN | BOB | SALESMAN | 7698 | null | 1600 | 300 | 0149547243 | 30 |
7521 | WARD | PETER | SALESMAN | 7698 | null | 1250 | 500 | 0149545247 | 30 |
7566 | JONES | JOHN | MANAGER | 7839 | null | 2975 | null | 0149545456 | 20 |
7654 | MARTIN | JOE | SALESMAN | 7698 | null | 1250 | 1400 | 0149545784 | 30 |
7698 | BLAKE | BOB | MANAGER | 7839 | null | 2850 | null | 0149545254 | 30 |
7782 | CLARK | JOHN | MANAGER | 7839 | null | 2450 | null | 0149545245 | 10 |
7788 | SCOTT | GUY | ANALYST | 7566 | null | 3000 | null | 0149545249 | 20 |
7839 | KING | GUY | PRESIDENT | null | null | 5000 | null | 0149545241 | 10 |
7844 | TURNER | PETER | SALESMAN | 7698 | null | 1500 | 0 | 0149548243 | 30 |
7876 | SMITH | JAMES | CLERK | 7566 | null | 750 | 0 | 0149545243 | 30 |
```

(2) [Tips Exercise 3](#)

Exercise 4: Security

What is the Security flow to the method above ? What are others Cons from this basic native method ?



In any case:

In server-Side: Never Trust Entry from the Client !

Even if you apply a pre-treatment e.g.: like in Javascript

4. Prepared Request

Following your remarks in Exercise 4, we'll discover a new Object that helps to filter arguments passed to a query. It prepares the query, it's called [PreparedStatement](#):

- This can pre-compile SQL performing optimizations.
- Prepared statements are resilient against [SQL injection](#) and potential others wrong arguments passed in the query,

How to write it :

```
PreparedStatement preparedStatement = connexion.prepareStatement(  
    "SELECT * FROM emp WHERE efirst = ? AND ename = ?" );  
  
Scanner sc = new Scanner(System.in);  
String firstName = sc.next();  
preparedStatement.setString( 1, firstName );  
String lastName = sc.next();  
preparedStatement.setString( 2, lastName );  
  
ResultSet results = preparedStatement.executeQuery();
```

Exercise 5: Modify moveDepartement() to use PreparedStatement

Exercise 6: Try with displayTable

Try to modify your method displayTable by using PreparedStatement with tableName as a parameter of PreparedStatement.

Why doesn't it work ?

Lot More on JDBC :

<https://jdbc.postgresql.org/>

Exercise 7 (Bonus to do at home because Long): With J2EE

Nowadays, Database and Java are more used together in J2EE, for Web Application Server. You'll use them together very often in that case. Your server in J2EE receives http requests from your client and has to communicate with your BDD to persist data. Transform your Java application to build a J2EE server and display answers in a web browser like Firefox.

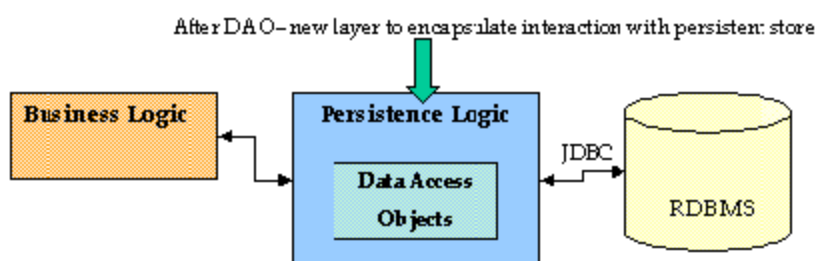
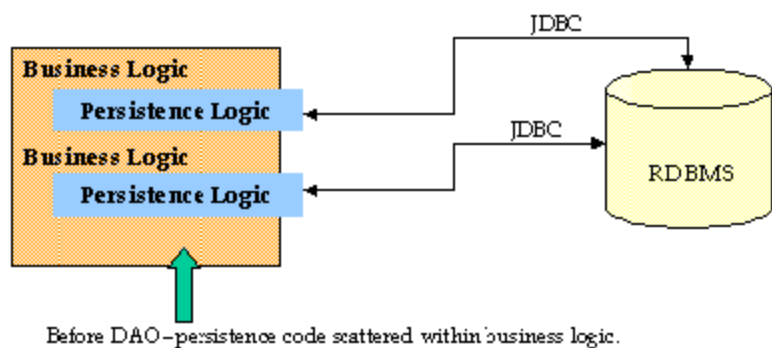
Part II : DAO.

Exercise 8 : What are cons of JDBC as used above ?

<http://www.oracle.com/technetwork/java/dataaccessobject-138824.html>

http://en.wikipedia.org/wiki/Data_access_object

<http://stackoverflow.com/questions/19154202/data-access-object-dao-in-java>



1. Bean - POJO Creation

Each Table of your Data Model needs to have its equivalent as a Java Class.

This class is called a [POJO \(Plain Old Java Object\)](#) or a Bean.

The name of Each column of your table must map an attribute of your class with the same type.

For Employee:

```
import java.util.Date;

public class Emp {

    private Long empNo;

    private String ename;

    private String efirst;

    private String job;

    private Emp mgr;

    private Date hireDate;

    private int sal;

    private int comm;

    private int tel;

    private Dept department;

    public Long getEmpNo() {
        return empNo;
    }

    public void setEmpNo(Long empNo) {
        this.empNo = empNo;
    }

    // ... others getters/setters

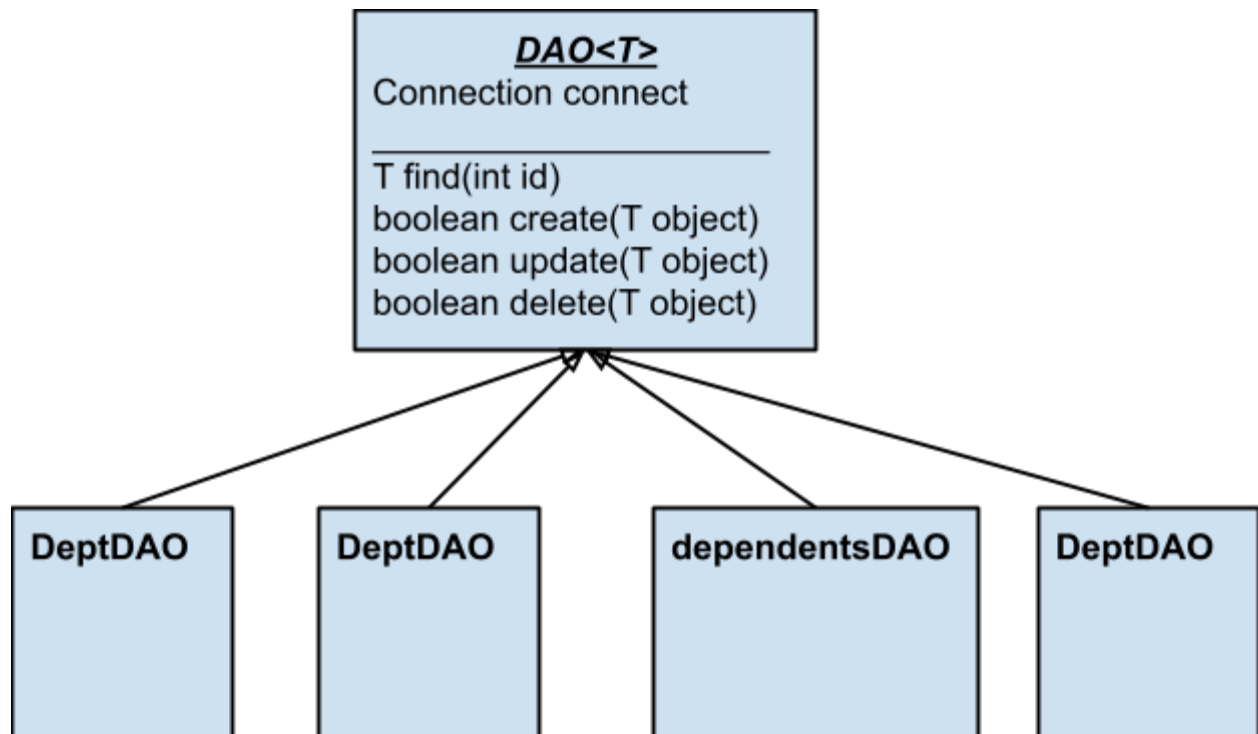
}
```

Exercise 9 : Write the Bean for Department

Following what we have written for the Employee, write the POJO class for the Department.

2. DAO Implementation

In our application, our objects will perform some [CRUD \(Create, Read, Update, Delete\)](#) operations.



Here is our DAO abstract class:

DAO.java

```
import java.sql.Connection;

public abstract class DAO<T> {
    protected Connection connect = null;

    public DAO(Connection connect) {
        this.connect = connect;
    }

    public abstract T find(int id);

    public abstract boolean create(T object);

    public abstract boolean update(T object);

    public abstract boolean delete(T object);
}
```

Exercise 10 : Write Implementation for method find for departement DAO.

In DeptDAO.java, implement find(int id) method by calling database, using the Connection object.

Then test it:

```

Connection connection = // instantiate here a new Connection.
DAO<Dept> departmentDao = new DeptDAO(connection);
Dept dept20 = departmentDao.find(20);
System.out.println(dept20); // Don't forget to add toString() method in Dept.java to be able
to pass it to System.out.println.

```

Exercise 11 : Write Implementation for method find for employee DAO.

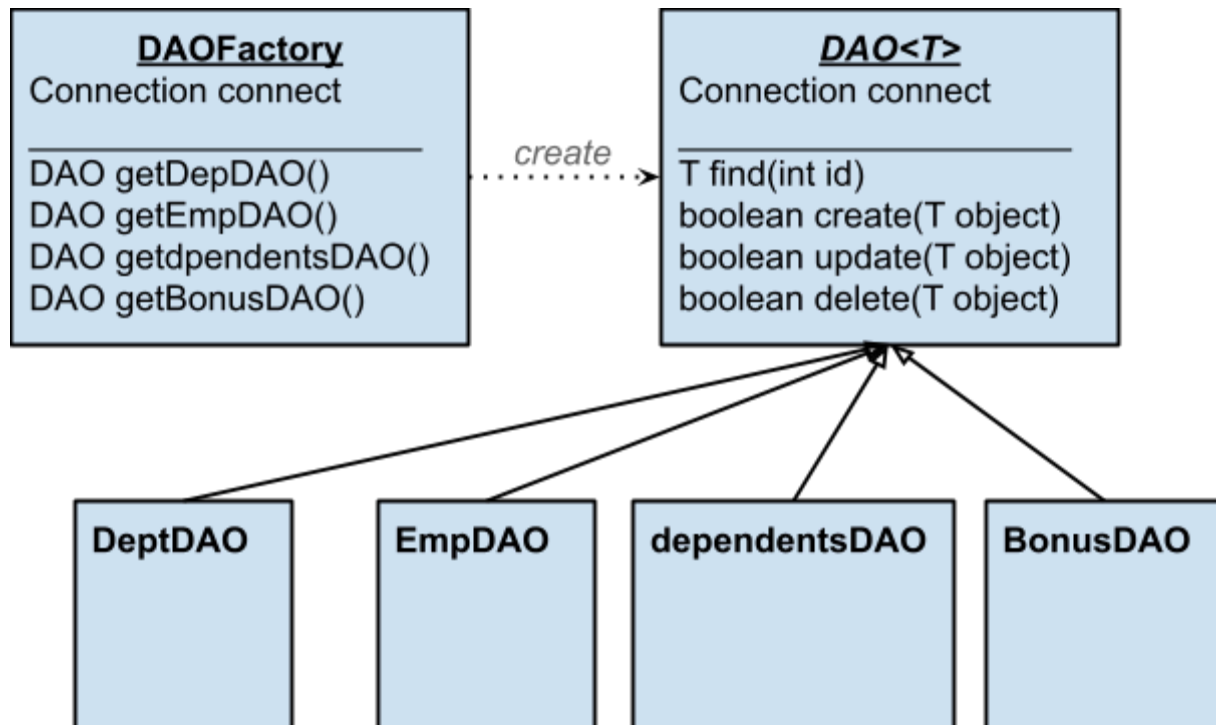
This one will be trickier as an Employee has a manager represented by an integer in the database but in the class Employee, the attribute manager, should be a type Employee. So for each employee, you need to call find(idManager) to get its manager. So you will need to build recursively the Employee

3. Factory Pattern

In the DAO pattern we can add the Factory Pattern which is a class that takes care of object instantiation.

The goal is that all object instantiation is done in the same place.

http://www.tutorialspoint.com/design_pattern/factory_pattern.htm



Exercice 12 : Create DAOFactory and use it in previous example

Part III : Spring Boot & JPA.

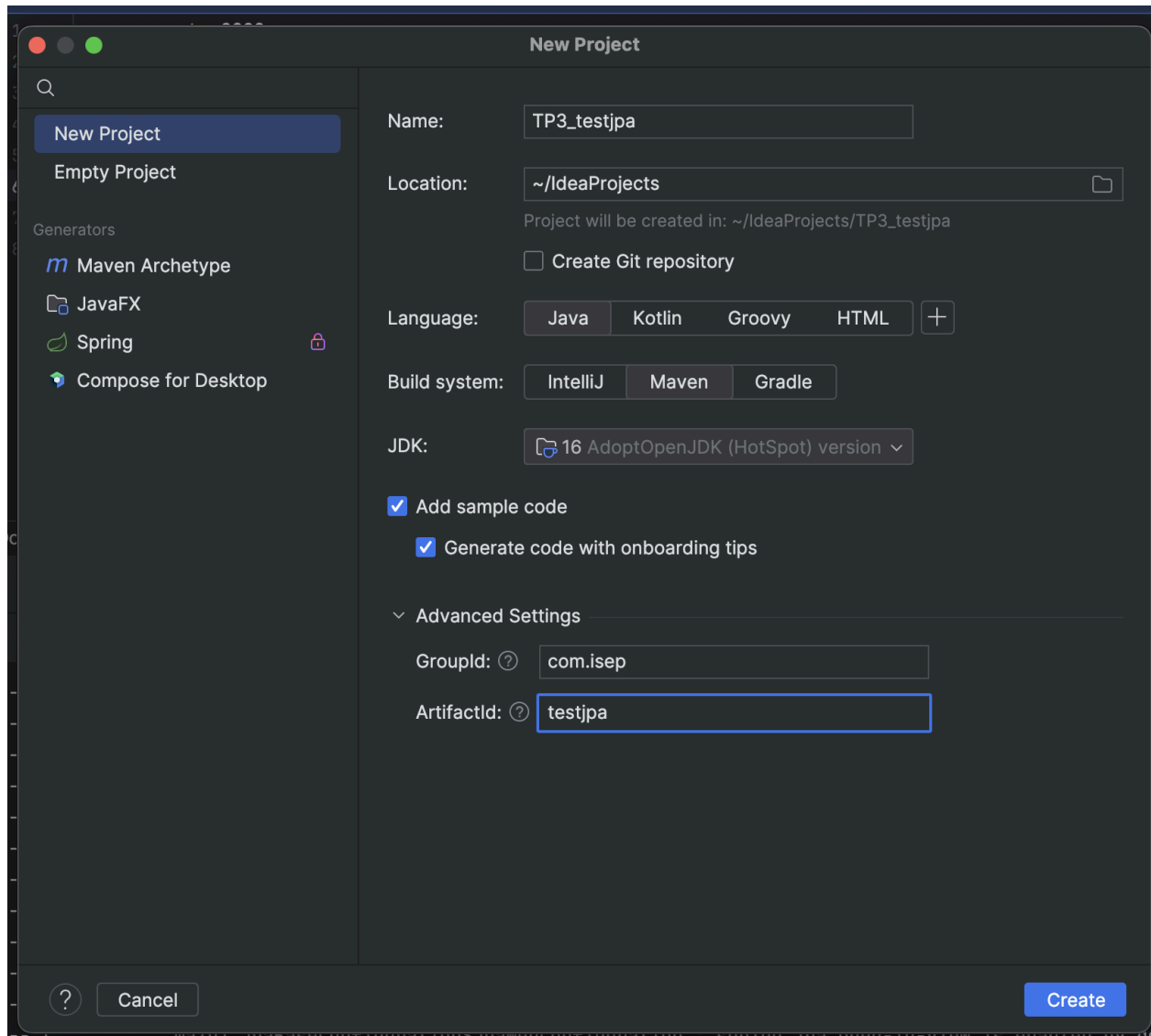
Some frameworks have been created to ease connection with Database. The most used one is JPA (Java Persistence API), especially with Spring Boot, it's widely use to map correctly the objects of your database to

Here is an Introduction to JPA: <http://arnaud.nauwynck.free.fr/CoursIUT/Intro-JPA-mapping.pdf>

To init a Spring project, you do it easily here: <https://start.spring.io/>

Exercice III.1 : Init your Spring project on IntelliJ

With your IDE (prefer IntelliJ), create a **Maven** Project with Java **Version 1.16 (The Connector given has been tested with Java 16)**.



When asked name, group and artifact, enter these values:

Name: TP3_testjpa

Group : com.isep

Artifact: testjpa

version : 0.0.1-SNAPSHOT

At the root of the project, change the pom.xml with the following content:

```
pom.xml

<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>org.springframework.boot</groupId>
```

```

<artifactId>spring-boot-starter-parent</artifactId>
<version>2.2.5.RELEASE</version>
<relativePath> <!-- lookup parent from repository -->
</parent>

<groupId>com.isep</groupId>
<artifactId>testjpa</artifactId>
<version>0.0.1-SNAPSHOT</version>
<name>testjpa</name>
<description>Demo project for Spring Boot</description>

<properties>
  <java.version>1.8</java.version>
</properties>

<dependencies>

  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
  </dependency>

  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
    <exclusions>
      <exclusion>
        <groupId>org.junit.vintage</groupId>
        <artifactId>junit-vintage-engine</artifactId>
      </exclusion>
    </exclusions>
  </dependency>

  <!--
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
  </dependency>
  <dependency>
    <groupId>org.postgresql</groupId>
    <artifactId>postgresql</artifactId>
    <version>42.7.3</version>
  </dependency>
  -->
</dependencies>

</project>

```



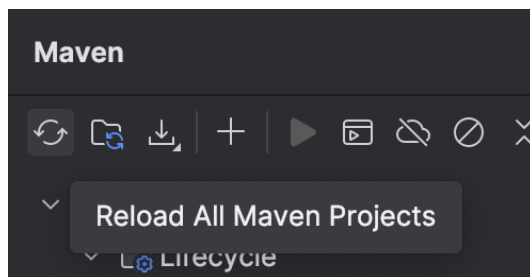
This pom.xml looks barbaric !
In the end, it's not so difficult
Let's understand it:

1. First, you have a parent tag, which refers to the Spring framework. This will help your project get the default version for Spring dependencies.
2. Then you have groupId/artifact/version: It describes your project. This is needed when you'll publish your application to distinguish your application.
3. You have the dependencies of your application:
 - a. First one is 'spring-boot-starter-web' : This includes a library to help you expose HTTP endpoints via an embedded web server. And many other HTTP features.
 - b. 'Spring-boot-starter-test' : The basics of Spring Framework.
 - c. 'Spring-boot-starter-data-jpa' Help you do links between Java and a Database.
 - d. org.postgresql the special connector for Postgres JDBC.

And the best Read The Doc:

<https://maven.apache.org/guides/introduction/introduction-to-the-pom.html>

If you see RED lines in your pom.xml, click "reload all Maven Projects":



Create a package `com.isep.testjpa`

Then add a class `TestjpaApplication` in that package:

TestjpaApplication.java

```
package com.isep.testjpa;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class TestjpaApplication {

    public static void main(String[] args) {
        SpringApplication.run(TestjpaApplication.class, args);
    }

}
```

In this package create another package/directory 'controller', and create this class:

SimpleController.java

```
package com.issep.testjpa.controller;

import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class SimpleController {

    @RequestMapping(value="/", method= RequestMethod.GET)
    public String hello(@RequestParam(value = "name", required = false) String name) {
        return "Hello " + name;
    }
}
```

At the root of your project, add an application.yml with the following content:

application.yml

```
server.port: 9090
```

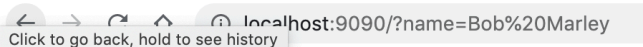
Now launch the application (Right Click on the class 'TestjpaApplication.java' and press Run TestjpaApplication.java).

In your browser, enter localhost:9090 you should see:



Hello null

You can also add a GET parameter like this : <http://localhost:9090/?name=Bob%20Marley>



Hello Bob Marley

That's it, a Hello World for Spring Project. Let's see how to handle Database Connection now:

Exercise III.2 : Activate JPA and connect to your Database

In your pom.xml, un-comment this part:

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
  <groupId>org.postgresql</groupId>
  <artifactId>postgresql</artifactId>
  <version>42.7.3</version>
</dependency>
```



“Uncomment” means remove the surrounding <!-- and -->

In your application.yml, add the following lines:

application.yml

```
spring.datasource.url: jdbc:postgresql://localhost/thibautdebroca
spring.datasource.username: postgres
spring.datasource.password:
spring.datasource.driver.class: org.postgresql.Driver
```

Note with Oracle this can be also represented like this:

```
spring.datasource:
  url: jdbc:oracle:thin:@163.172.110.161:49161:xe
  username: system
  password: oracle
  driver.class: oracle.jdbc.driver.OracleDriver
```



Be sure to replace the value of **spring.datasource.url** **username** and **password** **thibautdebroca** by your specific values !

Launch the application (Green Run button) and check if you have no error. If you have errors, read carefully the Exception message and resolve the error. If you don't understand the message, you can Google the Exception, this should help you.



[Troubleshooting \(help from others Students\)](#)

- If you have this: "Driver has been compiled by a more recent version of the Java Runtime (class file version 54.0) , this version of the Java Runtime only recognizes class file versions up to 52.0" you configure your IDE to use JDK 10.

Exercise III.3 : Read Data in your Database

In com.isep.testjpa, add a package model and insert the following class:

Emp.java

```
package com.isep.testjpa.model;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;

@Entity
public class Emp {
    @Id
    @GeneratedValue
    private Long empno;
    @Column(name = "ename")
    private String ename;
    @Column(name = "efirst")
    private String efirst;
    @Column(name = "job")
    private String job;
    @Column(name = "mgr")
    private Long mgr;
    @Column(name = "sal")
    private Long sal;
    public Long getEmpno() {
        return empno;
    }
    public void setEmpno(Long empno) {
        this.empno = empno;
    }
}
```

Also, create a package repository

EmpRepository.java

```
package com.isep.testjpa.repository;

import com.isep.testjpa.model.Emp;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

@Repository
public interface EmpRepository extends JpaRepository<Emp, Long> {

}
```

In your class SimpleController, add a new HTTP Endpoint to GET the list of Employees. This will look like this:

SimpleController.java

```
package com.isep.testjpa.controller;

import com.isep.testjpa.repository.EmpRepository;
import com.isep.testjpa.model.Emp;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.data.repository.query.Param;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RestController;

import java.util.List;

@RestController
public class SimpleController {

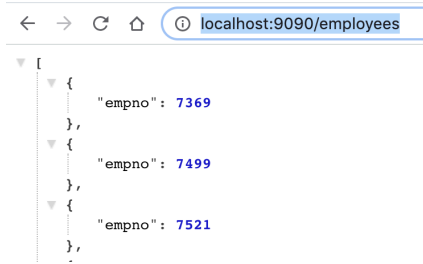
    @Autowired
    private EmpRepository empRepository;

    @RequestMapping(value="/", method= RequestMethod.GET)
    public String hello(@Param("name") String name) {
        return "Hello " + name;
    }

    @RequestMapping(value="/employees", method= RequestMethod.GET)
    public List<Emp> getEmployees() {
        return empRepository.findAll();
    }

}
```

Restart the Application and in your Browser, go to <http://localhost:9090/employees>
You should see something like this:



Great news ! We clearly see that we have our Employees from the Database in a nice JSON.

Bad news: There is only the Employee ID.

This is because, in the Employee class, there is [Getter/setter](#) only for the field *empno*.

If you want all the fields, you should add Getter/Setter for all the fields.

You can do this by generating it with your IDE (in IntelliJ: Code => Generate)



Tips : You can

Add the Maven dependencies here:

<https://mvnrepository.com/artifact/org.projectlombok/lombok/1.18.12>

In your pom.xml

Then in Emp.java, above **public class Emp {**

Add the following line:

@Data

Example/Doc here: <https://projectlombok.org/features/Data>

This will automatically generate the Getter and Setter on the fly when you compile.

And your code will be very light !

In your IDE, be sure to activate “Annotation Processing”, example in IntelliJ, go to Preferences, and search “Annotation”, then in Annotation Processors, click on “Enable annotation Processing”.

With Eclipse:

<https://howtodoinjava.com/automation/lombok-eclipse-installation-examples/>

Restart Application, and go to <http://localhost:9090/employees> you should now see all the employees attributes:

Exercise III.4 : Create an Object via JPA and expose it on a REST endpoint

In SimpleController, add the last method

SimpleController.java

```
package com.isep.testjpa.controller;

import com.isep.testjpa.repository.EmpRepository;
import com.isep.testjpa.model.Emp;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.data.repository.query.Param;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RestController;

import java.util.List;

@RestController
public class SimpleController {

    @Autowired
    private EmpRepository empRepository;

    @RequestMapping(value="/", method= RequestMethod.GET)
    public String hello(@Param("name") String name) {
        return "Hello " + name;
    }

    @RequestMapping(value="/employees", method= RequestMethod.GET)
    public List<Emp> getEmployees() {
        return empRepository.findAll();
    }

    @PostMapping(value="/employees")
    public Emp addEmployee(@RequestBody Emp emp) {
        return empRepository.save(emp);
    }
}
```

For Web annotations to work (@PostMapping, @RequestMapping ...), the dependencies needed in your pom.xml are already added:

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

<https://search.maven.org/search?q=a:spring-boot-starter-web>

Simple isn't it !?

Now, how to test it ?

You can test via different HTTP client:

- Curl command line

- Postman
- ...

We'll do even better, and automatically generate a simple frontend that uses the endpoint.
We'll use **Swagger** !

In your pom.xml, add these dependencies (and refresh dependencies with IDE):

```

pom.xml

....
<dependency>
  <groupId>io.springfox</groupId>
  <artifactId>springfox-swagger2</artifactId>
  <version>2.9.2</version>
</dependency>
<dependency>
  <groupId>io.springfox</groupId>
  <artifactId>springfox-swagger-ui</artifactId>
  <version>2.9.2</version>
</dependency>
....

```

Create a Package Configuration, and add the following class:

```

SpringFoxConfig.java

package com.isep.testjpa.configuration;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import springfox.documentation.builders.PathSelectors;
import springfox.documentation.builders.RequestHandlerSelectors;
import springfox.documentation.spi.DocumentationType;
import springfox.documentation.spring.web.plugins.Docket;
import springfox.documentation.swagger2.annotations.EnableSwagger2;

@Configuration
@EnableSwagger2
public class SpringFoxConfig {
    @Bean
    public Docket api() {
        return new Docket(DocumentationType.SWAGGER_2)
            .select()
            .apis(RequestHandlerSelectors.any())
            .paths(PathSelectors.any())
            .build();
    }
}

```

Relaunch the app and go to <http://localhost:9090/swagger-ui.html#>

You'll see an interface where you have tabs for each of your Endpoints ! Isn't it nice ?

Indeed, Thanks to Spring Boot Annotations, Swagger has parsed all the endpoints and generated this page / JSON: <http://localhost:9090/v2/api-docs>

The UI is a frontend over this JSON.

Very useful !

Exercise III.5 : GET by ID / Update by id / Delete by ID / Get All, via JPA / REST

Now you should add endpoints to Get by ID / get All / Update / Delete Employee via a REST endpoint.

To give back your work, you should copy/paste all content of your files (no screenshot) in your final report (the PDF).

Note: The best way would be of course to give back results on github but for correction it's easier to have everything in a single file for the professor.

Be sure to respect REST API Conventions to have a clean API: [REST API - URL Naming Conventions](#)

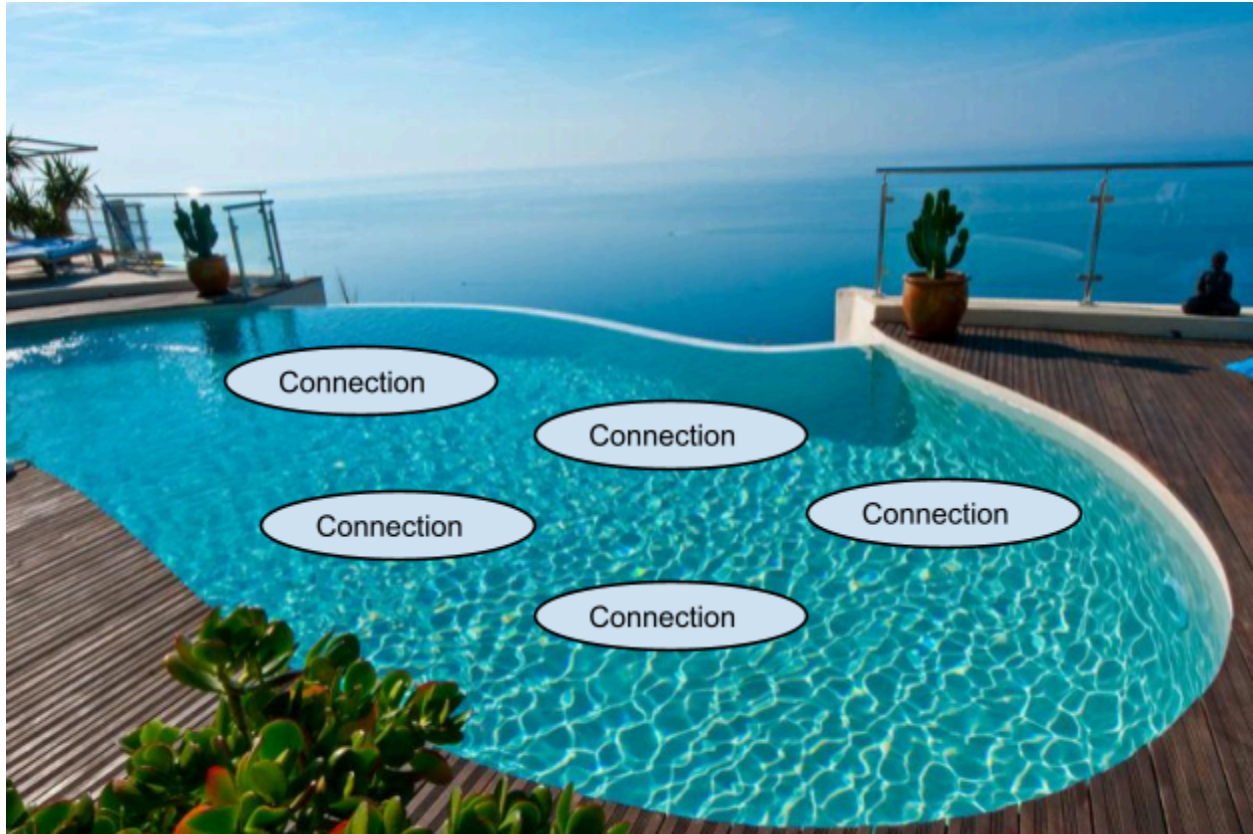
Bonus : Follow this very great tutorial of Open Classroom on how to use Spring & JPA: <https://openclassrooms.com/fr/courses/4668056-construisez-des-microservices>

Part IV : Connections Pool (Bonus)



This is a bonus, because you should prefer Spring JPA which will handle the connection pool for you. But even if you use Spring JPA. Don't forget that you can configure/tune it (to have more connection for example).

You can read this: <https://www.baeldung.com/spring-boot-hikari#spring-boot-2>



Update accepted : "Cela me donne envie de partir en vacances !!"

1. A Connection Originally Kept

Last but not least, we'll investigate the performance of what we've done. In a normal J2EE application, your application will serve many clients. Each call to the bdd will be done in Servlet (Object which treats an Http Request). Several clients can call at the same time via different threads. Let's analyse what will consume the most time: The Connection to BDD. Each connection to the bdd can cost several hundred a milliseconds which would be enormous for too many calls to the bdd or with too many clients at the same time.

That's why, you will want to open several Connections to the BDD and be able to re-use them from client to client.

At the start of your server, you'll open a certain number of Connections and store them in a pool (a list as a stack for example). A client c1 will take a Connection do its request and once he has finished with it (at the moment of the close() statement) , he will replace it in the pool for the use of another client.

Exercise IV.1 : (Bonus): Manually Create a List which store connection and use it in your code to make request

Here, You will need to stop using the DriverManager and use [DataSource](#) which is adapted for creating several Connections with multi-thread applications.

http://en.wikipedia.org/wiki/Connection_pool

<http://www.javaranch.com/journal/200601/JDBCConnectionPooling.html>

Exercise IV.2 (Bonus): use BoneCP to create your Connection Pool

The fact that we have isolate the instance of our Connection will help us to modify quickly our code to use an implementation of a Connection Pool

<https://github.com/wwadge/bonecp>

Read More on DAO:

[http://en.wikipedia.org/wiki/Hibernate_\(Java\)](http://en.wikipedia.org/wiki/Hibernate_(Java))

http://www.tutorialspoint.com/hibernate/hibernate_tutorial.pdf

Tips

(1) Get entry from User:

```
Scanner sc = new Scanner(System.in);  
int i = sc.nextInt();
```

(2) Tips Exercise 3:

First thing to do is: [Google : how to know column name result set ResultSetMetaData](#) can give information on the resultSet

```
ResultSetMetaData rsmd = results.getMetaData();  
int columnsNumber = rsmd.getColumnCount();  
String firstColumnName = rsmd.getColumnName(1);
```

Also, even if a field is marked as "Integer" in the database, you are allow results.getString(i) on the field.

(3) Link to the original Google Doc:

https://docs.google.com/document/d/1W4fz6mD7qCibN7pE1jtT7tPUP6ffoMrtn_sTJxdSMVY/