SPRING DATA

# 1. Database Access with JDBC (Java DataBase Connection)

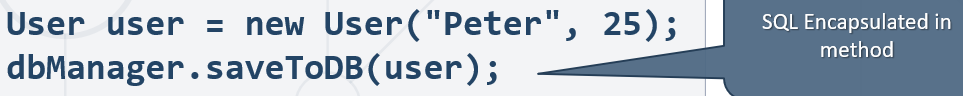
JDBC дава възможност да се свързваме към всякакъв вид бази

## 1.1. Accessing Data Via Client Application



### ORM (Object relational mapping) Frameworks Overview

* In development, programmers use **object relational mapping** frameworks.
  + Mapping Java classes and data types to **DB tables** and **SQL data types**
  + Generate SQL calls and **relieves** the developer from the **manual** **handling**
    - E.g. (pseudo-code)



* ORM frameworks **do not** drop the need to write SQL!
  + At some point you might need some **manual query optimization**
* ORM Frameworks **examples**:
  + Java – **Hibernate**, EclipseLink, TopLink, OpenJP…
  + .NET – Entity Framework, NHibernate…
  + PHP – Doctrine, Laravel(Eloquent)...

**Java bin концепция/конвенция**

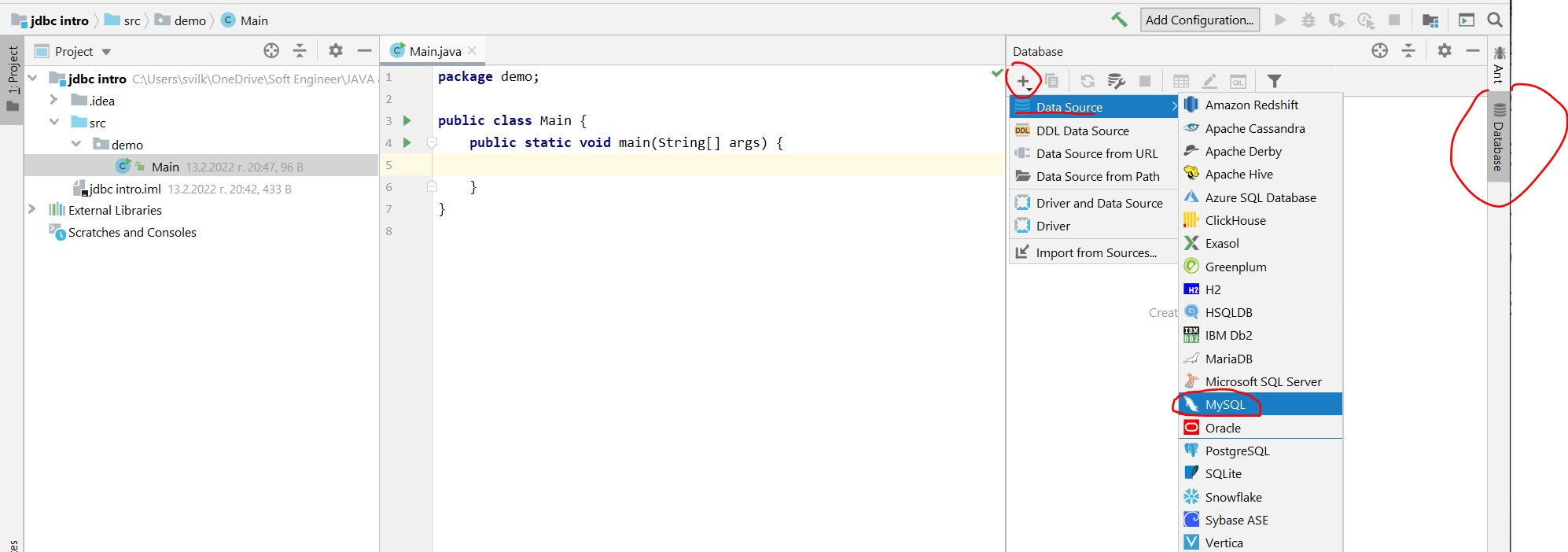
Кое пропърти от класа на кое място в базата данни да отива

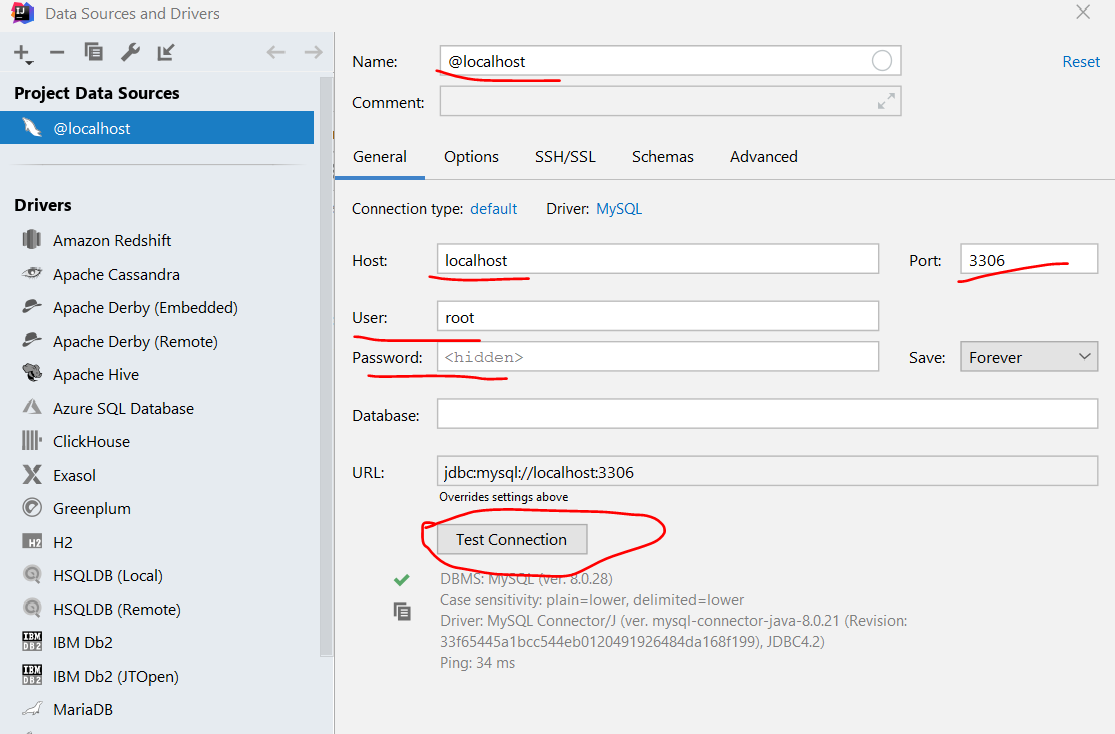
Клас към таблица

## 1.2. Demos – how to connect to a database

### Connection to DB Via Java App Demo – part 1

Import-ваме базата данни чрез Plug-in -a Database отдясно отсрани

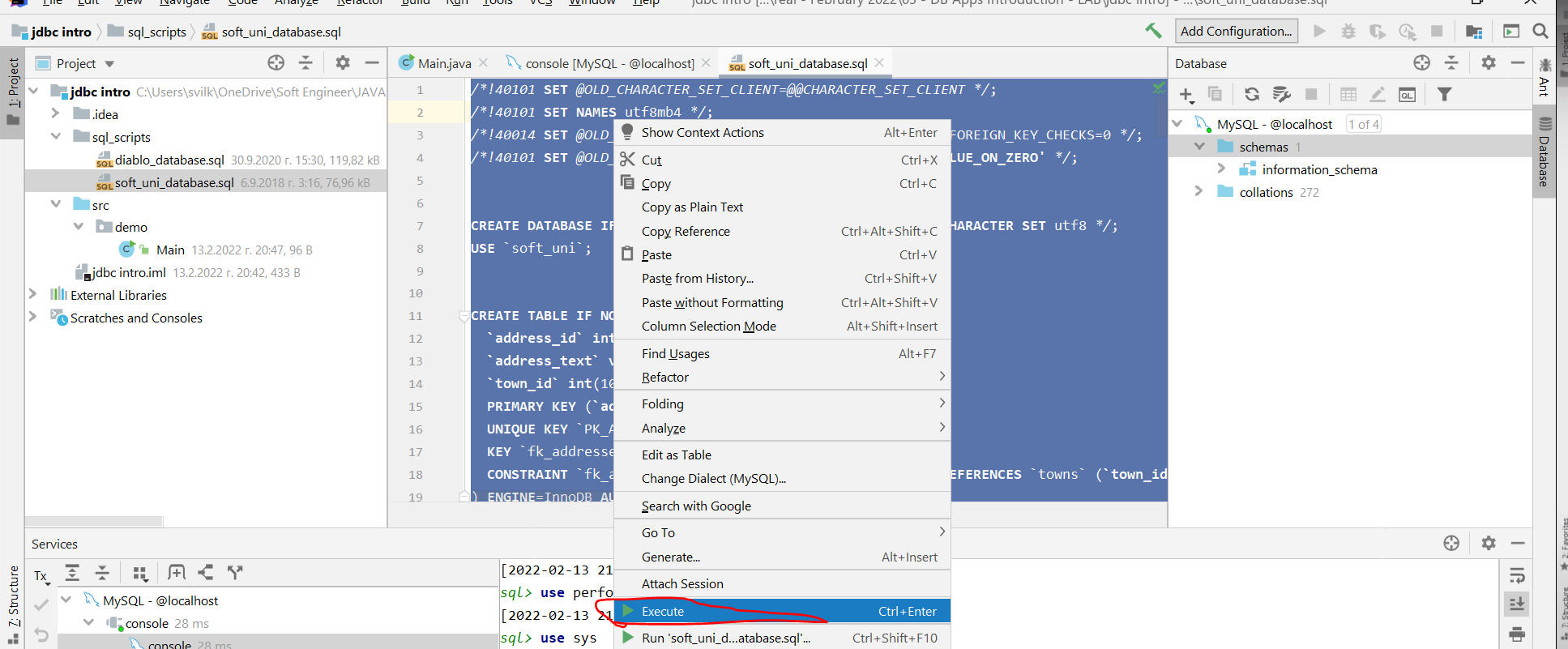




**Избираме диалекта – MySQL / Maria DB match best**

**Изпълнение на sql скрипт/заявка в IntelliJ**

**Ctrl + A – select all text**



**MySQL -> console**

### Connection to DB Via Java App Demo – part 2

**Първо обясненията, след това целият код написан на Java**

* You are given a simple application that:
  + Establishes connection with the "soft\_uni" DB
  + Executes simple MySQL statement to retrieve the employees names by given salary criteria
* Let's analyze the program:

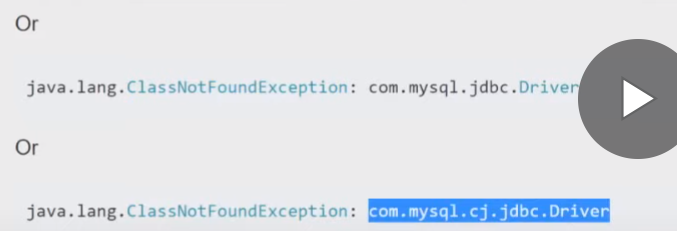
Connection to DB is established by asking the user to give credentials:

* Using an external library (**MySQL Connector/J**) we make a connection via a **DriverManager** and a **Connection** class.

Новият и старият driver:

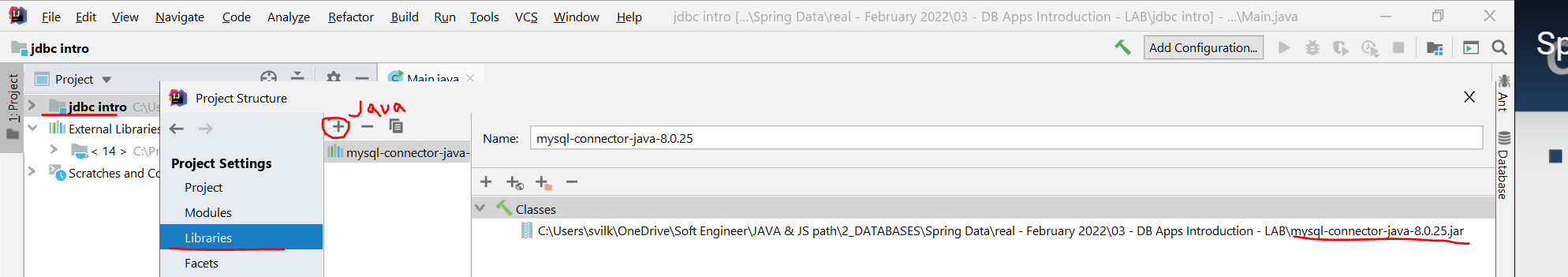
– за MySQL 8 e "com.mysql.cj.jdbc.Driver"

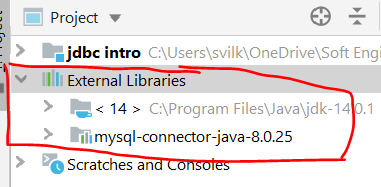
* за MySQL 5.0 и надолу е другия. То само си ги разпознава.

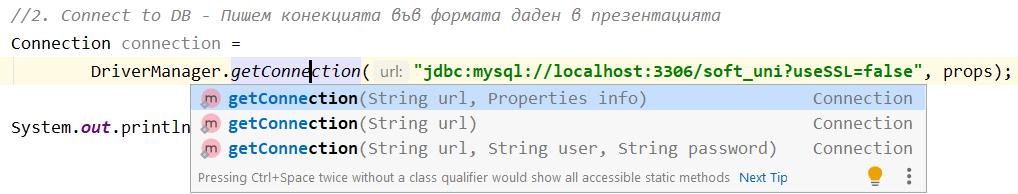


Plain Инсталиране/слагане на **JAVA driver-а** в IntelliJ за **MySQL Connector/J**

File -> Project structure …..







* We retrieve the result with the **ResultSet** and the **PreparedStatement** classes.

Всяка въпросителна е пореден параметър.

**За база данни, номер на позицията винаги започва от 1, а не от 0.**

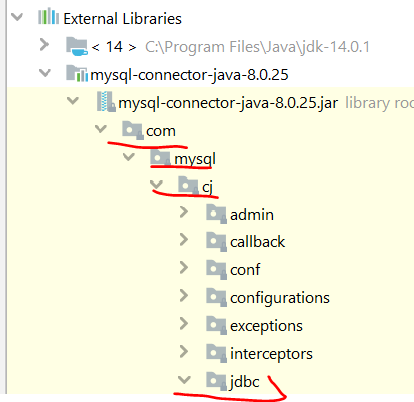
In the following example of setting a parameter, connection represents an active connection:

public interface PreparedStatement extends Statement;

PreparedStatement pstmt = connection.prepareStatement(**"UPDATE EMPLOYEES SET** SALARY = ? **WHERE** ID = ?**"**);  
pstmt.setBigDecimal(1, 153833.00);  
pstmt.setInt(2, 110592);

### The whole Demo itself

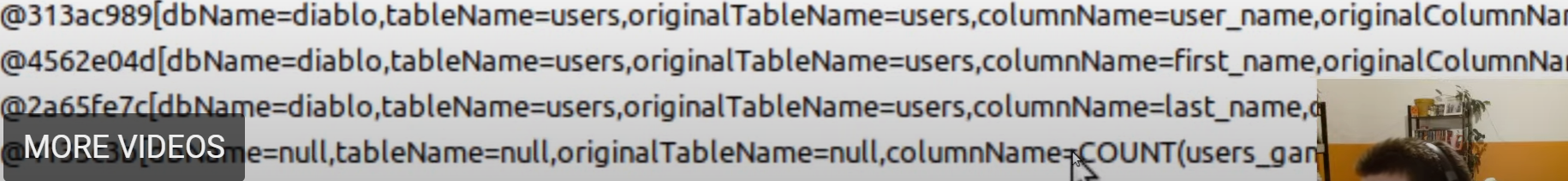
**package** demo;  
  
**import** java.sql.\*;  
  
**import** java.util.Properties;  
**import** java.util.Scanner;  
  
**public class** Main {  
 **public static void** main(String[] args) **throws** SQLException {  
 Scanner sc = **new** Scanner(System.***in***);  
 System.***out***.print(**"Enter username default (root): "**);  
 String user = sc.nextLine().trim();  
 user = user.equals(**""**) ? **"root"** : user;  
 System.***out***.print(**"Enter password default (empty):"**);  
 String password = sc.nextLine().trim();  
  
 Properties props = **new** Properties();  
 props.setProperty(**"user"**, user);  
 props.setProperty(**"password"**, password);  
  
 *//****1. Load jdbc driver - optional*****try** {  
 Class<?> aClass = Class.*forName*(**"com.mysql.cj.jdbc.Driver"**);

  
 } **catch** (ClassNotFoundException e) {  
 e.printStackTrace();  
 System.*exit*(0);  
 }  
 System.***out***.println(**"Driver loaded successfully"**);  
  
  
 *//****2. Connect to DB - Пишем конекцията във формата даден в презентацията***Connection connection =  
 DriverManager.*getConnection*(**"jdbc:mysql://localhost:3306/soft\_uni?useSSL=false"**, props);

*//Вариант без props - username root, след това двоеточие, след това парола, след това @*Connection connection =  
 DriverManager.*getConnection*(**"jdbc:mysql://root:@localhost:3306/soft\_uni?useSSL=false"**);

System.***out***.println(**"Connected successfully"**);  
  
  
 ***//3.PreparedStatement – класът, който отговаря НО САМО за една SQL заявка***PreparedStatement stmt = connection.prepareStatement(**"SELECT** *\** **FROM employees WHERE salary > ?"**); *//SQL Query* System.***out***.print(**"Enter minimal salary (default 20 000): "**);  
 String salaryStr = sc.nextLine().trim();  
 **double** salary = salaryStr.equals(**""**) ? 20000 : Double.*parseDouble*(salaryStr);  
 stmt.setDouble(1, salary); **//сетваме на първата въпросителна дадена стойност**  
 ResultSet rs = stmt.**executeQuery()**;***//Runs the SQL statement and returns retrieved result***

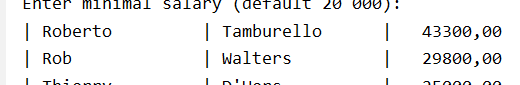
rs.**getMetaData()**; //взема неща за самата таблица, като имена на колони (но ние можем да си ги видим и от plugin-a на IntelliJ Database, както и от друг клиент на базата данни). И все пак:



***//Iterating over the result*** *– one way forward only and not updatable – не натоварва така паметта:*

**if***(*rs.next()*)* **while** (rs.next()) {  
 System.***out***.printf(**"| %-15.15s | %-15.15s | %10.2f\n"**,  
 rs.getString(**"first\_name"**), *//column label – може и по позиция на колона* rs.getString(**"last\_name"**), *//column label* rs.getDouble(**"salary"**)); *//get by column label* }  
  
 connection.close();  
 }  
}

Изходът от конзолата



### The whole DEMO with try with resources (try-with-resources)

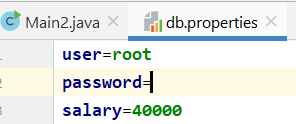
**package** demo;  
  
**import** java.sql.\*;  
**import** java.util.Properties;  
**import** java.util.Scanner;  
  
**public class** Main1 {  
 **public static void** main(String[] args) **throws** SQLException {  
 Scanner sc = **new** Scanner(System.***in***);  
 System.***out***.print(**"Enter username default (root): "**);  
 String user = sc.nextLine().trim();  
 user = user.equals(**""**) ? **"root"** : user;  
 System.***out***.print(**"Enter password default (empty):"**);  
 String password = sc.nextLine().trim();  
  
 Properties props = **new** Properties();  
 props.setProperty(**"user"**, user);  
 props.setProperty(**"password"**, password);  
  
 *//1. Load jdbc driver - optional* **try** {  
 Class<?> aClass = Class.*forName*(**"com.mysql.cj.jdbc.Driver"**);  
 } **catch** (ClassNotFoundException e) {  
 e.printStackTrace();  
 System.*exit*(0);  
 }  
 System.***out***.println(**"Driver loaded successfully"**);  
  
 *//Using try with resources  
 //2. Connect to DB and 3. PreparedStatement in try with resources  
 //Interface-а Connection e AutoCloseable* **try** (Connection connection = DriverManager  
 .*getConnection*(**"jdbc:mysql://localhost:3306/soft\_uni?useSSL=false"**, props);  
 PreparedStatement stmt =  
 connection.prepareStatement(**"SELECT** *\** **FROM employees WHERE salary > ?"**)) {  
 System.***out***.println(**"Connected successfully"**);  
  
 System.***out***.print(**"Enter minimal salary (default 20 000): "**);  
 String salaryStr = sc.nextLine().trim();  
 **double** salary = salaryStr.equals(**""**) ? 20000 : Double.*parseDouble*(salaryStr);  
 stmt.setDouble(1, salary);  
 ResultSet rs = stmt.executeQuery();*//Runs the SQL statement and returns retrieved result* **while** (rs.next()) {  
 System.***out***.printf(**"| %-15.15s | %-15.15s | %10.2f\n"**,  
 rs.getString(**"first\_name"**), *//column label* rs.getString(**"last\_name"**), *//column label* rs.getDouble(**"salary"**)); *//get by column label* }  
 }  
 }  
}

### The whole DEMO with external file

**import** java.io.FileInputStream;  
**import** java.io.IOException;  
**import** java.sql.\*;  
**import** java.util.Properties;  
  
**public class** Main2 {  
 **public static void** main(String[] args) **throws** SQLException, IOException {  
 Properties props = **new** Properties();

//търси го в out/production когато програмата е заредила/компилирала  
 String appConfigPath = Main2.**class**.getClassLoader()  
 .getResource(**"db.properties"**).getPath();

String appConfigPath = **"C:\\Users\\svilk\\OneDrive\\Soft Engineer\\JAVA & JS path\\2\_DATABASES\\Spring Data\\real - February 2022\\03 - DB Apps Introduction - LAB\\jdbc intro\\src\\db.properties"**;

  
  
 props.load(**new** FileInputStream(appConfigPath));  
*// props.loadFromXML(new FileInputStream(appConfigPath));  
  
 //1. Load jdbc driver - optional* **try** {  
 Class<?> aClass = Class.*forName*(**"com.mysql.cj.jdbc.Driver"**);  
 } **catch** (ClassNotFoundException e) {  
 e.printStackTrace();  
 System.*exit*(0);  
 }  
 System.***out***.println(**"Driver loaded successfully"**);  
  
 *//Using try with resources  
 //2. Connect to DB and 3. PreparedStatement in try with resources  
 //Interface-а Connection e AutoCloseable* **try** (Connection connection = DriverManager  
 .*getConnection*(**"jdbc:mysql://localhost:3306/soft\_uni?useSSL=false"**, props);  
 PreparedStatement stmt =  
 connection.prepareStatement(**"SELECT** *\** **FROM employees WHERE salary > ?"**)) {  
 System.***out***.println(**"Connected successfully"**);  
  
 System.***out***.print(**"Enter minimal salary (default 20 000): "**);  
 String salaryStr = props.getProperty(**"salary"**, **"20000"**);  
 **double** salary = Double.*parseDouble*(salaryStr);  
 stmt.setDouble(1, salary);  
 ResultSet rs = stmt.executeQuery();*//Runs the SQL statement and returns retrieved result* **while** (rs.next()) {  
 System.***out***.printf(**"| %-15.15s | %-15.15s | %10.2f\n"**,  
 rs.getString(**"first\_name"**), *//column label* rs.getString(**"last\_name"**), *//column label* rs.getDouble(**"salary"**)); *//get by column label* }  
 }  
 }  
}

### Demo Conclusion

* We can access databases on a programmer level.
  + No manual actions needed
* In a bigger applications we can:
  + Encapsulate custom SQL logic in methods
  + Achieve database abstraction

SSL – Security Socket Layer – двойка mapping key

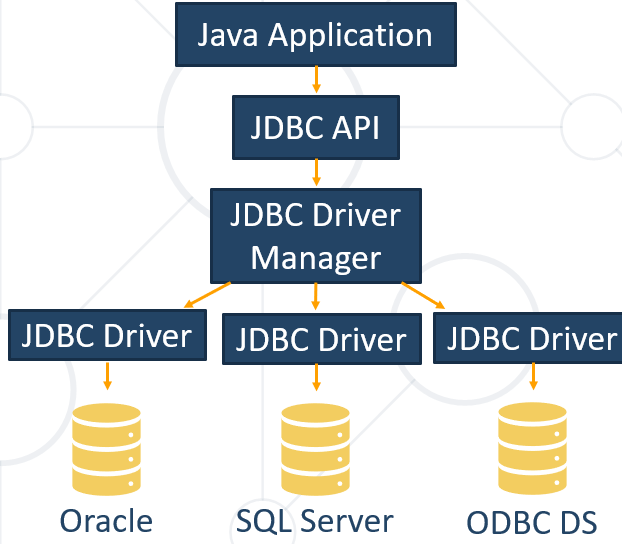
TLS - Transport Layer Security

## 1.3. Client Access to a Database

### Java Database Connectivity (JDBC)

* JDBC is a standard Java API for database-independent connectivity
* Includes APIs for:
  + Making a connection to a database
  + Creating and executing **SQL** queries in the database
  + Viewing & Modifying the resulting records

### JDBC Architecture



Write once, runs it anywhere – sun microsystems Java

* **JDBC API** – provides the connection between the application and the driver manager
* **JDBC Driver Manager** – establishes the connection with the correct driver
  + Supports multiple drivers connected to different types of databases
* **JDBC Driver** - handles the communications with the database

### JDBC API

* JDBC API provides several interfaces and classes:
  + **DriverManager** – matches requests from the application with the proper DB driver
  + **Driver** – handles the communication with the DB server
  + **Connection** – all methods for contacting a database
  + **Statement and PreparedStatement** – methods and properties that enable you to send SQL
  + **ResultSet** – retrieved data (set of table rows)
  + **SQLException**

### JDBC API – ResultSet Class

* ResultSet maintains a **cursor** pointing to its **current row of data**
  + Not updatable
  + Iterable only once and only from the first row to the last row
* Provides getter methods for retrieving column values from the current row
* E.g. from previous demo:

*//Iterating over the result – one way forward and only – не натоварва така паметта:* **while** (rs.next()) {  
 System.***out***.printf(**"| %-15.15s | %-15.15s | %10.2f\n"**,  
 rs.getString(**"first\_name"**), *//column label – може и по позиция на колона* rs.getString(**"last\_name"**), *//column label* rs.getDouble(**"salary"**)); *//get by column label* }

* Retrieved information is reached by getter methods:
  + E.g.:
    - getString("column\_name")
    - getDouble("column\_name")
    - getBoolean("column\_name") etc.
* The driver converts the underlying data to the Java type

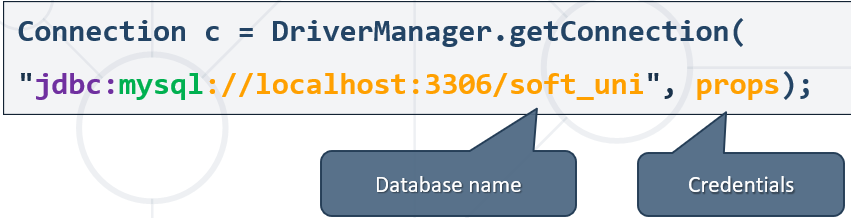
### Java.sql\* and MySQL Driver

* The java.sql package provides all previously mentioned JDBC classes
* In order to work with JDBC we need to download a MySQL Driver – Connector/J
  + It can be found on the following webpage:

[**https://dev.mysql.com/downloads/connector/j/**](https://dev.mysql.com/downloads/connector/j/)

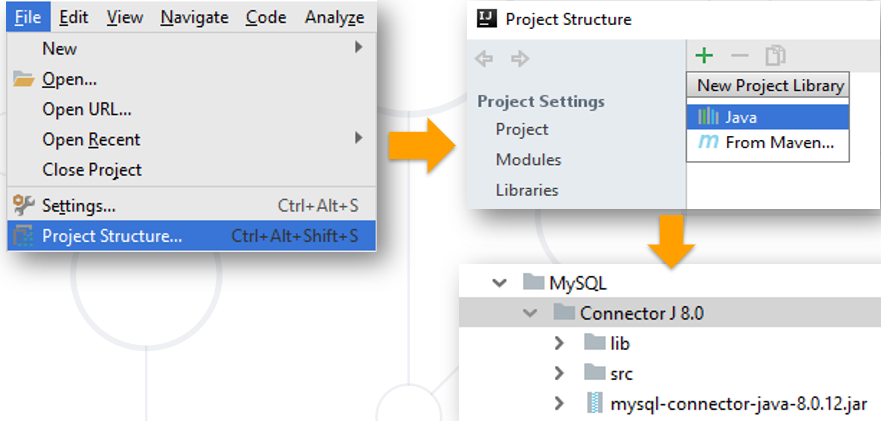
### MySQL Driver Connection

* Connection with the database is established via **connection** **string**
  + jdbc:<driver protocol>:<connection details>
  + E.g. connection from previous demo:



### Setting Up the Driver in IntelliJ IDEA

* Add the driver as an external library:
  + "File" -> "Project Structure" -> "Libraries"



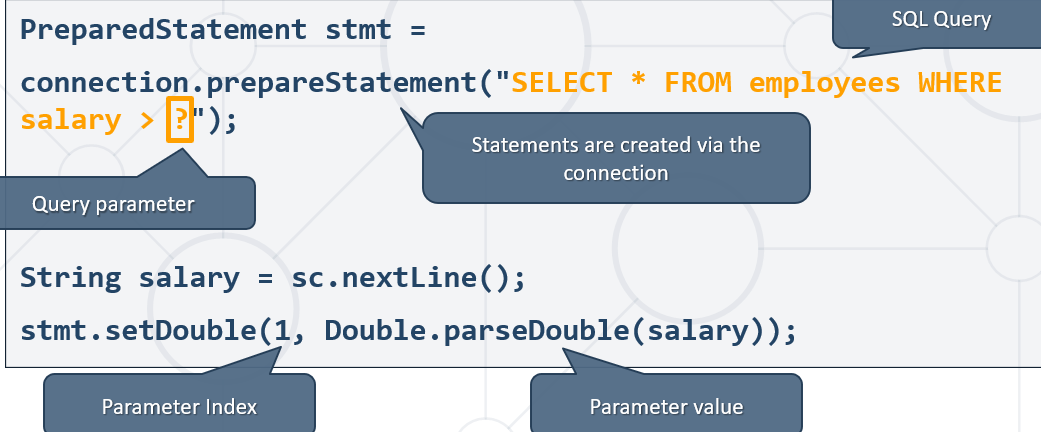
## 1.4. Statement, PreparedStatement, CallableStatement

### Statements

### The JDBC **Statement interface** defines the methods and properties that enable you to send SQL commands to the database.



#### Example(PreparedStatement) from previous demo:



***//3.PreparedStatement – класът, който отговаря НО САМО за една SQL заявка***PreparedStatement stmt = connection.prepareStatement(  
 **"SELECT v.name,** *COUNT***(DISTINCT mv.minion\_id) AS `count\_of\_minions`\n"** +  
 **"FROM minions\_db.villains AS v\n"** +  
 **"LEFT JOIN minions\_db.minions\_villains AS mv\n"** +  
 **"ON v.id = mv.villain\_id\n"** +  
 **"GROUP BY mv.villain\_id\n"** +  
 **"HAVING `count\_of\_minions` >15\n"** +  
 **"ORDER BY `count\_of\_minions` DESC;"**);  
  
ResultSet rs = stmt.executeQuery();  
  
**while** (rs.next()) {  
 System.***out***.printf(**"%-15.15s %d\n"**,  
 rs.getString(**"name"**),  
 rs.getInt(**"count\_of\_minions"**)); //по приятен начин за get-ване като сме използвали в заявката AS **`count\_of\_minions`**

## 1.5. How to Prevent It? – SQL Injection

### What is SQL Injection?

* Placement of **malicious** code in SQL Statements
  + Usually done via user input
* To protect our data, we can place parameters in our statements
  + We can do it by using **PreparedStatement**

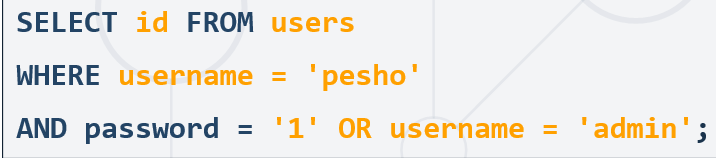
### SQL Injection Example: Login Form Input by User

* Ask the user to input username and password in fields
  + If we don't secure our statements, we risk SQL Queries to be written as an input
  + E.g. :
    - username: "example\_user "
    - password: "12345"
    - The following query will be built and executed to the data source:



----------------

* In result the **id of the user** will be returned.
  + User will be authenticated to do actions in the application
* Without validating and securing our statements information might get exposed:
  + Value for password: ''1' OR username = 'admin';'
  + The following query will be executed:



-------------

* In result the id **an admin** will be returned
  + Will permit actions to the user that can harm our application and database
* We can validate the input by setting rules
  + Length, special characters, digits etc.
  + Set up validation in our code in different layers (front-end, back-end etc.)

## 1.6. Transactions and DAO Pattern

### JDBC Transaction Pattern

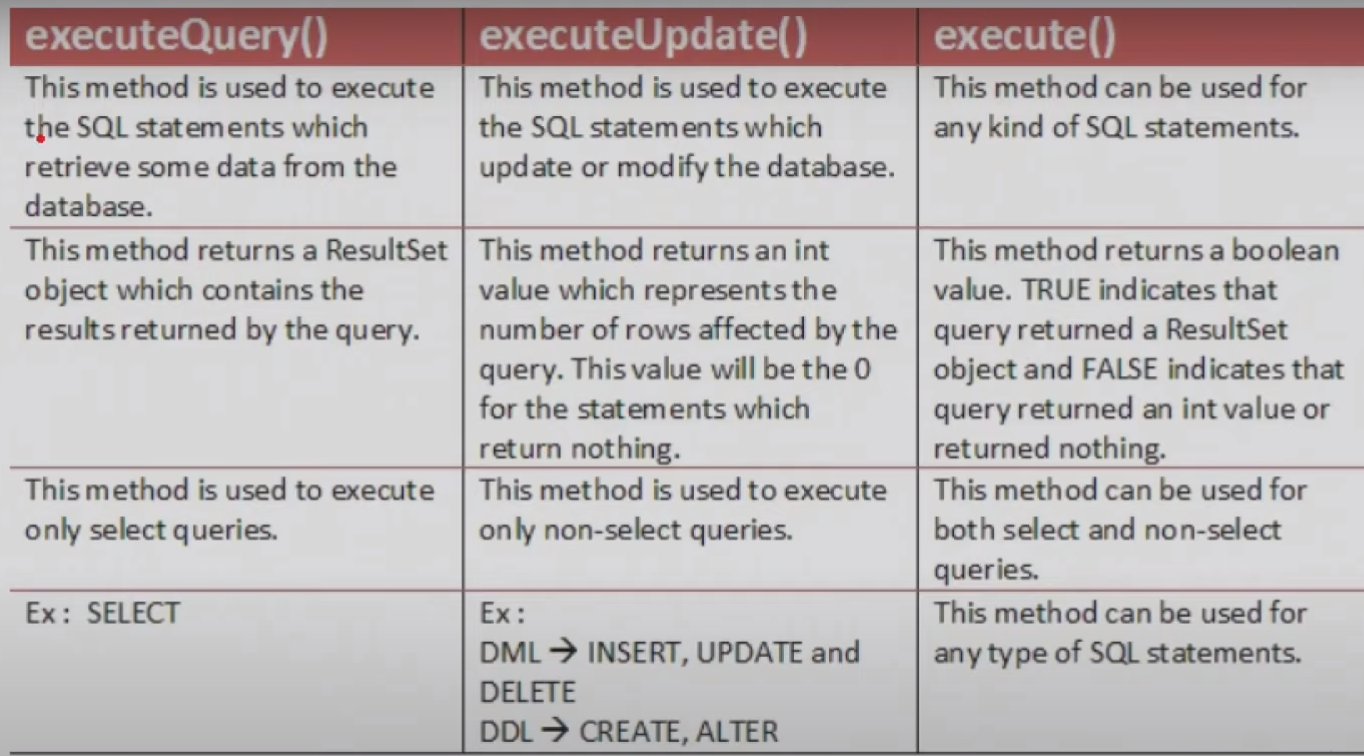
* Every JDBC Connection is set to **auto-commit** by default
  + SQL statements are committed on completion
* In bigger applications we want greater control
  + If and when changes are applied to the database
* Turn off auto-commit:

connection.setAutoCommit(**false**);

* Example (**pseudo** **code**):

**try** {  
 connection.**setAutoCommit**(**false**);  
 Statement stmt = conn.createStatement();  
 String sql = **"…"**;  
 stmt.**executeUpdate**(sql);  
 *// If there is no error* connection.**commit();**  
} **catch**(SQLException se){  
 *// If there is any error* conn.**rollback();**  
}

## 1.7. Queries in JDBC



## 1.8. Insert заявка

**private static int** getOrInsertTown(Connection connection, String minionTown) **throws** SQLException {  
 PreparedStatement **selectTownStatement** = connection.prepareStatement(  
 **"SELECT id FROM towns WHERE name=?"**);  
 selectTownStatement.setString(1, minionTown);  
  
 ResultSet townSet = selectTownStatement.executeQuery();  
  
 **int** townId = 0;  
 *//Ако няма такъв град* **if** (!townSet.next()) {  
 PreparedStatement **insertTownStatement** = connection.prepareStatement(**""** +  
 **"INSERT INTO towns(name) VALUES(?)"**);  
 insertTownStatement.setString(1, minionTown);  
 insertTownStatement.**executeUpdate();** ***//в случая с тази заявка Добавяме град Insert***ResultSet newTownSet = **selectTownStatement**.**executeQuery(); // Select**  
 newTownSet.next(); ***//премести показателя на последния добавен град***townId = newTownSet.getInt(**"id"**);  
 System.***out***.println(String.*format*(**"Town %s was added to the database."**, minionTown));  
 } **else** {  
 townId = townSet.getInt(**"id"**);  
 }  
  
 **return** townId;  
}

## 1.9. Update заявка

PreparedStatement **updateTownNames** = connection.prepareStatement(**"UPDATE towns\n"** +  
 **"SET name =** *UPPER***(name)\n"** +  
 **"WHERE country = ?;"**);  
updateTownNames.setString(1, countryName);  
  
**int** updatedCount = updateTownNames.**executeUpdate();** *//връща броя на update-ваните записи или 0 ако не се Update-ва нищо*System.***out***.println(updatedCount);  
  
**if** (updatedCount == 0) {  
 System.***out***.println(**"No town names were affected."**);  
 **return**;  
}  
  
System.***out***.println(updatedCount + **" town names were affected."**);

## 1.10. Delete заявка + Транзакция в Java

Първо трием мапинг таблицата, след това едната таблица с foreign key, след това и втората таблица с foreign key!!!

Connection connection =  
 DriverManager.*getConnection*(**"jdbc:mysql://root:@localhost:3306/minions\_db?useSSL=false"**);

connection.**setAutoCommit**(**false**);  
  
**try** {  
 PreparedStatement deleteMinionsVillains = connection.prepareStatement(  
 **"DELETE FROM minions\_villains WHERE villain\_id=?"**);  
 deleteMinionsVillains.setInt(1, villainId);  
 **int** countReleasedMinions = deleteMinionsVillains.**executeUpdate();**  
  
 PreparedStatement deleteVillain = connection.prepareStatement(  
 **"DELETE FROM villains WHERE id=?"**);  
 deleteVillain.setInt(1, villainId);  
 deleteVillain.executeUpdate();  
  
 connection.**commit();**  
 System.***out***.println(villainName + **" was deleted"**);  
 System.***out***.println(countReleasedMinions + **" minions released"**);  
} **catch** (SQLException e){  
 connection.**rollback();**  
}

## 1.11. Create procedure and CallableStatement

**import** java.sql.\*;  
**import** java.util.Scanner;  
  
**public class** \_09\_IncreaseAgeStoredProcedure {  
 **public static void** main(String[] args) **throws** SQLException {  
 Connection connection =  
 DriverManager.*getConnection*(**"jdbc:mysql://root:@localhost:3306/minions\_db?useSSL=false"**);  
  
 Scanner sc = **new** Scanner(System.***in***);  
 **int** idMinionToUpdate = Integer.*parseInt*(sc.nextLine());  
  
 ***//Създаване на процедура***String createProcedure =  
 **"CREATE PROCEDURE usp\_get\_older(minion\_id INT) \n"** +  
 **"BEGIN\n"** +  
 **"\tUPDATE minions\n"** +  
 **"\tSET age=age+1 \n"** +  
 **"\tWHERE id=minion\_id;\n"** +  
 **"END"**;  
 Statement stmt = connection**.createStatement();**  
 stmt.**executeUpdate(createProcedure);**  
  
 ***//prepare and execute CallableStatement*****CallableStatement** callableStatement = **connection.prepareCall**(**"{CALL** *usp\_get\_older***(?)}"**);  
 **callableStatement**.setInt(1, idMinionToUpdate);  
 **callableStatement.executeQuery(); или**

**callableStatement.execute();**  
  
 *//output the result* PreparedStatement selectOutputUpdatedMinion = connection.prepareStatement(  
 **"SELECT name, age FROM minions WHERE id=?"**);  
 selectOutputUpdatedMinion.setInt(1, idMinionToUpdate);  
 ResultSet resultSet = selectOutputUpdatedMinion.executeQuery();  
 resultSet.next();  
 System.***out***.println(resultSet.getString(**"name"**) + **" "** + resultSet.getString(**"age"**));

*//dropping the procedure*String queryDrop = **"DROP PROCEDURE IF EXISTS usp\_get\_older"**;  
Statement stmtDrop = connection**.createStatement();**  
System.***out***.println(**"Calling DROP PROCEDURE"**);  
stmtDrop.**execute(queryDrop);**

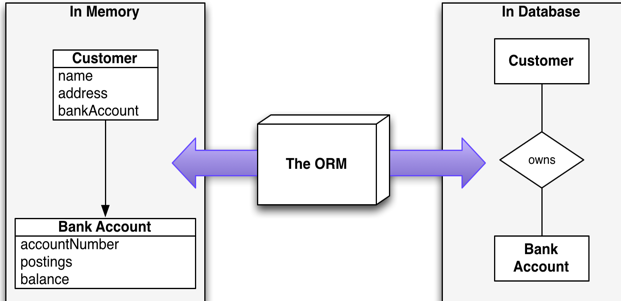
}  
}

## 1.12. С подготвяне на query заявка и String.format

**String query** = String.*format*(**"UPDATE minions\n SET age=age+1, name=LOWER(SUBSTRING(name, 1))\n WHERE id IN %s;"**,  
 minionsIDsUpdate);  
  
PreparedStatement updateSepcifiedMinions = connection.prepareStatement(**query**);  
updateSepcifiedMinions.executeUpdate();

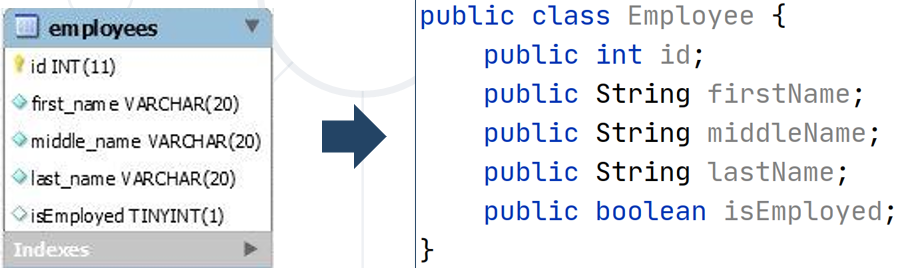
# 2. ORM Fundamentals

## 2.1. ORM Introduction - Object-Relational Mapping



### What is ORM?

* **Technique** for **converting data** betweenincompatible type systems using **object-oriented programming** languages
* **Object-Relational Mapping** (ORM) allows manipulating databases **using common classes and objects**
  + **Java/C#/etc. classes** 🡺 **Database Tables**
  + **Database Tables** 🡺 **Java/C#/etc. classes**



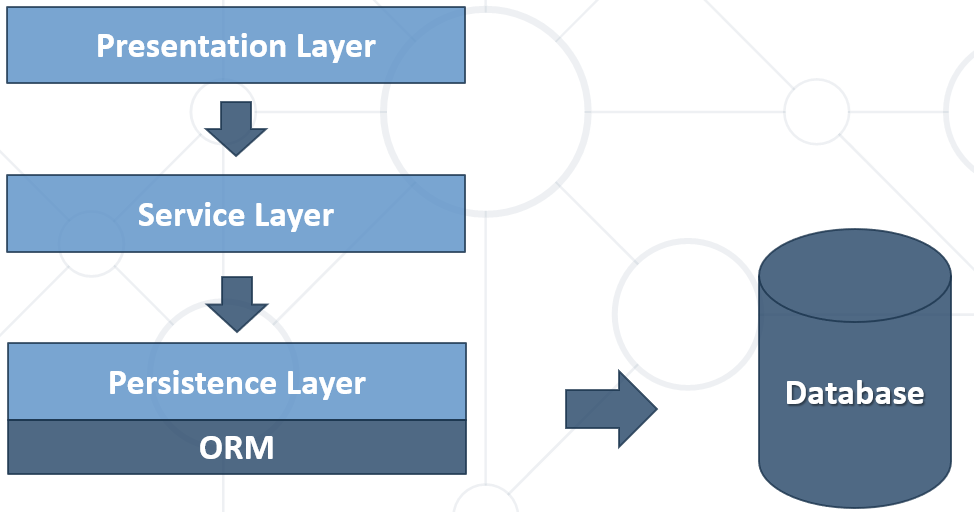
### Why do we need ORM?

* In OOP, data-management tasks act on **objects** that are almost always **non-scalar** values
* Many **database** can only store and manipulate **scalar** values, organized within **tables – scalar са обикновените данни, а non-scalar са релационни данни примерно**
* We must **manually** convert values into groups of simpler values to store in DB and convert them back when we retrieve data

### JDBC and ORM

* The main difference, between JDBC and ORM, is **complexity**
* **JDBC/SQL**
  + If the application is simple as to present data directly from the database
* **ORM**
  + If the application is domain driven and the relations among objects is complex

### Application Architecture

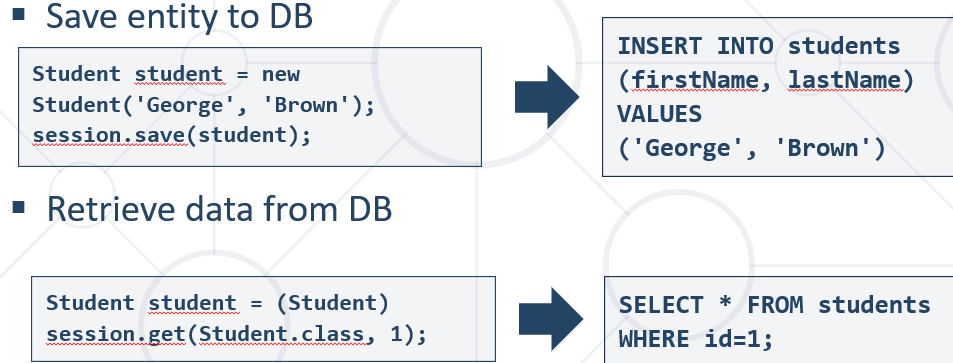


### ORM Frameworks: Features

* **ORM frameworks** typically **provide** the following functionality:
  + **Automatically generate SQL** to perform data operations as:
    - persist, update, delete, merge, createQuery and so on.
  + **Object model from database schema** (**DB First model**)
  + **Database schema from object model** (**Code First model**)

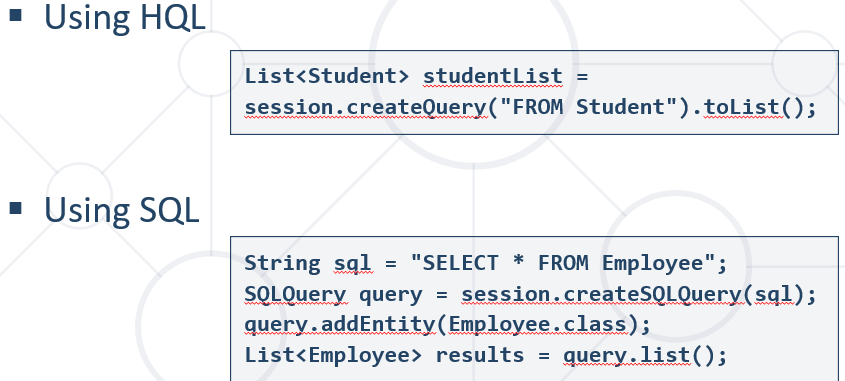
### Perform data operations with ORM

* + **Automatically generate SQL** to perform data operations



* + We can use and specific ORM Query Language as **HQL** or **SQL**

**Hybernate QueryLanguage = HQL**



### Как става в mapping-a - POJO + XML

* A bit old-fashioned, but very powerful
* Implemented in the "classical" ORM

Преди време, xml са свързвали информацията на класа към базата данни.

**...**

**<description>Mapping file</description>**

**<entity class="Employee">**

**<table name="EMPLOYEETABLE"/>**

**<attributes>**

**<id name="id">**

**<generated-value strategy="TABLE"/>**

**</id>**

**<basic name="name">**

**<column name="EMP\_NAME" length="100"/>**

**</basic>**

**<basic name="salary">**

**</basic>**

**</attributes>**

**</entity>**

**...**

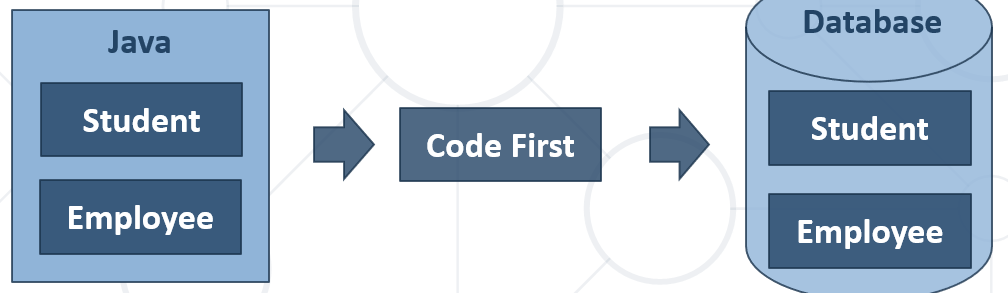
### POJO Mapped to DB Tables – using Annotations/Decorators

* Based on Java annotations and XML
* Easier to implement and maintain

@Entity  
@Table(name = **"employees"**)  
**public class** Employee {  
 @Id  
 **private int id**;  
 @Column(name = **"name"**)  
 **private** String **name**;  
 @Column(name = **"position"**)  
 **private** String **position**;  
}

### Code First Model

* **Models** the database after the entity classes – класове в java, които да бъдат съхранявани след това в база данни. Логиката да е същата.



## 2.2. ORM Advantages and Disadvantages

### ORM Advantages

* **Productivity**
  + Eliminates repetitive code
  + Generates database automatically
* **Maintainability**
  + Fewer lines of code
  + Easier to manage object model changes
* **Performance**
  + Lazy loading – докате не извикаме next(), не дърпаме информация от базата/сървъра
  + Caching
* **Database vendor independence – спирам да пиша SQL код без значение на SQL езика!!!**
  + The database is abstracted
  + Can be configured outside the application

### ORM Disadvantages

* **Reduced performance** 
  + Due to overhead or auto generated SQL
* **Reduces flexibility** 
  + Some operations are hard to implement
* **Lose understanding** 
  + What the code is actually doing - the developer is more in control using SQL

## 2.3. custom-orm with Maven(мениджър на нашите библиотеки)

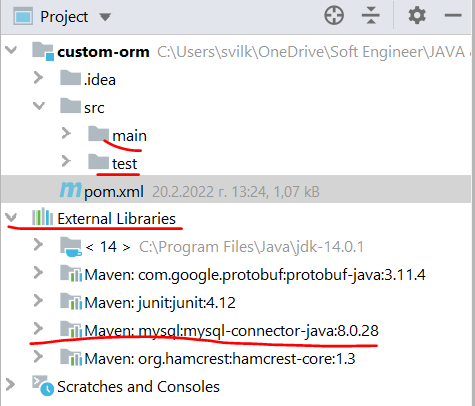
**Имената на елементите на анотациите са имената на променливите в базата данни!!!**

Следната задача не е рефактурирана с цел – че няма да мога да се ориентирам ако изнесем в много методи

### pom.xml файла

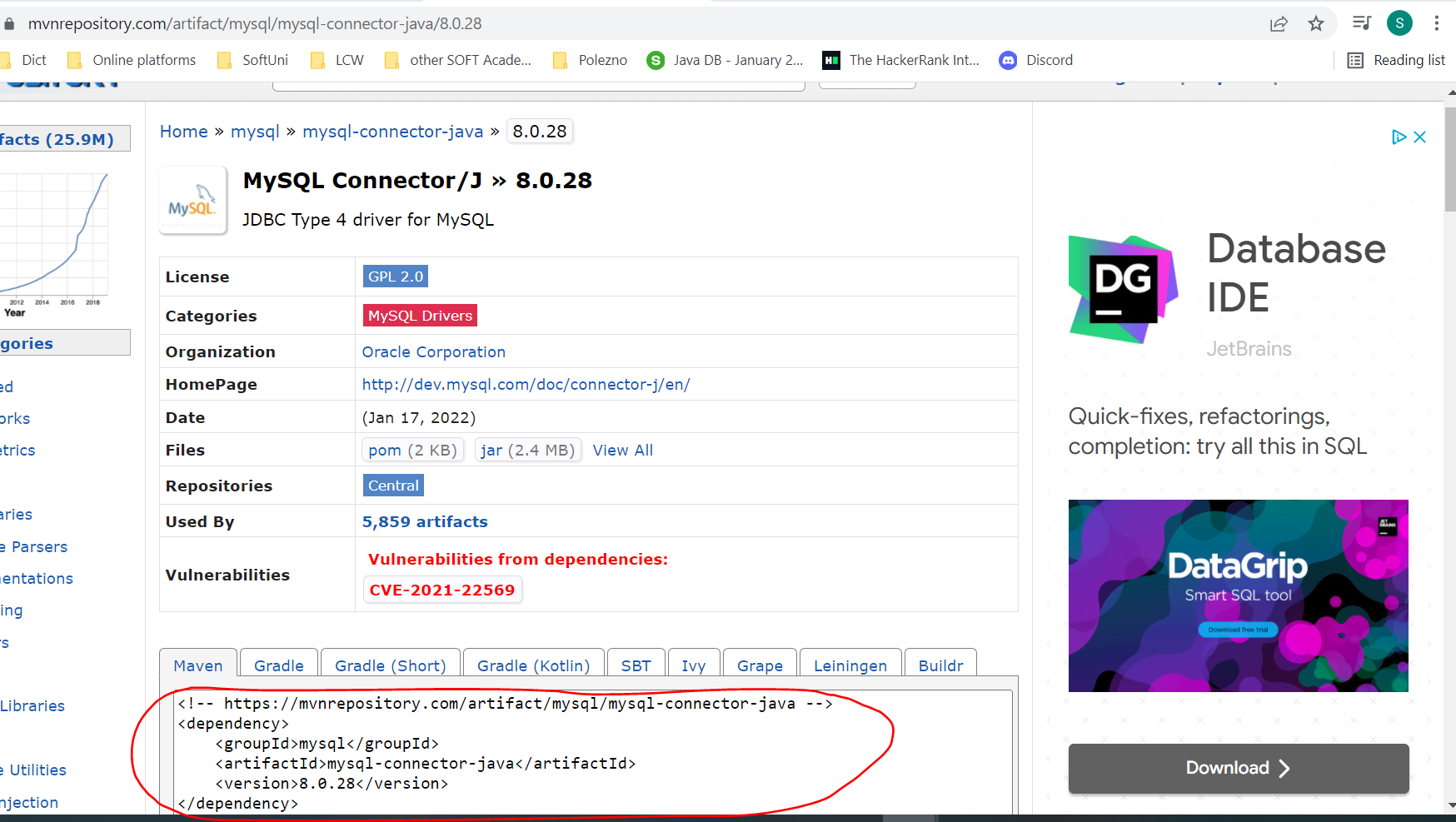
<**project xmlns="http://maven.apache.org/POM/4.0.0"  
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
 xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd"**>  
 <**modelVersion**>4.0.0</**modelVersion**>  
  
 <**groupId**>org-example</**groupId**>  
 <**artifactId**>custom-orm</**artifactId**>  
 <**version**>1.0-SNAPSHOT</**version**>  
  
 <**properties**>  
 <**maven.compiler.source**> 14 </**maven.compiler.source**>  
 <**maven.compiler.target**> 14 </**maven.compiler.target**>  
 </**properties**>  
  
 <**dependencies**>  
 <**dependency**>  
 <**groupId**>junit</**groupId**>  
 <**artifactId**>junit</**artifactId**>  
 <**version**>4.12</**version**>  
 <**scope**>test</**scope**>  
 </**dependency**>

//това ни добавя driver за mysql конектора  
 <**dependency**>  
 <**groupId**>mysql</**groupId**>  
 <**artifactId**>mysql-connector-java</**artifactId**>  
 <**version**>8.0.25</**version**>  
 </**dependency**>  
  
 </**dependencies**>  
</**project**>



Можем да заредим dependency и чрез сайта mvnrepository.com/

<https://mvnrepository.com/artifact/mysql/mysql-connector-java>



### Create Database Connection

**package** orm;  
  
**import** java.sql.Connection;  
**import** java.sql.DriverManager;  
**import** java.sql.SQLException;  
**import** java.util.Properties;  
  
**public class** MyConnector {  
 **private static** Connection *connection*;  
 **private static final** String ***jdbcString*** = **"jdbc:mysql://localhost:3306/"**;

//Прилагане на Singleton  
 *//всеки път ще работим с комплекта, който сме създали* **private** MyConnector() {} *// конструкторът по подразборане да не може да се инициализира*  
  
 **public static void** createConnection(String user, String password, String dbName) **throws** SQLException {  
 Properties properties = **new** Properties();  
 properties.setProperty(**"user"**, user);  
 properties.setProperty(**"password"**, password);  
  
 *connection* = DriverManager.*getConnection*(***jdbcStrin*** + dbName, properties);  
 }  
  
 **public static** Connection getConnection() {

//Прилагане на Singleton  
 **if** (*connection* == **null**){  
 **new** MyConnector.createConnection();*//ала бала* }  
  
 **return** *connection*;  
 }  
}

### Create Database Context

**package** orm;  
  
**public interface** DBContext<E> {  
 **boolean** persist(E entity);  
  
 Iterable<E> find(Class<E> table);  
  
 Iterable<E> find(Class<E> table, String where);  
  
 E findFirst(Class<E> table);  
  
 E findFirst(Class<E> table, String where);  
}

### Create annotation interfaces

@Retention(RetentionPolicy.***RUNTIME***)  
@Target(ElementType.***FIELD***)  
**public** @**interface** Column {  
 String name();  
}

@Retention(RetentionPolicy.***RUNTIME***)  
**public** @**interface** Entity {  
 String name();  
}

@Retention(RetentionPolicy.***RUNTIME***)  
@Target(ElementType.***FIELD***)  
**public** @**interface** Id {  
  
}

### Create class User and class Address and annotate on with the annotations interfaces

**import** annotations.Column;  
**import** annotations.Entity;  
**import** annotations.Id;  
  
**import** java.time.LocalDate;  
  
@Entity(name = **"users"**) *//в базата данни таблица users***public class** User {  
 @Id  
 @Column(name = **"id"**)  
 **private long id**;  
  
 @Column(name = **"username"**) *//в базата данни колона username* **private** String **username**;  
  
 @Column(name = **"age"**) *//в базата данни колона age* **private int age**;  
  
 @Column(name = **"registration\_date"**) *//в базата данни колона registration\_date* **private** LocalDate **registrationDate**;  
  
 *//Add new field when adding column to the database - we change only in the User class* @Column(name = **"last\_logged\_in"**)  
 **private** LocalDate **lastLoggedIn**;

**public** User(String username, **int** age, LocalDate registrationDate) {  
 **this**.**username** = username;  
 **this**.**age** = age;  
 **this**.**registrationDate** = registrationDate;  
 **this**.**lastLoggedIn** = LocalDate.*now*();  
 }

getters and setters here

}

@Entity(name = **"addresses"**) *//в базата данни таблица address****es*****public class** Address {  
 @Id  
 @Column(name = **"id"**)  
 **private int id**;  
  
 @Column(name = **"street"**) *//в базата данни колона street* **private** String **street**;  
  
 @Column(name = **"street\_number"**) *//в базата данни колона street\_number* **private int streetNumber**;  
  
 @Column(name = **"city"**) *//в базата данни колона city* **private** String **city**;  
  
 @Column(name = **"postal\_code"**) *//в базата данни колона postal\_code* **private** String **postalCode**;  
  
 **public** Address() {  
 }

### CREATE ENTITY MANAGER

**package** orm;  
  
**import** annotations.Column;  
**import** annotations.Entity;  
**import** annotations.Id;  
  
**import** java.lang.reflect.Field;  
**import** java.lang.reflect.InvocationTargetException;  
**import** java.sql.Connection;  
**import** java.sql.PreparedStatement;  
**import** java.sql.ResultSet;  
**import** java.sql.SQLException;  
**import** java.time.LocalDate;  
**import** java.util.\*;  
**import** java.util.stream.Collectors;  
  
**public class** EntityManager<E> **implements** DBContext<E> {  
 **private** Connection **connection**;  
  
 **public** EntityManager(Connection connection) {  
 **this**.**connection** = connection;  
 }  
  
 *//Part 1* @Override  
 *//Persist Object in the Database - insert a new row or update an existing row* **public boolean** persist(E entity) **throws** IllegalAccessException, SQLException {  
*// ensureTable(); - може да си го разпишем* Field pk\_idColumn = getIdColumn(entity.getClass());  
 pk\_idColumn.setAccessible(**true**);  
 Object idValue = pk\_idColumn.get(entity); *//вземи стойността на полето id* **if** (idValue == **null** || (**long**) idValue <= 0) {  
 **return** doInsert(entity);  
 }  
  
 **return** doUpdate(entity, (**long**) idValue); *//cast-ни го на long* }

### //Insert a new row in the database

**private boolean** doInsert(E entity) **throws** SQLException, IllegalAccessException {  
 String tableName = getTableName(entity.getClass());  
 List<String> tableFields = getColumnsWithoutId(entity.getClass()); *//username, age, registration\_date* List<String> tableValues = getColumnsValuesWithoitId(entity);  
  
 String insertQuery = String.*format*(**"INSERT INTO %s (%s) VALUES (%s)"**, tableName,  
 String.*join*(**","**, tableFields),  
 String.*join*(**","**, tableValues));  
  
 **return connection**.prepareStatement(insertQuery).execute();  
 }

### //Update a specific row in the database

**private boolean** doUpdate(E entity, **long** idValue) **throws** IllegalAccessException, SQLException {  
 String tableName = getTableName(entity.getClass());  
 List<String> tableFields = getColumnsWithoutId(entity.getClass()); *//username, age, registration\_date* List<String> tableValues = getColumnsValuesWithoitId(entity);  
  
  
 List<String> setStatements = **new** ArrayList<>();  
 **for** (**int** i = 0; i < tableFields.size() - 1; i++) {  
 setStatements.add(tableFields.get(i) + **"="** + tableValues.get(i));  
 }  
  
 String updateQuery = String.*format*(**"UPDATE %s SET %s WHERE id=%d"**,  
 tableName,  
 String.*join*(**","**, setStatements),  
 idValue);  
  
  
 **return connection**.prepareStatement(updateQuery).execute();  
 }  
  
 **private** List<String> getColumnsWithoutId(Class<?> aClass) {  
 List<String> collect = Arrays.*stream*(aClass.getDeclaredFields())  
 .filter(f -> !f.isAnnotationPresent(Id.**class**))  
 .filter(f -> f.isAnnotationPresent(Column.**class**)) *//само ги филтрирай - тези полета, които са анотирани с Column анотация* .map(f -> f.getAnnotationsByType(Column.**class**)) *//след като са налични полетата, ги вземи* .map(a -> a[0].name()) *//Вземи името на полето - само веднъж имаме върху класа User анотация с Entity анотация, или анотация Entity се използва само на един клас за момента* .collect(Collectors.*toList*());  
  
 **return** collect;  
 }  
  
 **private** List<String> getColumnsValuesWithoitId(E entity) **throws** IllegalAccessException {  
 Class<?> aClass = entity.getClass();  
 List<Field> fields = Arrays.*stream*(aClass.getDeclaredFields())  
 .filter(f -> !f.isAnnotationPresent(Id.**class**))  
 .filter(f -> f.isAnnotationPresent(Column.**class**))  
 .collect(Collectors.*toList*());  
  
 List<String> values = **new** ArrayList<>();  
 **for** (Field field : fields) {  
 field.setAccessible(**true**);  
 Object o = field.get(entity);  
  
 **if** (o **instanceof** String || o **instanceof** LocalDate) {  
 values.add(**"'"** + o + **"'"**);  
 } **else** {  
 values.add(o.toString());  
 }  
 }  
  
 **return** values;  
 }  
  
 **private** Field getIdColumn(Class<?> clazz) {  
 Field[] declaredFields = clazz.getDeclaredFields();  
 **for** (Field declaredField : declaredFields) {  
 **boolean** annotationPresent = declaredField.isAnnotationPresent(Id.**class**);  
 **if** (annotationPresent) {  
 **return** declaredField;  
 }  
 }  
  
 **throw new** UnsupportedOperationException(**"Entity does not have primary key"**);  
 }  
  
 **private** String getTableName(Class<?> aClass) {  
 Entity[] annotationsByType = aClass.getAnnotationsByType(Entity.**class**);  
 **if** (annotationsByType.**length** == 0) {  
 **throw new** UnsupportedOperationException(**"Class must be Entity"**);  
 }  
  
 **return** annotationsByType[0].name();  
 }

### //Fetching Results – findFirst and Find

@Override  
**public** Iterable<E> find(Class<E> table) **throws** InvocationTargetException, SQLException, InstantiationException, IllegalAccessException, NoSuchMethodException {  
 **return** find(table, **null**);  
}  
  
@Override  
**public** Iterable<E> find(Class<E> table, String where) **throws** SQLException, NoSuchMethodException, IllegalAccessException, InvocationTargetException, InstantiationException {  
 **return** find(table, where, **null**);  
}  
  
@Override  
**public** Iterable<E> find(Class<E> table, String where, String... colsToDisplay) **throws** SQLException, NoSuchMethodException, IllegalAccessException, InvocationTargetException, InstantiationException {  
 String tableName = getTableName(table);  
  
 String **selectedCols**;  
 **if** (colsToDisplay == **null**) {  
 selectedCols = **"\*"**;  
 } **else** {  
 selectedCols = Arrays.*stream*(colsToDisplay).collect(Collectors.*joining*(**","**));  
 }  
  
 String selectQuery = String.*format*(**"SELECT %s FROM %s %s"**,  
 **selectedCols**,  
 tableName,  
 where != **null** ? **"WHERE "** + where : **""**);  
  
 PreparedStatement statement = **connection**.prepareStatement(selectQuery);  
 ResultSet resultSet = statement.executeQuery();  
  
 List<E> output = **new** ArrayList<>();  
 **while** (resultSet.next()) {  
 *//resultEntity - като го променим в метода fillEntity, той ще се промени и тук в метода findFirst* E resultEntity = table.getDeclaredConstructor().newInstance(); *//new instance of class the object/instance of which is table* fillEntity(table, resultSet, resultEntity);  
 output.add(resultEntity);  
 }  
  
 **return** output;  
}

@Override  
 **public** E findFirst(Class<E> table) **throws** InvocationTargetException, SQLException, InstantiationException, IllegalAccessException, NoSuchMethodException {  
 **return** findFirst(table, **null**);  
 }  
  
 @Override  
 **public** E findFirst(Class<E> table, String where) **throws** SQLException, NoSuchMethodException, IllegalAccessException, InvocationTargetException, InstantiationException {  
 String tableName = getTableName(table);  
  
 String selectQuery = String.*format*(**"SELECT \* FROM %s %s LIMIT 1"**, tableName,  
 where != **null** ? **"WHERE "** + where : **""**);  
  
 PreparedStatement statement = **connection**.prepareStatement(selectQuery);  
 ResultSet resultSet = statement.executeQuery();  
  
 resultSet.next();  
  
 *//resultEntity - като го променим в метода fillEntity, той ще се промени и тук в метода findFirst* E resultEntity = table.getDeclaredConstructor().newInstance(); *//new instance of class the object/instance of which is table* fillEntity(table, resultSet, resultEntity);  
  
 **return** resultEntity;  
 }  
  
 **private void** fillEntity(Class<E> table, ResultSet resultSet, E resultEntity) **throws** SQLException, IllegalAccessException {  
 Field[] declaredFields = table.getDeclaredFields();

#### //използване на getMetaData за селекция на колони за дисплейване

ResultSetMetaData metaData = **resultSet.getMetaData();**  
 **int** columnCount = **metaData.getColumnCount();**  
 List<String> sqlSelectColumns = **new** ArrayList<>();  
 **for** (**int** i = 1; i <= columnCount ; i++) {  
 String columnName = metaData.getColumnName(i);  
 sqlSelectColumns.add(columnName);  
 }  
  
 **for** (Field declaredField : declaredFields) {  
 String fieldNameClass = declaredField.getAnnotationsByType(Column.**class**)[0].name();  
 **if** (**sqlSelectColumns.contains(fieldNameClass))** {  
 declaredField.setAccessible(**true**);  
 fillField(declaredField, resultSet, resultEntity);  
 }  
 }  
}

*//from SQL type we convert to JAVA data type - for each field - обратното на getSQLType метода* **private void** fillField(Field declaredField, ResultSet resultSet, E resultEntity) **throws** SQLException,  
 IllegalAccessException {  
 Class<?> fieldType = declaredField.getType();  
 *//String fieldName = declaredField.getName(); //връща името на полета на изкуствената инстанция, която е взела имена на полетата от SQL базата/таблицата* String fieldName = declaredField.getAnnotationsByType(Column.**class**)[0].name(); *//чрез използване на анотация връща името на полето от изкуствената инстанция на класа* **if** (fieldType == Integer.**class** || fieldType == **int**.**class**) {  
 **int** value = resultSet.getInt(fieldName); *//a java.sql command  
  
 //resultEntity e инстанция на table  
 //declared field е едно от полетата на table  
 //fieldName e името на полето  
 //value е резултат от SQL заявка (java.sql)* declaredField.set(resultEntity, value);  
 } **else if** (fieldType == **long**.**class**) {  
 **long** value = resultSet.getLong(fieldName); *//a java.sql command* declaredField.set(resultEntity, value);  
 } **else if** (fieldType == LocalDate.**class**) {  
 LocalDate value = LocalDate.*parse*(resultSet.getString(fieldName));  
 declaredField.set(resultEntity, value);  
 } **else** {  
 String value = resultSet.getString(fieldName);  
 declaredField.set(resultEntity, value);  
 }  
 }

*//Part 2*

### //Create Table

**public void** doCreate(Class<E> entityClass) **throws** SQLException {  
 String tableName = getTableName(entityClass);  
 String fieldsWithTypes = getSQLFieldsWithTypes(entityClass);  
  
 String createQuery = String.*format*(**"CREATE TABLE %s ("** +  
 **"id INT PRIMARY KEY AUTO\_INCREMENT, %s)"**, tableName, fieldsWithTypes);  
  
 PreparedStatement statement = **connection**.prepareStatement(createQuery);  
 statement.execute();  
 }  
  
 **private** String getSQLFieldsWithTypes(Class<E> entityClass) {  
 **return** Arrays.*stream*(entityClass.getDeclaredFields())  
 .filter(f -> !f.isAnnotationPresent(Id.**class**)) *//без id-то на класа* .filter(f -> f.isAnnotationPresent(Column.**class**))  
 .map(field -> {  
 String fieldName = field.getAnnotationsByType(Column.**class**)[0].name(); *//тези полета на класа User, които са анотирани с Column анотация  
  
 //Определяме SQL типа на база JAVA типа* String sqlType = getSQLType(field.getType());  
  
 **return** fieldName + **" "** + sqlType;  
 })  
 .collect(Collectors.*joining*(**","**));  
 }  
  
 *//Определяме SQL типа на база JAVA типа - от Java към SQL тип* **private** String getSQLType(Class<?> type) {  
 String sqlType = **""**;  
 **if** (type == Integer.**class** || type == **int**.**class**) {  
 sqlType = **"INT"**;  
 } **else if** (type == String.**class**) {  
 sqlType = **"VARCHAR(200)"**;  
 } **else if** (type == LocalDate.**class**) {  
 sqlType = **"DATE"**;  
 }  
 **return** sqlType;  
 }

### //Alter Table – add new columns

**public void** doAlter(Class<E> entityClass) **throws** SQLException {  
 String tableName = getTableName(entityClass);  
 String addColumnStatements = getAddColumnStatementsForNewFields(entityClass);  
  
 String alterQuery = String.*format*(**"ALTER TABLE %s %s"**, tableName, addColumnStatements);  
  
 PreparedStatement statement = **connection**.prepareStatement(alterQuery);  
 statement.execute();  
 }  
  
 **private** String getAddColumnStatementsForNewFields(Class<E> entityClass) **throws** SQLException {  
 Set<String> sqlColumns = getSQLColumnNames(entityClass);  
  
 List<Field> fields = Arrays.*stream*(entityClass.getDeclaredFields())  
 .filter(f -> !f.isAnnotationPresent(Id.**class**)) *//без id-то на класа* .filter(f -> f.isAnnotationPresent(Column.**class**))  
 .collect(Collectors.*toList*());  
  
 List<String> allAddStatements = **new** ArrayList<>();  
 **for** (Field field : fields) {  
 String fieldName = field.getAnnotationsByType(Column.**class**)[0].name(); *//тези полета на класа User, които са анотирани с Column анотация* **if** (sqlColumns.contains(fieldName)) {  
 **continue**;  
 }  
  
 *//Определяме SQL типа на база JAVA типа - от Java към SQL тип* String sqlType = getSQLType(field.getType());  
  
 String addStatement = String.*format*(**"ADD COLUMN %s %s"**, fieldName, sqlType);  
 allAddStatements.add(addStatement);  
 }  
  
 **return** allAddStatements.stream().collect(Collectors.*joining*(**","**));  
 }  
  
 **private** Set<String> getSQLColumnNames(Class<E> entityClass) **throws** SQLException {

String tableName = getTableName(entityClass);  
  
*//вземи всички имена на колони без id-то - SQL заявка*String schemaQuery = String.*format*(**"SELECT `COLUMN\_NAME` FROM `information\_schema`.`columns`\n"** +  
 **"WHERE `TABLE\_SCHEMA` = 'custom-orm' AND `COLUMN\_NAME` != 'id'\n"** +  
 **"AND `TABLE\_NAME` = %s;"**, tableName);  
  
*/\*String schemaQuery = "SELECT `COLUMN\_NAME` FROM `information\_schema`.`columns`\n" +  
 "WHERE `TABLE\_SCHEMA` = 'custom-orm' AND `COLUMN\_NAME` != 'id'\n" +  
 "AND `TABLE\_NAME` = 'users';"; \*/*

PreparedStatement statement = **connection**.prepareStatement(schemaQuery);  
  
 ResultSet resultSet = statement.executeQuery();  
  
 Set<String> result = **new** HashSet<>();  
 **while** (resultSet.next()) {  
 String columnName = resultSet.getString(**"COLUMN\_NAME"**);  
 result.add(columnName);  
 }  
  
 **return** result;  
 }

### //Delete row

@Override  
 **public boolean** delete(E userToDelete) **throws** IllegalAccessException, SQLException {  
 String tableName = getTableName(userToDelete.getClass());  
 Field idColumn = getIdColumn(userToDelete.getClass());  
  
 String idColumnName = idColumn.getAnnotationsByType(Column.**class**)[0].name(); *//чрез използване на анотация връща името на полето от изкуствената инстанция на класа* idColumn.setAccessible(**true**);  
 Object idColumnValue = idColumn.get(userToDelete);  
  
 String queryDelete = String.*format*(**"DELETE FROM %s WHERE %s = %s"**,  
 tableName,  
 idColumnName,  
 idColumnValue);  
  
 **return connection**.prepareStatement(queryDelete).execute();  
 }  
  
}

### Test Insert a row

**public class** Main {  
 **public static void** main(String[] args) **throws** SQLException, IllegalAccessException {  
 MyConnector.*createConnection*(**"root"**, **""**, **"custom-orm"**);  
 Connection connection = MyConnector.*getConnection*();  
  
 EntityManager<User> userEntityManager = **new** EntityManager<>(connection);  
  
 User user = **new** User(**"pesho"**, 25, LocalDate.*now*());  
 userEntityManager.persist(user);

connection.close();

}  
}

### Test Update a row

**public class** Main {  
 **public static void** main(String[] args) **throws** SQLException, IllegalAccessException {  
 MyConnector.*createConnection*(**"root"**, **""**, **"custom-orm"**);  
 Connection connection = MyConnector.*getConnection*();  
  
 EntityManager<User> userEntityManager = **new** EntityManager<>(connection);  
  
 User user = **new** User(**"pesho"**, 25, LocalDate.*now*());  
 user.setId(2); **//user с id 2 му променяме**  
 user.setUsername(**"pesho\_new\_new"**); **//променяме името му**userEntityManager.persist(user);

connection.close();  
 }  
}

### Test Create new table

**public class** Main {  
 **public static void** main(String[] args) **throws** SQLException, IllegalAccessException {  
 MyConnector.*createConnection*(**"root"**, **""**, **"custom-orm"**);  
 Connection connection = MyConnector.*getConnection*();  
  
 EntityManager<User> userEntityManager = **new** EntityManager<>(connection);  
  
 User user = **new** User(**"pesho"**, 25, LocalDate.*now*());  
 userEntityManager.doCreate(User.**class**);  
 userEntityManager.persist(user);

connection.close();  
 }  
}

### Test Alter existing table – add new columns

**public class** Main {  
 **public static void** main(String[] args) **throws** SQLException, IllegalAccessException {  
 MyConnector.*createConnection*(**"root"**, **""**, **"custom-orm"**);  
 Connection connection = MyConnector.*getConnection*();  
  
 EntityManager<User> userEntityManager = **new** EntityManager<>(connection);  
  
 User user = **new** User(**"pesho"**, 25, LocalDate.*now*());  
 userEntityManager.doAlter(User.**class**);  
 userEntityManager.persist(user); //изпълнява се задължително

connection.close();  
 }  
}

*//Add new field when adding column to the database - we change only in the* ***User class***@Column(name = **"last\_logged\_in"**)  
 **private** LocalDate **lastLoggedIn**;

### Test Find First

**public class** Main {  
 **public static void** main(String[] args) **throws** SQLException, IllegalAccessException, NoSuchMethodException, InstantiationException, InvocationTargetException {  
 MyConnector.*createConnection*(**"root"**, **""**, **"custom-orm"**);  
 Connection connection = MyConnector.*getConnection*();  
  
 EntityManager<User> userEntityManager = **new** EntityManager<>(connection);  
  
 User user = **new** User(**"pesho"**, 25, LocalDate.*now*());  
 User first = userEntityManager.findFirst(User.**class**, **"id = 2"**);  
 System.***out***.println(first);

connection.close();  
}  
}

### Test Find

**public class** Main {  
 **public static void** main(String[] args) **throws** SQLException, IllegalAccessException, NoSuchMethodException, InstantiationException, InvocationTargetException {  
 MyConnector.*createConnection*(**"root"**, **""**, **"custom-orm"**);  
 Connection connection = MyConnector.*getConnection*();  
  
 EntityManager<User> userEntityManager = **new** EntityManager<>(connection);  
  
 User user = **new** User(**"Svilen"**, 36, LocalDate.*now*());  
Iterable<User> first = userEntityManager.find(User.**class**, **"id<5"**);

Iterable<User> first = userEntityManager.find(User.**class**, **"id<5",** );  
 System.***out***.println(first);

Iterable<User> testDisplayingSpecificColumns = userEntityManager.find(User.**class**, **"age>27"**,  
 **"age"**, **"registration\_date"**);  
System.***out***.println(testDisplayingSpecificColumns);

[User{id=0, username='null', **age=28, registrationDate=2022-02-28**, lastLoggedIn=null}, User{id=0, username='null', **age=36, registrationDate=2022-02-23**, lastLoggedIn=null}]

connection.close();}  
}

### Test Delete – a specific row from the table

**public class** Main {  
 **public static void** main(String[] args) **throws** SQLException, IllegalAccessException, NoSuchMethodException, InstantiationException, InvocationTargetException {  
 MyConnector.*createConnection*(**"root"**, **""**, **"custom-orm"**);  
 Connection connection = MyConnector.*getConnection*();  
  
 EntityManager<User> userEntityManager = **new** EntityManager<>(connection);  
  
 User user = **new** User(**"Ala bala"**, 36, LocalDate.*now*());  
User userToDelete = userEntityManager.findFirst(User.**class**, **"id=6"**);  
 System.***out***.println(userToDelete);  
 userEntityManager.delete(userToDelete);

connection.close();  
}  
}

# 3. Hibernate Introduction

## 3.0. Особености

И при HQL и при JPQL, работим с имената на класовете и с имената на полета на тези класове, вместо с името на таблицата от базата данни и имената на колоните от таблиците от базата данни.

Java типовете стоят в QL заявката!!!

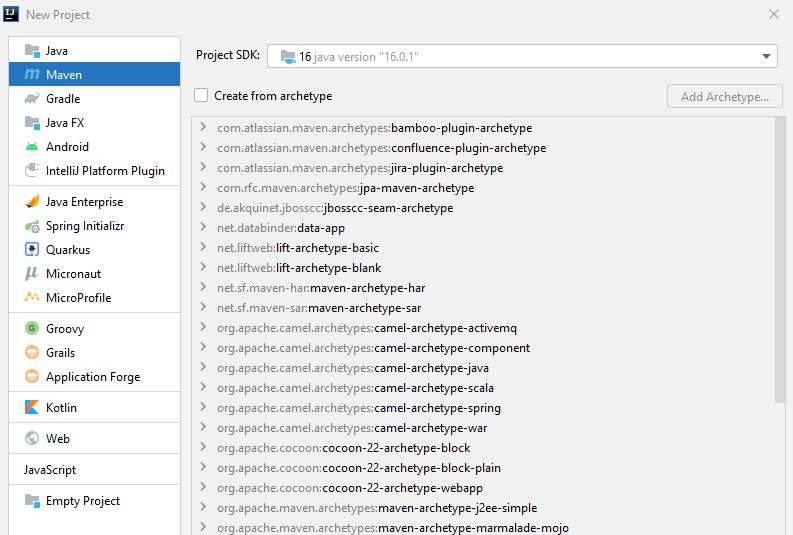
## 3.1. Maven - Project Management and Comprehension

### Maven Overview

* Maven is a built automation tool.
  + Describes how software is built and its dependencies
  + Uses XML files
* Dynamically downloads **Java libraries** and **Maven plug-ins**
  + Projects are configured using a **P**roject **O**bject **M**odel, which is stored in a **pom.xml** file

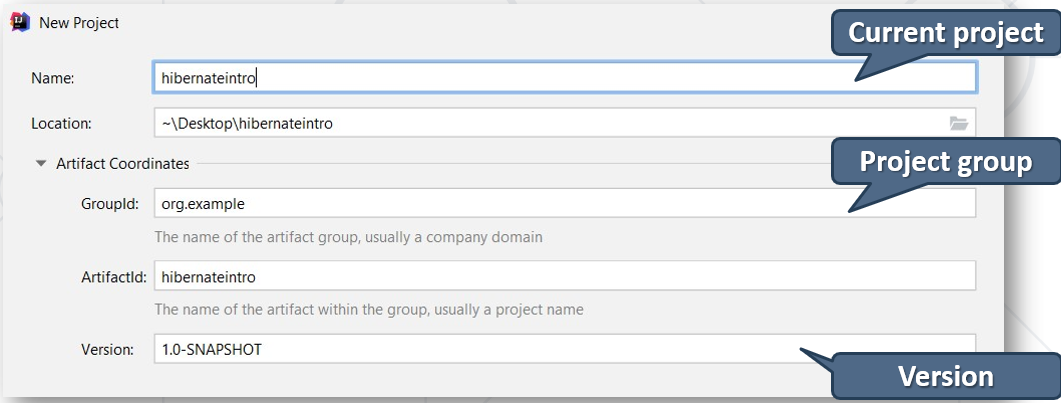
### Setup – Creating a Maven Project

* Select "Maven" project from the new project panel:



**GroupId** – името на организацията, в която ранотим

**ArtefactId** – ако имаме различни проекти в нашата организация, то името на конкретния проект/библиотека



### Maven Configurations

* A **P**roject **O**bject **M**odel(**POM**) is the fundamental unit of work in Maven
* **Configurations** are held in the **pom.xml** file
* When executing a task or goal, Maven looks for the POM file in the current directory

### pom.xml файла с есктри

<**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 http://maven.apache.org/xsd/maven-4.0.0.xsd"**>  
 <**modelVersion**>4.0.0</**modelVersion**>  
  
 <**groupId**>org-example</**groupId**>  
 <**artifactId**>custom-orm</**artifactId**>  
 <**version**>1.0-SNAPSHOT</**version**>

<**packaging**>jar</**packaging**> //ако искаме да deploy-ваме

**Properties обобщават долуизброените, и ако се наложи да се променя на много места, то можем да направим грешка. А тук само на едно място променяме версията**  
<**properties**>  
 <**maven.compiler.source**> 14 </**maven.compiler.source**>  
 <**maven.compiler.target**> 14 </**maven.compiler.target**>  
 <**hibernate.version**>5.4.30.Final</**hibernate.version**>  
 <**msql.version**>8.0.25</**msql.version**>  
 <**jaxb.version**>2.2.11</**jaxb.version**> //три dependencies или повече могат да използват това  
</**properties**>

<**build**>  
 <**plugins**>  
 <**plugin**>  
 <**groupId**>org.apache.maven.plugins</**groupId**>  
 <**artifactId**>maven-compiler-plugin</**artifactId**>  
 <**version**>3.6.1</**version**> //слагаме и version number   
 <**configuration**>  
 <**source**>14</**source**>  
 <**target**>14</**target**>  
 <**annotationProcessorPaths**> //използване на lombok  
 <**path**>  
 <**groupId**>org.projectlombok</**groupId**>  
 <**artifactId**>lombok</**artifactId**>  
 <**version**>1.18.20</**version**>  
 </**path**>  
 </**annotationProcessorPaths**>  
 </**configuration**>  
 </**plugin**>  
 </**plugins**>  
</**build**>

<**dependencies**>

#### //това ни добавя hibernate-core

<**dependency**>  
 <**groupId**>org.hibernate</**groupId**>  
 <**artifactId**>hibernate-core</**artifactId**>  
 <**version**>${hibernate.version}</**version**>  
</**dependency**>

#### //това ни добавя driver за mysql конектора

<**dependency**>  
 <**groupId**>mysql</**groupId**>  
 <**artifactId**>mysql-connector-java</**artifactId**>  
 <**version**>${msql.version}</**version**>  
</**dependency**>

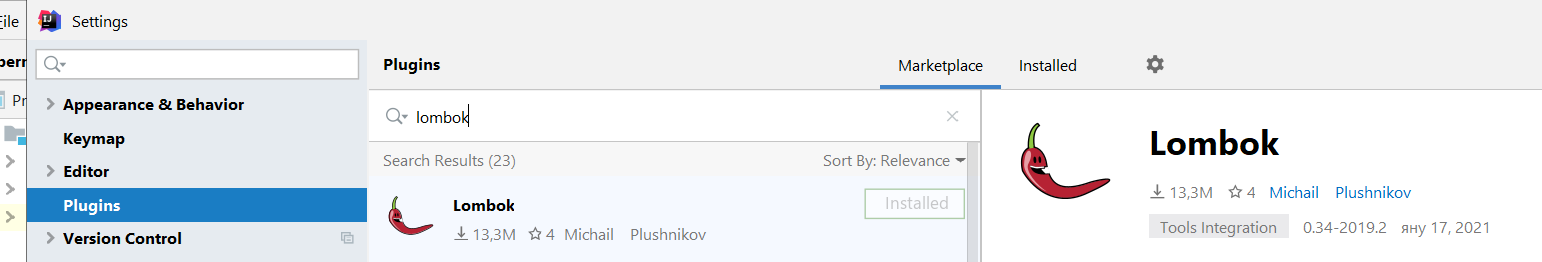
Как реферираме version от Properties

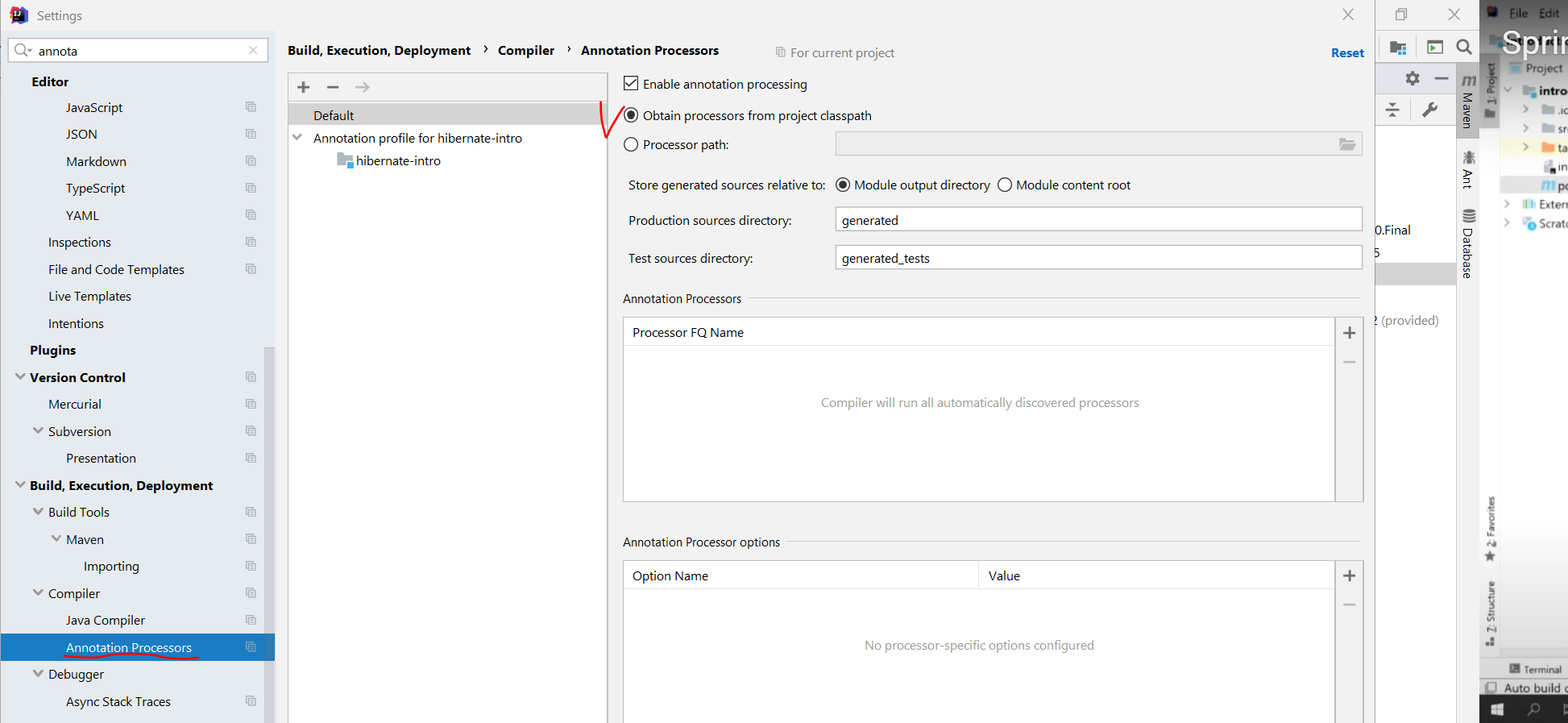
<**dependency**>  
 <**groupId**>com.sun.xml.bind</**groupId**>  
 <**artifactId**>jaxb-api</**artifactId**>  
 <**version**>${jaxb.version}</**version**>  
</**dependency**>  
  
<**dependency**>  
 <**groupId**>com.sun.xml.bind</**groupId**>  
 <**artifactId**>jaxb-core</**artifactId**>  
 <**version**>${jaxb.version}</**version**>  
</**dependency**>  
  
<**dependency**>  
 <**groupId**>com.sun.xml.bind</**groupId**>  
 <**artifactId**>jaxb-impl</**artifactId**>  
 <**version**>${jaxb.version}</**version**>  
</**dependency**>

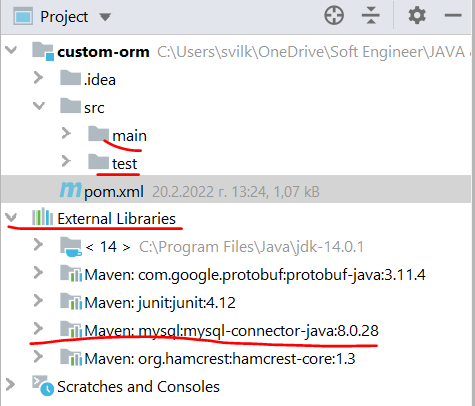
**//Използване на lombok**

//генерира необходимите getters, setters, hashCode и т.н.

<**dependency**>  
 <**groupId**>org.projectlombok</**groupId**>  
 <**artifactId**>lombok</**artifactId**>  
 <**version**>1.18.12</**version**>  
 <**scope**>provided</**scope**> //няма смисъл да отива в jar файла. Служи само за компилация  
</**dependency**>

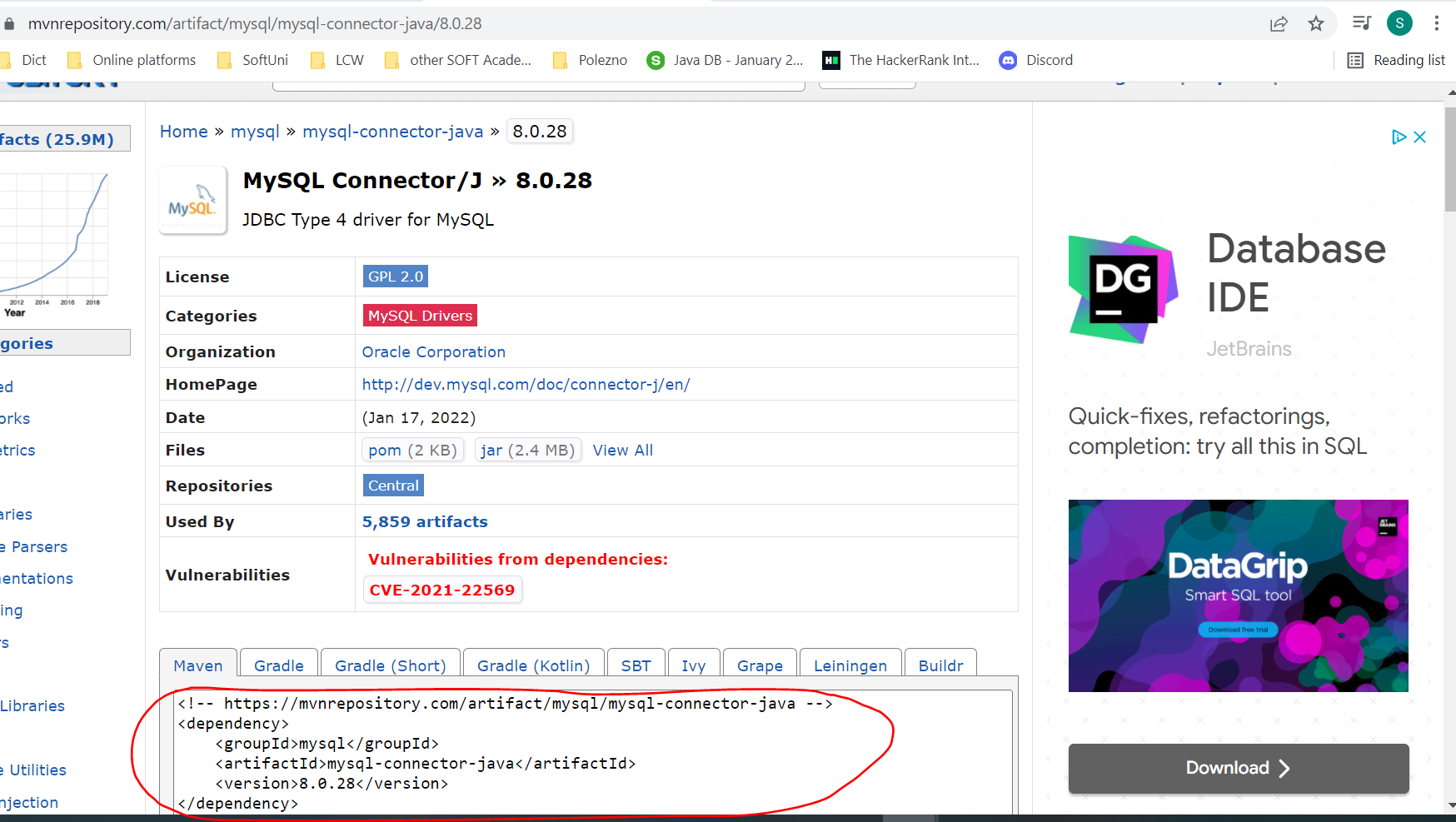






Можем да заредим dependency и чрез сайта mvnrepository.com/

<https://mvnrepository.com/artifact/mysql/mysql-connector-java>

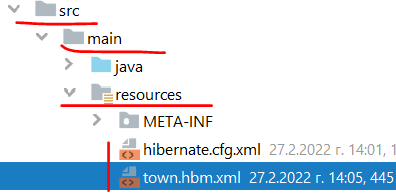


## 3.2. Java ORM Approaches

* Different approaches to **Java** **ORM**:
  + **1) POJO (Plain Old Java Objects Without annotations) + XML mappings**
    - A bit old-fashioned, but very powerful
    - Implemented in the "classical" Hibernate without JPA
  + **2) Annotated Java classes (POJO) mapped to DB tables – имаме кое какво мапва на едно място, пример с класа User от custom-orm** 
    - Based on Java annotations and XML
    - Easier to implement and maintain
    - Implemented in **both Hibernate classical and Hibernate with JPA frameworks**
    - Code generation - tools

### 1) Classical Hibernate XML mappings using POJO (Plain Old Java Objects Without annotations)

#### student.hbm.xml – create it in **src/main/resources**



*<?***xml version="1.0" encoding="utf-8"***?>***<!DOCTYPE hibernate-mapping PUBLIC** //Mapping file **"-//Hibernate/Hibernate Mapping DTD//EN"  
 "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd"*>***

//тук мапваме различните класове към базата данни<**hibernate-mapping**>   
 <**class name="entities.Student" table="students"**> //Class mapping – име на клас, след това име на таблица от базата

<**meta attribute="class-description"**>  
 This class contains the student details  
</**meta**>

<**id name="id" column="id"**> //Field mapping – име на поле от класа, след това име на колона на таблицата от базата

//показва, че е auto-generator/auto-generated

<**generator class="identity"**></**generator**> //когато няма body в тага, то можем да използваме съкратен запис както следва:

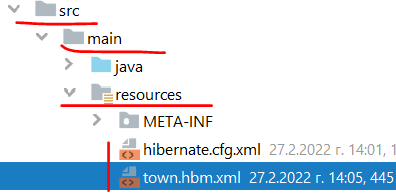
<**generator class="identity"** />  
 </**id**>

<**property name="name" column="first\_name"** /> //Field mapping – име на поле от класа, след това име на колона на таблицата от базата

<**property name="registrationDate" column="registration\_date" type="timestamp"**/> //Field mapping – име на поле от класа, след това име на колона на таблицата от базата

</**class**>  
</**hibernate-mapping**>

#### hibernate.cfg.xm – **create it in src/main/resources**



*<?***xml version='1.0' encoding='utf-8'***?>***<!DOCTYPE hibernate-configuration** //Configuration **PUBLIC "-//Hibernate/Hibernate Configuration DTD//EN"  
 "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd"*>***<**hibernate-configuration**>  
 <**session-factory**>  
 <**property name="hibernate.dialect"**>  
 org.hibernate.dialect.MySQL8Dialect //SQL dialect  
 </**property**>

<**property name="hibernate.connection.driver\_class"**>  
 com.mysql.cj.jdbc.Driver //driver  
 </**property**>

*<!-- Connection Settings -->* <**property name="hibernate.connection.url"**> jdbc:mysql://localhost:3306/school?createDatabaseIfNotExist=true //connection string  
 </**property**>

<**property name="hibernate.connection.username"**>  
 root //user  
 </**property**>

<**property name="hibernate.connection.password"**>  
 “” //pass  
 </**property**>

<**property name="hbm2ddl.auto"**>  
 update //auto-strategy  
 </**property**>

<**property name="show\_sql"**>true</**property**> //за показване на конзолата самата заявка  
<**property name="format\_sql"**>true</**property**>  
<**property name="use\_sql\_comments"**>true</**property**>

*<!-- List of XML mapping files -->* <**mapping resource="student.hbm.xml"**/> //mapping files  
 </**session-factory**>  
</**hibernate-configuration**>

#### pom.xml

Native Hibernate e hibernate-core

**<build>……..**

<**dependencies**>  
 <**dependency**>  
 <**groupId**>org.hibernate</**groupId**>  
 <**artifactId**>hibernate-core</**artifactId**>  
 <**version**>5.4.22.Final</**version**>  
 </**dependency**>  
  
 <**dependency**>  
 <**groupId**>mysql</**groupId**>  
 <**artifactId**>mysql-connector-java</**artifactId**>  
 <**version**>8.0.25</**version**>  
 </**dependency**>  
 </**dependencies**>  
  
</**project**>

#### NativeHibernateMain

**import org.hibernate**.Session;  
**import org.hibernate**.SessionFactory;  
**import org.hibernate**.cfg.Configuration;  
  
**public class** NativeHibernateMain {  
 **public static void** main(String[] args) {  
 Configuration cfg = **new** Configuration();  
 cfg.configure(); *//в скобите можем да сложим от кой файл искаме да го заредим* SessionFactory sessionFactory = cfg.buildSessionFactory();  
  
 Session session = sessionFactory.openSession();  
 session.beginTransaction();  
  
 Town town = session.get(Town.**class**, 1);  
  
 System.***out***.println(town);  
  
 session.getTransaction().commit();  
 session.close();  
 }  
}

### 2) Annotated Java classes (annotated POJO) mapped to DB tables

#### Entity Class: Student

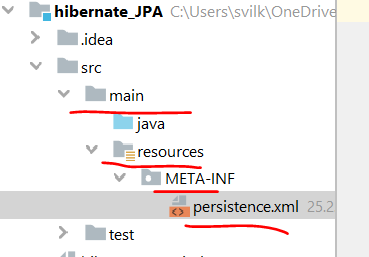
@Entity

@Table(name = **"students"**) //set the class with name Student to be in the database as table students, подобно на xml mapping файла  
**public class** Student {  
 @Id //primary key  
 @GeneratedValue(strategy = GenerationType.IDENTITY) //identity  
 @Column(name = **"id"**) //column name  
 **private long id**;

@Column(name = **"name"**, length = 50) //column name and length  
 **private** String **name**;  
  
 *// Getters and setters //тук си създаваме конструктори/getters/setters ръчно*}

#### persistence.xml – create it in **src/main/resources/META-INF**

META-INF е стандартно име на папка в Java, в която папка биха стояли конфигурации



*<?***xml version="1.0" encoding="UTF-8"***?>*<**persistence xmlns="http://java.sun.com/xml/ns/persistence" version="2.0"**>  
 <**persistence-unit name="school"**> //persistence unit name  
 <**properties**>  
 <**property name="hibernate.connection.url"  
 value="jdbc:mysql://localhost:3306/school?createDatabaseIfNotExist=true"**/>  
 <**property name="hibernate.connection.driver\_class" value="com.mysql.cj.jdbc.Driver"**/>  
 <**property name="hibernate.connection.username" value="root"**/>  
 <**property name="hibernate.connection.password" value=""**/>  
 <**property name="hibernate.dialect" value="org.hibernate.dialect.MySQL8Dialect"**/>

<**property name="hibernate.hbm2ddl.auto" value="update"**/>

<**property name="hibernate.hbm2ddl.auto" value="create-drop"**/> //изтрий таблицата и я пресъздай наново с новите/верните/променените колони

<**property name="hibernate.hbm2ddl.auto" value="validate"**/> //за валидиране на миграции на данни

<**property name="hibernate.show\_sql" value="true"**/>

</**properties**>  
 </**persistence-unit**>  
</**persistence**>

#### pom.xml

JPA е javax.persistence-api

**<build>……..**

<**dependencies**>  
 <**dependency**>  
 <**groupId**>javax.persistence</**groupId**>  
 <**artifactId**>javax.persistence-api</**artifactId**>  
 <**version**>2.2</**version**>  
 </**dependency**>  
  
 <**dependency**>  
 <**groupId**>mysql</**groupId**>  
 <**artifactId**>mysql-connector-java</**artifactId**>  
 <**version**>8.0.25</**version**>  
 </**dependency**>  
 </**dependencies**>  
</**project**>

#### JpaHibernateMain

**import javax.persistence.**EntityManager;  
**import javax.persistence.**EntityManagerFactory;  
**import javax.persistence.**Persistence;  
  
**public class** JpaHibernateMain {  
 **public static void** main(String[] args) {  
 EntityManagerFactory emf = Persistence.*createEntityManagerFactory*(**"soft\_uni"**);  
 EntityManager entityManager = emf.createEntityManager();  
  
 entityManager.getTransaction().begin();  
 Town town = entityManager.find(Town.**class**, 1);  
  
 System.***out***.println(town);  
  
 entityManager.getTransaction().commit();  
 entityManager.close();  
 }  
}

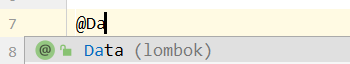
## 3.3. Hibernate Classical Framework (before JPA) - Classical Hibernate XML mappings using POJO

### Hibernate Framework

* Hibernate is a Java ORM framework
  + Mapping an object-oriented model to a relational database
    - It is implemented by the configuration of an **XML** **file** or by using **Java** **Annotations**
  + Maintain the database schema

### Използване на Lombok

**package** demos.hibernate.model;  
  
**import** lombok.Data;  
  
**import** java.util.Date;

  
@Data //пишем/избираме @Data анотация и не се налага да пишем след това getters, setters, etc за всеки елемент/поле/метод на класа – яко 😊

Но самите полета на класа не са анотирани в смисъл анотирани към базата данни, както е в следващата точка 3.4.  
@NoArgsConstructor //празен конструктор да има  
@RequiredArgsConstructor  
@AllArgsConstructor  
**public class** Student {  
 **private int id**;  
  
 @NonNull *//птрави полето name като required argument на конструктора* **private** String **name**;  
   
 **private** Date **registrationDate** = **new** Date();  
}

### Hibernate Configuration

#### pom.xml

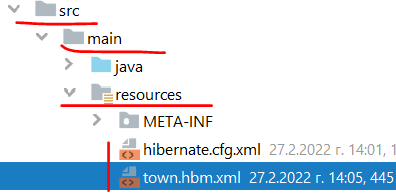
##### //това ни добавя hibernate-core

<**dependency**>  
 <**groupId**>org.hibernate</**groupId**>  
 <**artifactId**>hibernate-core</**artifactId**>  
 <**version**>5.4.30.Final</**version**>  
</**dependency**>

##### //това ни добавя driver за mysql конектора

<**dependency**>  
 <**groupId**>mysql</**groupId**>  
 <**artifactId**>mysql-connector-java</**artifactId**>  
 <**version**>8.0.25</**version**>  
 </**dependency**>

#### hibernate.cfg.xm – **create it in src/main/resources**



*<?***xml version='1.0' encoding='utf-8'***?>***<!DOCTYPE hibernate-configuration** //Configuration **PUBLIC "-//Hibernate/Hibernate Configuration DTD//EN"  
 "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd"*>***<**hibernate-configuration**>  
 <**session-factory**>  
 <**property name="hibernate.dialect"**>  
 org.hibernate.dialect.MySQL8Dialect //SQL dialect  
 </**property**>

<**property name="hibernate.connection.driver\_class"**>  
 com.mysql.cj.jdbc.Driver //driver  
 </**property**>

*<!-- Connection Settings -->* <**property name="hibernate.connection.url"**> jdbc:mysql://localhost:3306/school?createDatabaseIfNotExist=true //connection string  
 </**property**>

<**property name="hibernate.connection.username"**>  
 root //user  
 </**property**>

<**property name="hibernate.connection.password"**>  
 “” //pass  
 </**property**>

<**property name="hbm2ddl.auto"**>  
 update //auto-strategy  
 </**property**>

<**property name="show\_sql"**>true</**property**> //за показване на конзолата самата заявка  
<**property name="format\_sql"**>true</**property**>  
<**property name="use\_sql\_comments"**>true</**property**>

*<!-- List of XML mapping files -->* <**mapping resource="student.hbm.xml"**/> //mapping files  
 </**session-factory**>  
</**hibernate-configuration**>

#### student.hbm.xml – create it in **src/main/resources**

*<?***xml version="1.0" encoding="utf-8"***?>***<!DOCTYPE hibernate-mapping PUBLIC** //Mapping file **"-//Hibernate/Hibernate Mapping DTD//EN"  
 "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd"*>***

//тук мапваме различните класове към базата данни<**hibernate-mapping**>   
 <**class name="entities.Student" table="students"**> //Class mapping – име на клас, след това име на таблица от базата

<**meta attribute="class-description"**>  
 This class contains the student details  
</**meta**>

<**id name="id" column="id"**> //Field mapping – име на поле от класа, след това име на колона на таблицата от базата

//показва, че е auto-generator/auto-generated

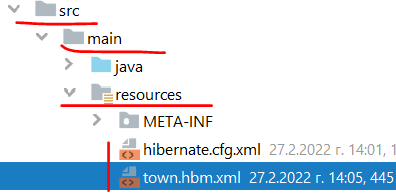
<**generator class="identity"**></**generator**> //когато няма body в тага, то можем да използваме съкратен запис както следва:

<**generator class="identity"** />  
 </**id**>

<**property name="name" column="first\_name"** /> //Field mapping – име на поле от класа, след това име на колона на таблицата от базата

<**property name="registrationDate" column="registration\_date" type="timestamp"**/> //Field mapping – име на поле от класа, след това име на колона на таблицата от базата

</**class**>  
</**hibernate-mapping**>



### Hibernate Sessions

**import** org.hibernate.Session;  
**import** org.hibernate.SessionFactory;  
**import** org.hibernate.cfg.Configuration;  
  
**public class** HibernateDenoWithXMLNoAnnotationsMain {  
 **public static void** main(String[] args) {  
 Configuration cfg = **new** Configuration(); *//Service registry* cfg.configure(); *//в скобите можем да сложим от кой файл искаме да го заредим* SessionFactory sessionFactory = cfg.buildSessionFactory();  
  
 Session session = sessionFactory.openSession(); *//session* session.beginTransaction();  
  
 *// Your Code Here* session.getTransaction().commit(); *//transaction commit* session.close();  
 }  
}

### Hibernate Save Data

Има няколко начина – чрез **persist** или **merge** (част от JPA),

или чрез **строго специфичните за Hibernate неизискуеми от JPA** методи **save** и **update** и **saveorupdate**

#### Пример 1:

**public static void** main(String[] args) {  
 *//…* session.beginTransaction();  
  
 Student example = **new** Student();  
 session.**save**(example); //Save object  
  
 session.getTransaction().commit();  
 session.close();  
}

#### Пример 2

**import** org.hibernate.FlushMode;  
**import** org.hibernate.Session;  
**import** org.hibernate.SessionFactory;  
**import** org.hibernate.cfg.Configuration;  
  
**public class** HibernateDemoWithXMLNoAnnotationsMain {  
 **public static void** main(String[] args) {  
 Configuration config = **new** Configuration(); *//Service registry  
// config.configure("src/java/hibernate.cfg.xml"); //в скобите можем да сложим от кой файл искаме да го заредим* config.configure(); *//в скобите можем да сложим от кой файл искаме да го заредим  
  
 //try with resources / try-with-resources* **try** (SessionFactory sessionFactory = config.buildSessionFactory();  
 Session session = sessionFactory.openSession()) {  
  
 Student student = **new** Student(**"Ivan Petrov"**);  
 **try** {  
 session.**save**(student);  
 //session.persist(student);  
 //session.detach(student);  
 } **catch** (Exception e) {  
 **if** (session.getTransaction() != **null**) {  
 session.getTransaction().rollback();  
 }  
 **throw** e;  
 }  
  
 session.beginTransaction();  
 session.setHibernateFlushMode(FlushMode.***MANUAL***);  
 session.getTransaction().commit(); *//transaction commit* session.close();  
 }  
 }  
}

### Hibernate Retrieve Data by Get

**public static void** main(String[] args) {  
…  
  
 Student student = session.**get**(Student.**class**, 1L); //Get object  
  
 session.close();  
 }

### Hibernate Retrieve Data by Query

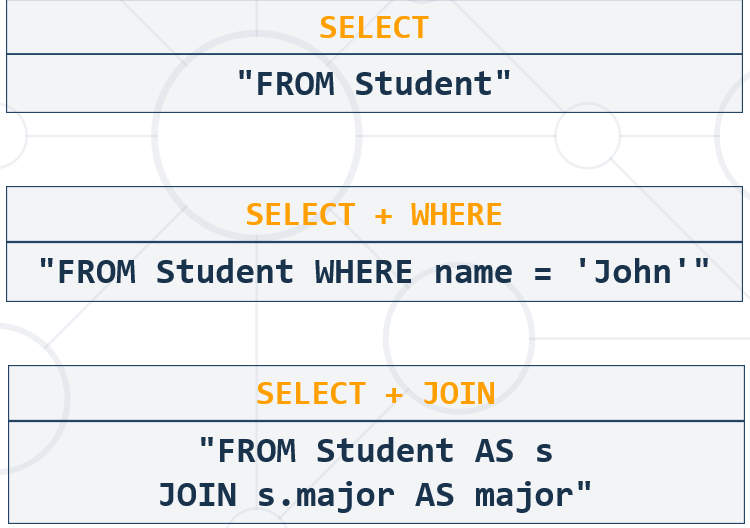
**public static void** main(String[] args) {  
 *// …* session.beginTransaction();  
  
 List<Student> studentList =  
 session.**createQuery**(**"FROM Student "**, Student.**class**).**list();**

**for** (Student student : studentList) {  
 System.***out***.println(student.**getId**());  
 }

session.getTransaction().commit();  
 session.close();  
}

### Hibernate Querying Language – HQL

Използваме имената на класовете и на техните полета вместо таблицата в базата данни с нейните имена колони



### Hibernate Retrieve Data by Criteria

**public static void** main(String[] args) {  
 *//…* session.beginTransaction();  
 **CriteriaBuilder** builder = session.getCriteriaBuilder(); **//by CriteriaBuilder it is type-safe**  
 **CriteriaQuery** criteria = builder.**createQuery()**;  
 **Root**<Student> r = criteria.**from**(Student.**class**);  
 criteria.**select**(r).**where**(builder.like(r.get(**"name"**), **"P%"**));

List<Student> studentList = session.createQuery(criteria).**getResultList();** //Get list of objects by criteria

**for** (Student student : studentList) {  
 System.***out***.println(student.getName());  
 }  
  
 session.getTransaction().commit();  
 session.close();  
}

## 3.4. JPA framework – Annotated Java classes (annotated POJO) mapped to DB tables

### About JPA – все едно огромно количество interface да се поддържат

* What is **Java Persistence API** (JPA)? – спецификация в Java казваща какво трябва да има един ORM, за да се класифицира като Java ORM
  + Database persistence technology for Java (**official** **standard**) – specific requirements/interfaces must be implemented so that we call it ORM engine acc. to Java official standard
    - Object-relational mapping (ORM) technology
    - Operates with POJO entities with annotations or XML mappings
    - Implemented by many **ORM** **engines**: **Hibernate**, **EclipseLink**, etc.
* JPA maps Java classes to database tables
  + Maps relationships between tables as associations between classes
* Provides **CRUD** functionality and queries
  + Create, read, update, delete + queries

### Entities in JPA

* A JPA entityis just a POJO class
  + Abstract or concrete **top level** Java class
  + Non-final fields/properties, **no-arguments constructor (задължително да има празен конструктор)**
  + No required interfaces
  + Direct field or property-based access
* Getter/setter can contain logic (e.g., validation)

### Entity Class: Student

@Entity

@Table(name = **"students"**) //set the class with name Student to be in the database as table students, подобно на xml mapping файла  
**public class** Student {  
 @Id //primary key  
 @GeneratedValue(strategy = GenerationType.IDENTITY) //identity  
 @Column(name = **"id"**) //column name  
 **private long id**;

@Column(name = **"name"**, length = 50) //column name and length  
 **private** String **name**;  
  
 *// Getters and setters //тук си създаваме конструктори/getters/setters ръчно*}

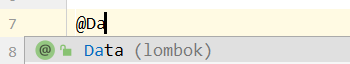
### Entity Class: Employee

**import** javax.persistence.\*;  
**import** java.math.BigDecimal;  
**import** java.time.LocalDateTime;  
**import** java.util.Set;

**//прави впечатление, че няма анотация на private полетата, и само на getters има, а на setters няма, и пак работи**  
@Entity  
@Table(name = **"employees"**)  
**public class** Employee {  
 **private** Integer **id**;  
 **private** String **firstName**;  
 **private** String **lastName**;  
 **private** String **middleName**;  
 **private** String **jobTitle**;  
 **private** Department **department**;  
 **private** Employee **manager**;  
 **private** LocalDateTime **hireDate**;  
 **private** BigDecimal **salary**;  
 **private** Address **address**;  
 **private** Set<Project> **projects**;  
  
  
 @Id  
 @GeneratedValue(strategy = GenerationType.***IDENTITY***)  
 @Column(name = **"employee\_id"**)  
 **public** Integer getId() {  
 **return id**;  
 }  
  
 **public void** setId(Integer id) {  
 **this**.**id** = id;  
 }  
  
 @Column(name = **"first\_name"**)  
 **public** String getFirstName() {  
 **return firstName**;  
 }  
  
 **public void** setFirstName(String firstName) {  
 **this**.**firstName** = firstName;  
 }  
  
 @Column(name = **"last\_name"**)  
 **public** String getLastName() {  
 **return lastName**;  
 }  
  
 **public void** setLastName(String lastName) {  
 **this**.**lastName** = lastName;  
 }  
  
 @Column(name = **"middle\_name"**)  
 **public** String getMiddleName() {  
 **return middleName**;  
 }  
  
 **public void** setMiddleName(String middleName) {  
 **this**.**middleName** = middleName;  
 }  
  
 @Column(name = **"job\_title"**)  
 **public** String getJobTitle() {  
 **return jobTitle**;  
 }  
  
 **public void** setJobTitle(String jobTitle) {  
 **this**.**jobTitle** = jobTitle;  
 }

### Entity Class: Student – вариант с lombok

**package** demos.hibernate.model;  
  
**import** lombok.Data;  
  
**import** java.util.Date;

  
@Data //пишем/избираме @Data анотация и не се налага да пишем след това getters, setters, etc за всеки елемент/поле/метод на класа – яко 😊

Но самите полета на класа не са анотирани в смисъл анотирани към базата данни, както е в следващата точка 3.4.  
@NoArgsConstructor //празен конструктор да има  
@RequiredArgsConstructor  
@AllArgsConstructor

@Entity

@Table(name = **"students"**)  
**public class** Student {

@Id //primary key  
 @GeneratedValue(strategy = GenerationType.IDENTITY) //identity  
 @Column(name = **"id"**) //column name  
 **private int id**;  
  
 @Column(name = **"name"**, length = 50) //column name and length**private** String **name**;

@Column(name = **"birth\_date"**)

@Temporal(TemporalType.**TIMESTAMP**)  
 **private** Date **birthDate** = **new** Date();

//тук благодарение на Lombok, са ни генерирани *конструктори/getters/setters авитоматично*  
}

### Annotations

* **@Entity** - Declares the class as an entity or a table
* **@Table** - Declares table name
* **@Basic** - Specifies non-constraint fields explicitly
* **@Transient** - Specifies the property that is not persistent, i.e., the value is never stored in the database – да не се добавя в базата данни

**За Id полето**

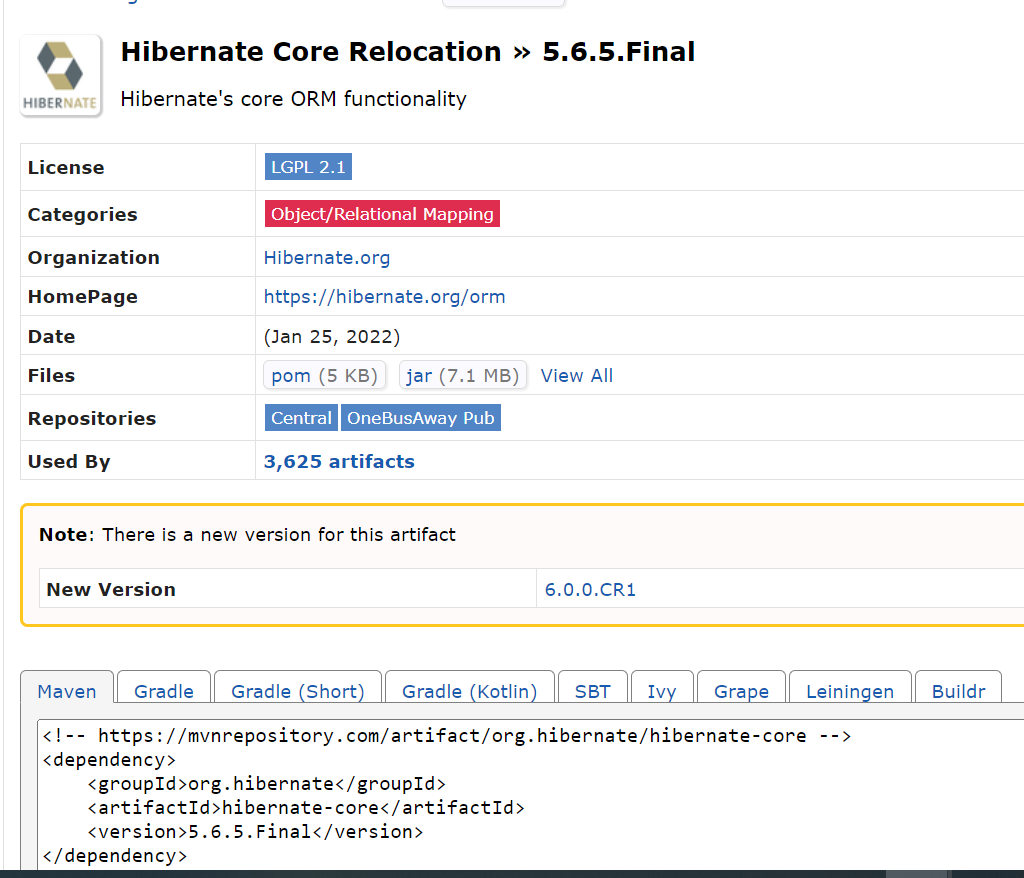
* **@Id** - Specifies the property, use for identity (primary key of a table) of the class
  + **@GeneratedValue** - specifies how the identity attribute can be initialized
    - Automatic, manual, or value taken from a sequence table
* **@Column** -Specifies the column attribute for the persistence property – ако няма да го достъпваме/оказваме, то можем да пропуснем @Column анотацията. Но за да е видимо в базата данни, му слагаме все пак тази анотация

### JPA Configuration in Maven, in the pom.xml файла

#### [javax.persistence-api](https://mvnrepository.com/artifact/javax.persistence/javax.persistence-api)

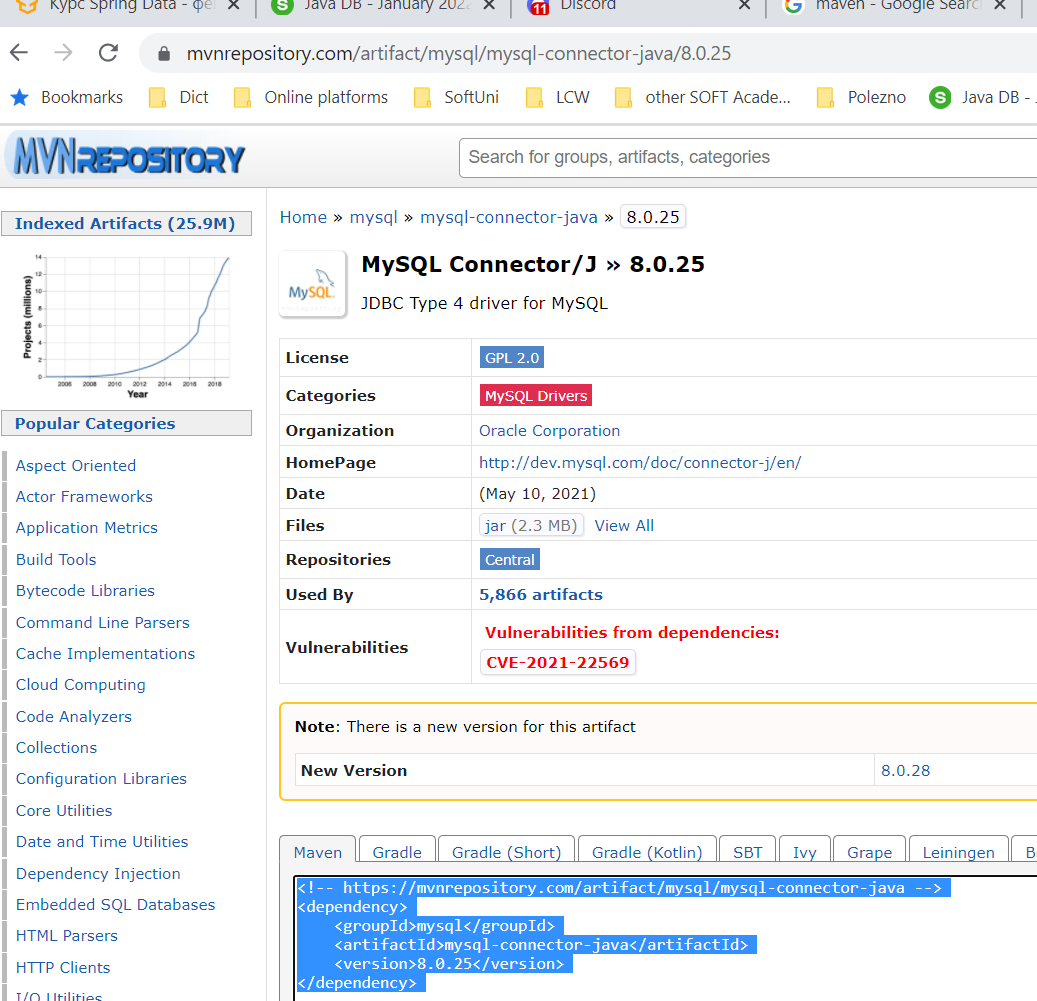
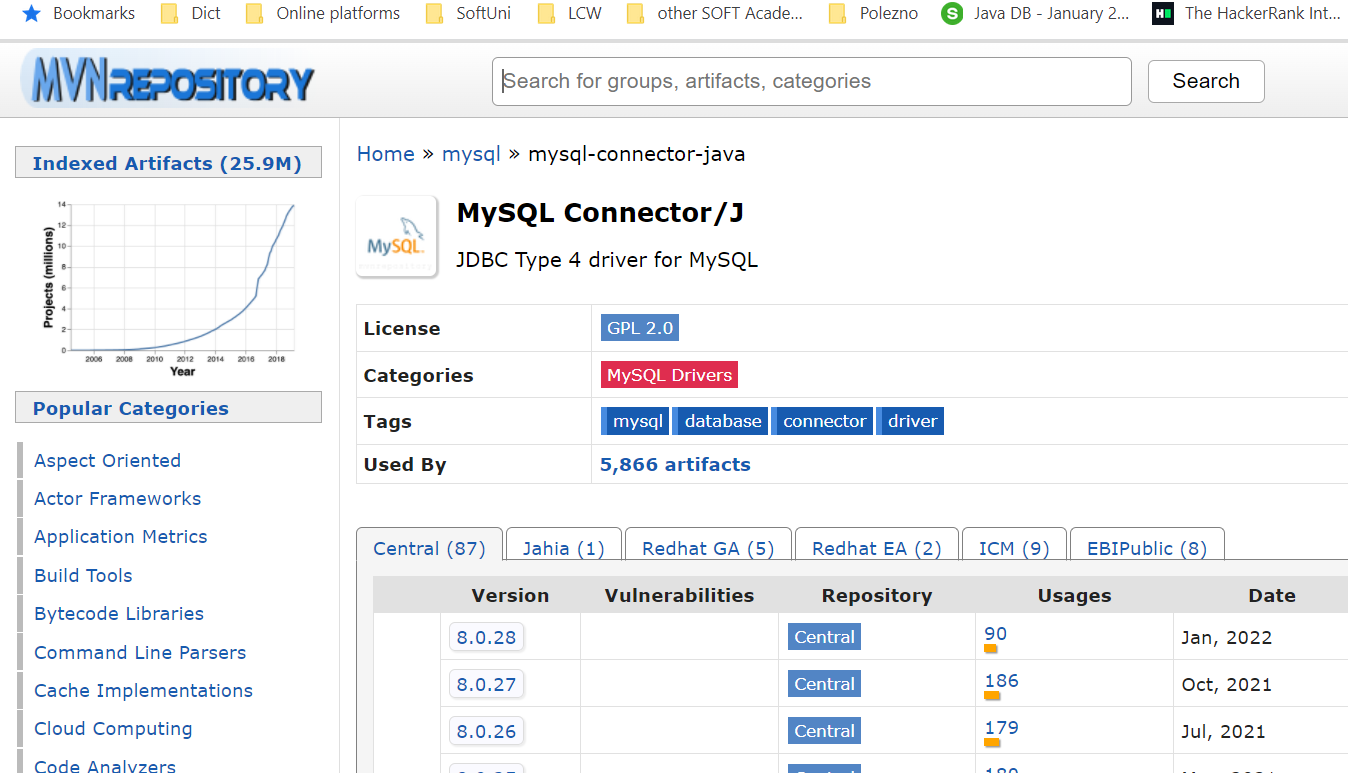
<**dependency**>  
 <**groupId**>javax.persistence</**groupId**>  
 <**artifactId**>javax.persistence-api</**artifactId**>  
 <**version**>2.2</**version**>  
</**dependency**>

#### hibernate-core



<**dependency**>  
 <**groupId**>org.hibernate</**groupId**>  
 <**artifactId**>hibernate-core</**artifactId**>  
 <**version**>5.4.30.Final</**version**>  
</**dependency**>

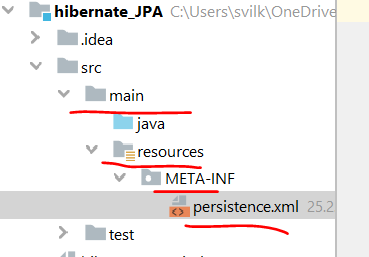
#### mysql-connector-java



<**dependency**>  
 <**groupId**>mysql</**groupId**>  
 <**artifactId**>mysql-connector-java</**artifactId**>  
 <**version**>8.0.25</**version**>  
</**dependency**>

#### persistence.xml – create it in **src/main/resources/META-INF**

META-INF е стандартно име на папка в Java, в която папка биха стояли конфигурации



*<?***xml version="1.0" encoding="UTF-8"***?>*<**persistence xmlns="http://java.sun.com/xml/ns/persistence" version="2.0"**>  
 <**persistence-unit name="school"**> //persistence unit name  
 <**properties**>  
 <**property name="hibernate.connection.url"  
 value="jdbc:mysql://localhost:3306/school?createDatabaseIfNotExist=true"**/>  
 <**property name="hibernate.connection.driver\_class" value="com.mysql.cj.jdbc.Driver"**/>  
 <**property name="hibernate.connection.username" value="root"**/>  
 <**property name="hibernate.connection.password" value=""**/>  
 <**property name="hibernate.dialect" value="org.hibernate.dialect.MySQL8Dialect"**/>

<**property name="hibernate.hbm2ddl.auto" value="update"**/>

<**property name="hibernate.hbm2ddl.auto" value="create-drop"**/> //изтрий таблицата и я пресъздай наново с новите/верните/променените колони

<**property name="hibernate.hbm2ddl.auto" value="validate"**/> //за валидиране на миграции на данни

<**property name="hibernate.show\_sql" value="true"**/>

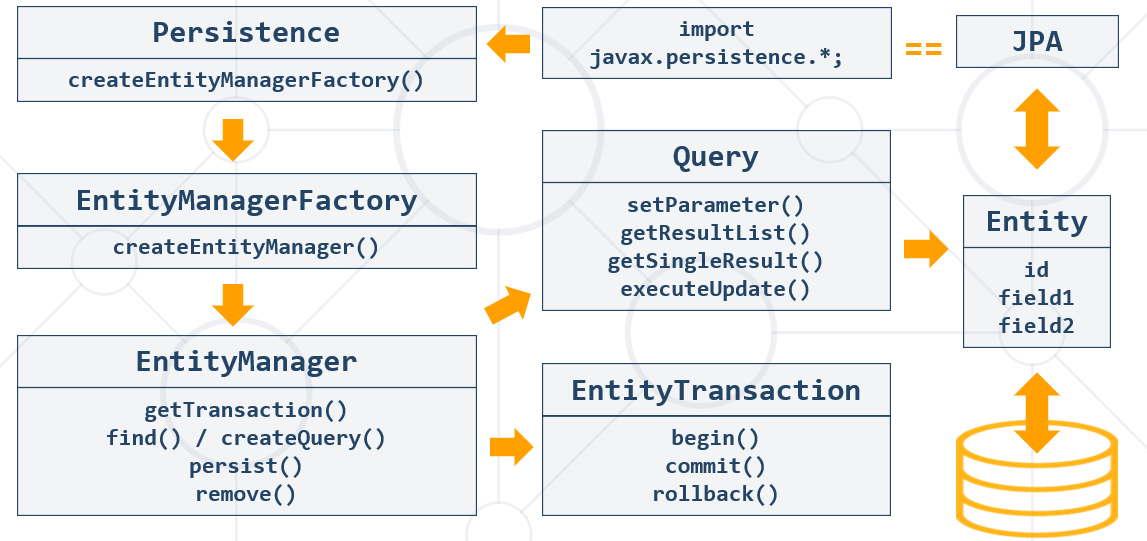
</**properties**>  
 </**persistence-unit**>  
</**persistence**>

### JPA Save Objects

**import** entities.Student;  
**import** javax.persistence.EntityManager;  
**import** javax.persistence.EntityManagerFactory;  
**import** javax.persistence.Persistence;  
  
**public class** Main {  
 **public static void** main(String[] args) {  
 EntityManagerFactory emf = Persistence.*createEntityManagerFactory*(**"school"**);

//persistence unit name e school  
  
 EntityManager em = emf.createEntityManager();  
 em.getTransaction().begin();  
  
 Student student = **new** Student(**"Teo"**);  
 em.persist(student);  
 em.getTransaction().commit();  
 em.close();  
 }  
}

### JPA schema – Java Persistence API

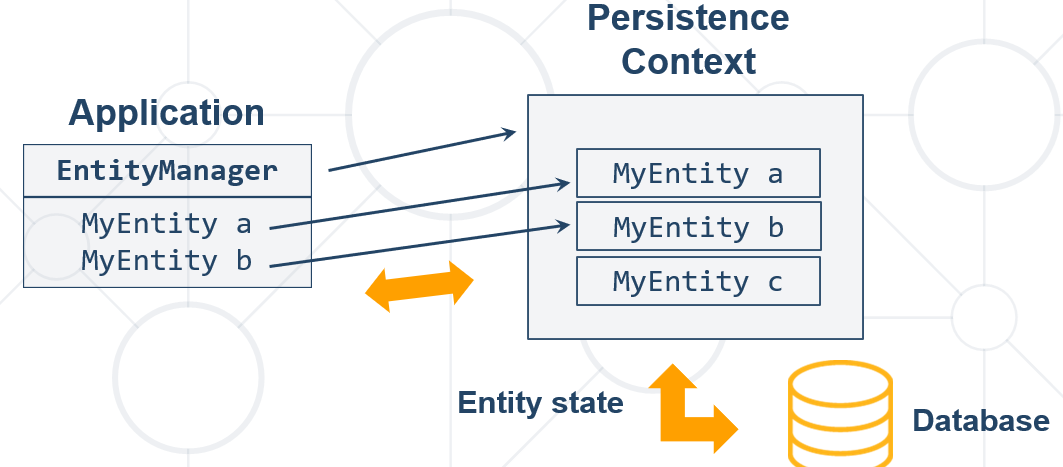


### Java Persistence Query Language = JPQL

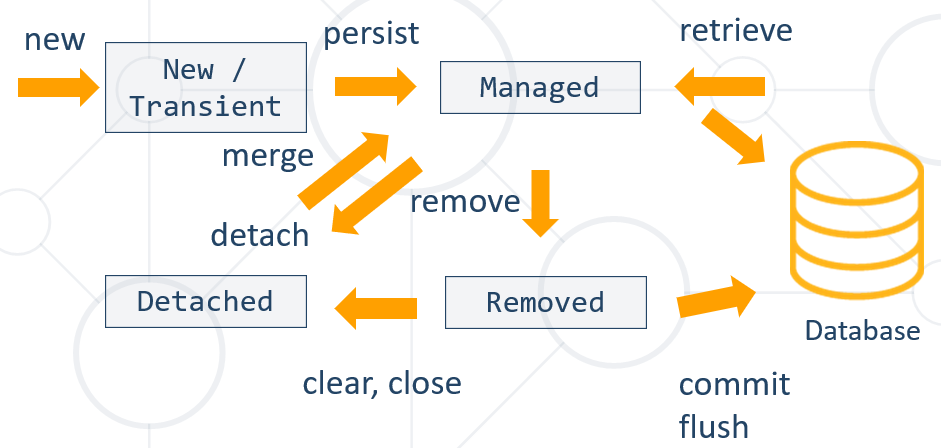
Query query1 = entitymanager.createQuery(**"Select MAX(e.salary) from Employee e"**);

Query query = entitymanager.createQuery( **"Select e "** + **"from Employee e "** + **"where e.salary "** + **"Between 30000 and 40000"** );

### Persistence Context (PC) and Entities



### Entity Object Life Cycle



### JPA Write Data Methods

**Имената на елементите на анотациите са имената на променливите в базата данни!!!**

* **persist()** – persists given entity object into the DB (SQL INSERT)
* **remove()** – deletes given entity into the DB (SQL DELETE by primary key)
* **refresh()** – reloads given entity from the DB (SQL SELECT by primary key)
* **detach()** – removes the object from the persistence context(PC)
* **merge()** – synchronize the state of detached entity with the PC
* **contains()** - determine if given entity is managed by the PC
* **flush()** – writes the changes from PC (Persistence context) in the database

### JPA Read Data Methods

* + **find()** - execute a simple Select query by primary key – one item result only

**public static void** main(String[] args) {  
 EntityManagerFactory emf = Persistence.*createEntityManagerFactory*(**"school"**);  
  
 EntityManager em = emf.createEntityManager();  
 em.getTransaction().begin();  
 em.**find**(Student.**class**, 1) //Get object  
 em.getTransaction().commit();  
}

### JPA Delete Objects

**public static void** main(String[] args) {  
 EntityManagerFactory emf = Persistence.*createEntityManagerFactory*(**"school"**);  
 EntityManager em = emf.createEntityManager();  
 em.getTransaction().begin();

Student student = em.find(Student.**class**, 1);  
 em.**remove**(student); //remove object  
 em.getTransaction().commit();

em.close();

}

### JPA Merge Objects

* Merges the state of **detached** entity into a **managed** **copy** of the detached entity. – ако сме изтрили дадено entity (инстанция на класа Student примерно), то изтритият обект стои в паметта и можем да го върнем в използване с командата **merge**
  + Returned entity has a different Java identity than the detached one
* May invoke SQL SELECT

**public** Student storeUpdatedStudent(Student student) {  
 **return** entityManager.**merge**(student);  
}

### JPA Retrieve Data by Criteria – more than one result

**public static void** main(String[] args) {  
EntityManagerFactory emf = Persistence.createEntityManagerFactory("school");

EntityManager em = emf.createEntityManager();

**em**.getTransaction().begin();

**CriteriaBuilder** builder = **em**.getCriteriaBuilder(); **//by CriteriaBuilder it is type-safe**  
 **CriteriaQuery** criteria = builder.**createQuery()**;  
 **Root**<Student> r = criteria.**from**(Student.**class**);  
 criteria.**select**(r).**where**(builder.like(r.get(**"name"**), **"P%"**));

List<Student> studentList = **em**.createQuery(criteria).**getResultList();** //Get list of objects by criteria

**for** (Student student : studentList) {  
 System.***out***.println(student.getName());  
 }  
  
 session.getTransaction().commit();  
 session.close();  
}

**Без да е type-safe вариантът**

**Навсякъде използваме име на клас, а не име на таблица**

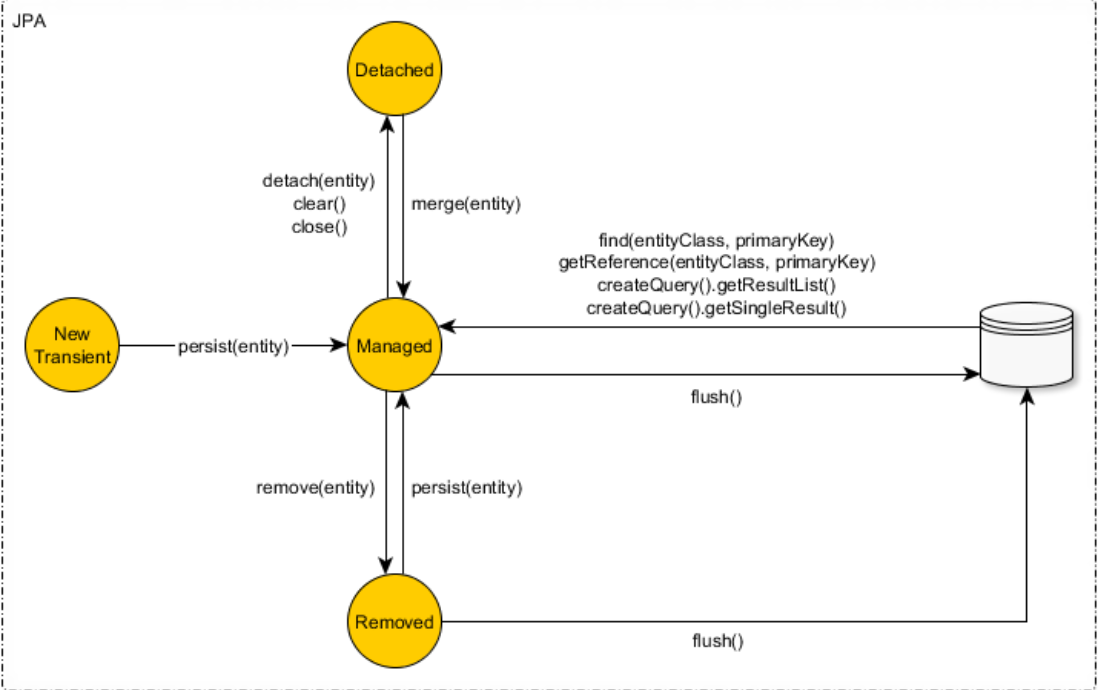
Query from\_town = entityManager.createQuery(**"SELECT t FROM Town t"**, Town.**class**);  
List<Town> resultList = from\_town.getResultList();  
  
**for** (Town town : resultList) {  
 System.***out***.println(town);  
}

## 3.5. Transaction Log and BackUp

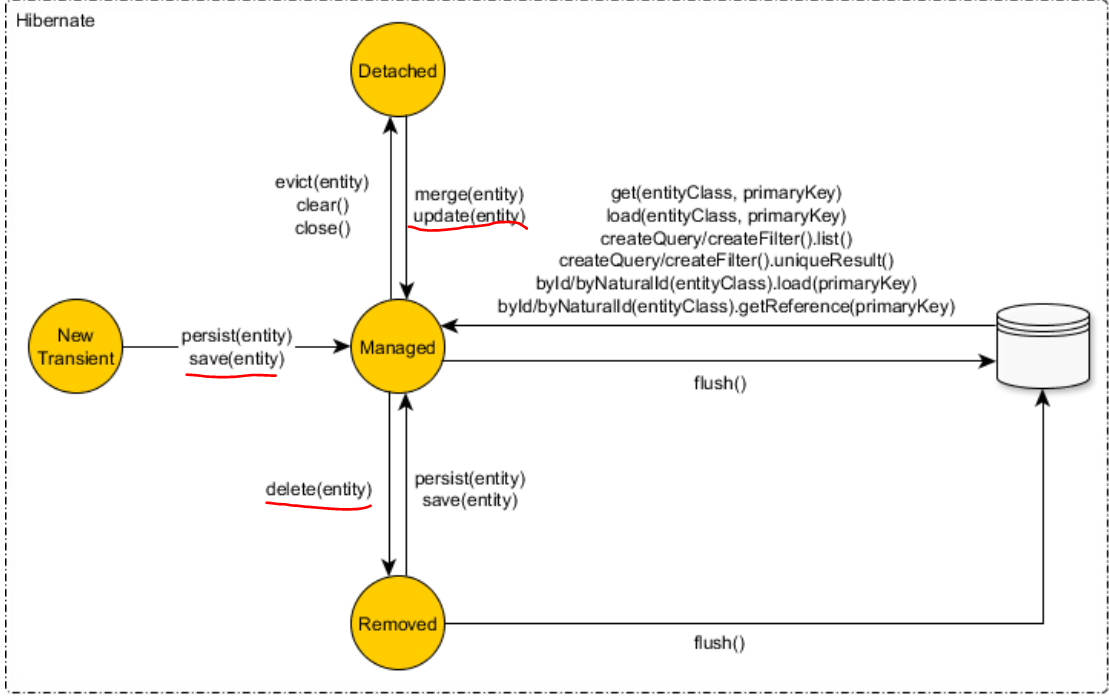
<https://liquibase.org/>

## 3.6. Differences between Hibernate without JPA (existed before JPA) and JPA

<https://vladmihalcea.com/jpa-persist-merge-hibernate-save-update-saveorupdate/>



The Hibernate Session implements all the JPA EntityManager methods and provides some additional entity state transition methods like save, saveOrUpdate and update.



JPA предоставя interfaces, а Hibernate предоставя implementation.

Interface Session (hibernate) extends interface EntityManager(от JPA)

**interface** Session **extends** EntityManager

**Каквото има в JPA, може да се ползва и от hibernate. Но за да използваме вида mapping (xml или annotations), то трябва да използваме само единият от двата варианта.**

За повече универсалност, (да може да се ползва и от други системи), то е добре да използваме Hibernate винаги с JPA методите / само с JPA методите, без собствените методи на Hibernate.

Можем ако сме на Hibernate with JPA да го извадим/преобрзуваме само на Hibernate и да си ползваме оригиналните методи на Hibernate, които не са част от JPA изискуемата спецификация.

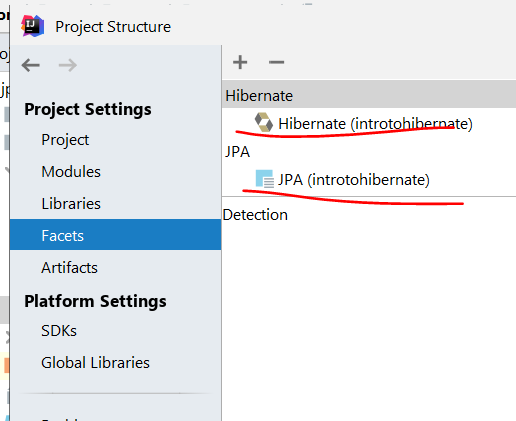
<https://www.netsurfingzone.com/hibernate/get-session-from-entitymanager-in-spring-boot/>

Consider a scenario, we have Spring Boot Application and we need session object to perform some specific operation. In order to access Hibernate APIs(for example Session methods) from JPA, we need a session object from EntityManager. In this post, we will see How To get Hibernate Session From EntityManager in Spring Boot with Example.

**public static void** main(String[] args) {  
 EntityManagerFactory emf = Persistence.*createEntityManagerFactory*(**"school"**);  
  
 EntityManager em = emf.createEntityManager();  
 **Session** unwrap = em.unwrap(**Session**.**class**);

**Session** session = (**Session**) em.**getDelegate()**;

Когато в един проект сме имплементирали и Hibernate и JPA Hibernate



## 3.7. Подход при изпълняване на JPQL заявки

### Подход с CriteriaBuilder – it is type-safety

#### Пример за LIKE употреба

CriteriaBuilder builder = entityManager.getCriteriaBuilder(); *//by CriteriaBuilder it is type-safe*CriteriaQuery criteria = builder.createQuery();  
Root<Employee> r = criteria.from(Employee.**class**);  
CriteriaQuery **criteriaQuery** = criteria.select(r).where(builder.like(r.get(**"firstName"**),  
 String.*format*(**"%s"**, pattern) + **"%"**));  
  
*//We put the criteria query here*List<Employee> resultList = entityManager.createQuery(**criteriaQuery**)  
 .getResultList();  
  
**for** (Employee e : resultList) {  
 String toPrint = String.*format*(**"%s %s - %s - ($%.2f)"**, e.getFirstName(), e.getLastName(),  
 e.getJobTitle(), e.getSalary());  
 System.***out***.println(toPrint);  
}

### Подход с директна Update JPQL заявка в базата данни *// entityManager.createQuery("UPDATE Town t SET t.name=lower(t.name) WHERE length(t.name) >=5")***.executeUpdate();**

### Подход взимаме нещата от базата, правим нещо с данните, и записваме/update-ваме само, което ни трябва

Query from\_town = entityManager.createQuery(**"SELECT t FROM Town t"**, Town.**class**);  
List<Town> resultList = from\_town.getResultList();  
  
**for** (Town town : resultList) {  
 String name = town.getName();  
 **if** (name.length() <= 5) {  
 String toUpper = name.toUpperCase();  
 town.setName(toUpper);  
  
 entityManager.persist(town);  
 }  
}

## 3.8. Специфики при използване на JPA JPQL заявки

### Класът Employee

@Entity  
@Table(name = **"employees"**)  
**public class** Employee {  
 **private** Integer **id**;  
 **private** String **firstName**;  
 **private** String **lastName**;  
 **private** String **middleName**;  
 **private** String **jobTitle**;  
 **private** Department **department**;  
 **private** Employee **manager**;  
 **private** LocalDateTime **hireDate**;  
 **private** BigDecimal **salary**;  
 **private** Address **address**;  
 **private** Set<Project> **projects**;

### Set-ване на полета – вместо ? въпросче, то използваме :Име на параметър в JPA

вместо ? въпросче,

то използваме две точки: Име на параметър и след това set-ваме параметъра

Както казахме още в съвсем началото Hibernate и JPA използва заявки имената на класовете и имената полетата на класовете!!!

Long employeeCount = entityManager.createQuery(**"SELECT COUNT(e) FROM Employee e"** +  
 **" WHERE e.firstName = :first\_name"** +  
 **" AND e.lastName = :last\_name"**,  
 Long.**class**) //типа на връщания резултат, който си искаме да ни върне  
 .**setParameter**(**"first\_name"**, searchFor[0])  
 .**setParameter**(**"last\_name"**, searchFor[1])  
 .**getSingleResult**(); //върни един резултат  
  
**if** (employeeCount > 0L) {  
 System.***out***.println(**"Yes"**);  
} **else** {  
 System.***out***.println(**"No"**);  
}

### getSingleResult и getResultList in JPA

List<String> resultList = entityManager.createQuery(**"SELECT e.firstName FROM Employee e"** +  
 **" WHERE e.salary > 50000"**,  
 String.**class**) //върни getSingleResult да е от тип String  
 .**getResultList()**; //върни колкото резултати String има под формата на List<String>

### getResultStream и заобикаляне на JOIN

String department = **"Research and Development"**;  
  
entityManager.createQuery(**"SELECT e FROM Employee e"** +  
 **" WHERE e.department.name = :departmentName"** + //без да използваме join 😊  
 **" ORDER BY e.salary ASC, e.id ASC"**,  
 Employee.**class**)  
 .setParameter(**"departmentName"**, department)  
 .**getResultStream()**  
 .**forEach**(e -> {  
 String format = String.*format*(**"%s %s from %s - $%.2f"**,  
 e.getFirstName(), e.getLastName(), department, e.getSalary());  
  
 System.***out***.println(format);  
 });

### getResultStream и SET на полета на обекти, които се оказва се записват и в базата данни

List<String> updateCriteria = Arrays.*asList*(**"Engineering"**, **"Tool Design"**, **"Marketing"**, **"Information Services"**);  
  
entityManager.createQuery(**"SELECT e FROM Employee e"** +  
 **" WHERE e.department.name IN (:params)"**,  
 Employee.**class**)  
 .setParameter(**"params"**, updateCriteria)  
 .**getResultStream()**  
 .**forEach**(е -> **е.setSalary**(е.getSalary().multiply(BigDecimal.*valueOf*(1.12)))); // set на полета на обекти, които се оказва се записват и в базата данни

### executeUpdate и параметър като цял обект

String addressText = **"Vitoshka 15"**;  
Address address = **new** Address(); //цял обект  
address.setText(addressText);  
  
entityManager.persist(address);

*//без да type-cast-ваме – няма опция да му кажем какъв обект връща createQuery*entityManager.createQuery(**"UPDATE Employee e"** +  
 **" SET e.address = :addr"** +  
 **" WHERE e.lastName = :employeeName"**)   
 .setParameter(**"employeeName"**, lastName)  
 .setParameter(**"addr"**, address)  
 **.executeUpdate();**

### Заобикаляне на COUNT и LIMIT

entityManager.createQuery(**"FROM Address a"** +  
 **" ORDER BY a.employees.size DESC"**, *//вместо COUNT(a.employees)* Address.**class**)  
 .**setMaxResults**(10) *//вместо LIMIT 10 в заявката* .getResultStream()

.limit(10) //тук вече сме преточили цялата база данни и от stream-а лимитираме 10  
 .forEach(System.***out***::println);

### IN в JPQL

**List<String>** updateCriteria = **Arrays.*asList***(**"Engineering"**, **"Tool Design"**, **"Marketing"**, **"Information Services"**);

List<Employee> resultList = entityManager.createQuery(**"SELECT e FROM Employee e"** +  
 **" WHERE e.department.name IN (:params)"**, //може и без скоби след IN  
 Employee.**class**)  
 .setParameter(**"params"**, updateCriteria)  
 .getResultList();  
  
resultList.stream()  
 .forEach(e -> System.***out***.println(String.*format*(**"%s %s ($%.2f)"**,  
 e.getFirstName(), e.getLastName(), e.getSalary())));

### TypedQuery

**public class** CustomEmployee {  
 **public** Department **department**;  
 **public** BigDecimal **salary**;  
  
 **public** CustomEmployee() {  
 }

String query = "FROM Employee e" +

                "  GROUP BY e.department" +

                " HAVING MAX(e.salary) NOT BETWEEN :min AND :max";

        TypedQuery<CustomEmployee> typQuery = entityManager.createQuery(query, CustomEmployee.**class**);

        typQuery.setParameter("min", 30000);

        typQuery.setParameter("max", 70000);

        List<CustomEmployee> resultList1 = typQuery.getResultList();

**for** (CustomEmployee customEmployee : resultList1) {

            System.out.println(String.format("%s %.2f", customEmployee.department.getName(), customEmployee.salary));

        }

### Върни List<Object[]> от SELECT параметрите

List<Object[]> resultList = entityManager.createQuery(**"SELECT e.department.name, MAX(e.salary) FROM Employee e"** +  
 **" GROUP BY e.department.id"** +  
 **" HAVING MAX(e.salary) NOT BETWEEN 30000 AND 70000"**,  
 Object[].**class**)  
 .getResultList();  
  
**for** (Object[] o : resultList) {  
 String departName = (String) o[0];  
 BigDecimal salary = (BigDecimal) o[1];  
 System.***out***.printf(**"%s %.2f%n"**, departName, salary);

### Подход дали да използваме заявка със Select или директно да връщаме класа започвайки с FROM

entityManager.createQuery(**"SELECT e FROM Employee e"**

И двата варианта са възможни.

### Заобикаляне на JOIN, плюс CONCAT и return String.class

List<String> resultList = entityManager.createQuery(**"SELECT CONCAT(d.name, ' ', MAX(e.salary)) FROM Employee AS e"** +  
 **" JOIN e.department AS d"** +  
 **" GROUP BY d.id"** +  
 **" HAVING MAX(e.salary) NOT BETWEEN 30000 AND 70000"**,  
 String.**class**)  
 .getResultList();  
  
**for** (String s : resultList) {  
 System.***out***.printf(**"%s%n"**, s);  
}

# 4. Hibernate (JPA) Code First Entity Relations - Advanced Mapping

## 4.1. Java Persistence API Inheritance

Fundamental Inheritance Concepts

### Inheritance

* Inheritance is a fundamental concept in most programming languages
  + SQL does not support this kind of relationships
* Implemented by any JPA framework by **inheriting** and **mapping** **Entities**

### JPA Inheritance Strategies

* Implemented by the **javax.persistence.Inheritance** annotation
* The following mapping strategies are used to map the entity data to the underlying database:
  + A single **table per class** hierarchy – една таблица за клас
  + A table per **concrete entity class**
  + "**Join**" strategy – mapping common fields in a single table

### 1. ***TABLE\_PER\_CLASS*** Strategy

* **Table creation for each entity**
  + A table defined for each concrete class in the inheritance
  + Allows inheritance to be used in the object model, when it does not exist in the data model
* Querying root or branch classes can be very difficult and **inefficient**

#### Sample:

**import** javax.persistence.\*;  
  
@Entity

@Table(name = **"vehicles"**)

@Inheritance(strategy = InheritanceType.***TABLE\_PER\_CLASS***)  
**public abstract class** Vehicle {  
 @Id  
 @GeneratedValue(strategy = GenerationType.***TABLE***) //*пазим уникални ключове измежду много таблици*  
 **private int id**;  
  
 @Basic  
 **private** String **type**;

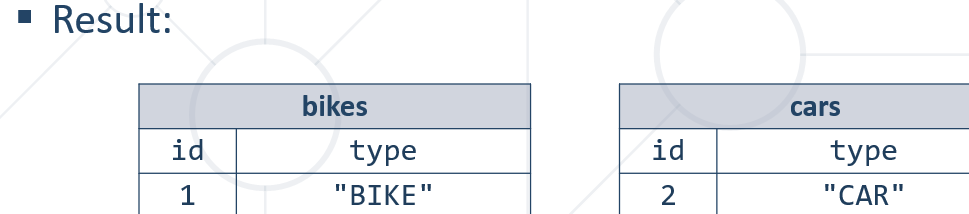
}

**import** javax.persistence.Entity;  
**import** javax.persistence.Table;  
  
@Entity  
@Table(name = **"bikes"**)  
**public class** Bike **extends** Vehicle {  
 **private static final** String ***BIKE\_TYPE*** = **"Bike"**;  
  
 **private int gearCount**;  
  
 **public** Bike(**int** gearCount) {  
 **super**(***BIKE\_TYPE***);  
 **this**.**gearCount** = gearCount;  
 }

}

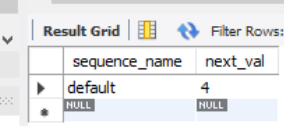
**import** javax.persistence.Entity;  
**import** javax.persistence.Table;  
  
@Entity  
@Table(name = **"cars"**)  
**public class** Car **extends** Vehicle {  
 **private static final** String ***CAR\_TYPE*** = **"Car"**;  
  
 **private int doors**;  
  
 **public** Car(**int** doors) {  
 **super**(***CAR\_TYPE***);  
 **this**.**doors** = doors;  
 }

EntityManagerFactory emf = Persistence.*createEntityManagerFactory*(**"soft\_uni"**);  
EntityManager entityManager = emf.createEntityManager();  
  
entityManager.getTransaction().begin();  
Bike bike = **new** Bike(21);  
Car car = **new** Car(5);  
  
entityManager.persist(bike);  
entityManager.persist(car);  
  
entityManager.getTransaction().commit();  
entityManager.close();



За няколко класа от общия abstract-ен клас Vehicle пази таблица **hibernate\_sequences** в базата данни, която следи за поредното id на абстрактния клас Vehicle.





#### Table Per Class Strategy: Conclusion

* **Disadvantages**:
  + Repeating information in each table
  + Changes in super class involves changes in all subclass tables – промяна в една таблица, промяна във таблиците навързани по йерархията
  + No foreign keys involved (unrelated tables)
* **Advantages**:
  + No NULL values – no unneeded fields
  + Simple style to implement inheritance mapping

### 2. ***JOINED*** Strategy

* Table is defined for each class in the inheritance hierarchy
  + Storing of that class **only the local attributes**
  + Each table must store object's **primary key**

#### Example:

@Entity  
@Table(name = **"vehicles"**)  
@Inheritance(strategy = InheritanceType.***JOINED***)  
**public abstract class** Vehicle {  
 @Id  
 @GeneratedValue(strategy = GenerationType.***TABLE***) // *пазим уникални ключове измежду много таблици*  
 **private int id**;  
  
 @Basic  
 **private** String **type**;  
  
 **private double price**;

@MappedSuperclass *//нещо междинно го анотира***public abstract class** TransportationVehicle **extends** Vehicle{  
 **private int loadCapacity**;  
  
 **public** TransportationVehicle() {  
 }  
  
 **public** TransportationVehicle(String type, **double** price, **int** loadCapacity) {  
 **super**(type, price);  
 **this**.**loadCapacity** = loadCapacity;  
 }

@Entity  
@Table(name = **"trucks"**)  
**public class** Truck **extends** TransportationVehicle {  
 **private static final** String ***TRUCK\_TYPE*** = **"Truck"**;  
  
 **public** Truck(**double** price, **int** loadCapacity) {  
 **super**(***TRUCK\_TYPE***, price, loadCapacity);  
 }  
}

main

Bike bike = **new** Bike(21);  
Car car = **new** Car(5);  
Truck truck = **new** Truck(25000, 40000);  
  
entityManager.persist(bike);  
entityManager.persist(car);  
entityManager.persist(truck);

#### Results – Joined Strategy

* **Disadvantages**:
  + Multiple JOINS - for deep hierarchies it may give poor performance
* **Advantages**:
  + No NULL values
  + No repeating information
  + Foreign keys involved
  + Reduced changes in schema on superclass changes – ако има промяна в truck, промяната се отразява само в trucks таблицата в базата данни

### 3. ***SINGLE\_TABLE*** Strategy – **най-използваната/предпочитаната**

* **Simplest** and typically the best performing and best solution
  + A single table is used to store all the instances of the **entire** **inheritance hierarchy**
  + A column for every attribute of every class
  + A **discriminator column** is used to determine to which class the particular row belongs to

Когато типа е колело, то от началото до края типа е колело в рамките на SQL базата данни!

#### Example

@Entity  
@Table(name = **"vehicles"**)  
@Inheritance(strategy = InheritanceType.***SINGLE\_TABLE***)  
@DiscriminatorColumn(name = **"type"**) //указваме коя колона от базата данни следи за типовете  
**public abstract class** Vehicle {  
 @Id  
 @GeneratedValue(strategy = GenerationType.***TABLE***) *//може в случая и .IDENTITY, за да не пазим уникални ключове измежду много таблици*

**private int id**;  
  
 @Basic  
 @Column(insertable = **false**, updatable = **false**)  
 **private** String **type**;  
  
 **private double price**;  
  
 **public** Vehicle() {  
 }  
  
 **public** Vehicle(String type, **double** price) {  
 **this**.**type** = type;  
 **this**.**price** = price;  
 }

@MappedSuperclass *//нещо междинно го анотира***public abstract class** TransportationVehicle **extends** Vehicle{  
 **private int loadCapacity**;  
  
 **public** TransportationVehicle() {  
 }  
  
 **public** TransportationVehicle(String type, **double** price, **int** loadCapacity) {  
 **super**(type, price);  
 **this**.**loadCapacity** = loadCapacity;  
 }

@Entity  
@Table(name = **"trucks"**)  
@DiscriminatorValue(**"truck"**) //какво да пише в дискриминационната колона type  
**public class** Truck **extends** TransportationVehicle {  
 **private static final** String ***TRUCK\_TYPE*** = **"Truck"**;  
  
 **public** Truck(**double** price, **int** loadCapacity) {  
 **super**(***TRUCK\_TYPE***, price, loadCapacity);  
 }  
}

@Entity  
@Table(name = **"bikes"**)  
@DiscriminatorValue(**"bike"**) //какво да пише в дискриминационната колона type  
**public class** Bike **extends** Vehicle {  
 **private static final** String ***BIKE\_TYPE*** = **"Bike"**;  
  
 **private int gearCount**;  
  
 **public** Bike(**int** gearCount) {  
 **super**(***BIKE\_TYPE***, 250);  
 **this**.**gearCount** = gearCount;  
 }

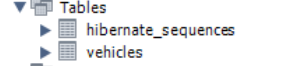
@Entity  
@Table(name = **"cars"**)  
@DiscriminatorValue(**"car"**) //какво да пише в дискриминационната колона type  
**public class** Car **extends** Vehicle {  
 **private static final** String ***CAR\_TYPE*** = **"Car"**;  
  
 **private int doors**;  
  
 **public** Car(**int** doors) {  
 **super**(***CAR\_TYPE***, 2500);  
 **this**.**doors** = doors;  
 }

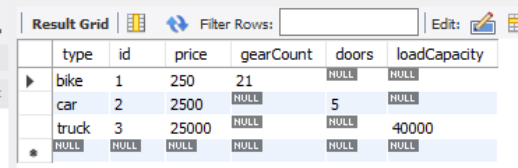
main

Bike bike = **new** Bike(21);  
Car car = **new** Car(5);  
Truck truck = **new** Truck(25000, 40000);  
  
entityManager.persist(bike);  
entityManager.persist(car);  
entityManager.persist(truck);

#### Results – Joined Strategy

Създава само една таблица **vehicles,** и hibernate\_sequences както обикновено





Hibernate не е прост, той ще задейства за даден тип само определените колони!

## 4.2. Table Relations

### Database Relationships

There are several types of database relationships:

* **One to One** Relationships
* **One to Many** and **Many to One** Relationships
* **Many to Many** Relationships
* **Self Referencing** Relationships

**Правило – да ги държим връзките еднопосочни по възможност!**

**Която таблица е по-важна, там държим връзката. В случая, шампоанът логично е по значим от съставките и от production batch-a.**

### One-To-One – Unidirectional - Еднопосочно

**Шампоана знае към кой етикет се отнася, но и етикетът НЕ знае към кой шампоан е**



@Entity  
@Table(name = **"shampoos"**)  
**public class** BasicShampoo {  
 @Id  
 @GeneratedValue(strategy = GenerationType.***IDENTITY***)  
 **private int id**;  
  
 **private** String **name**;  
  
 @OneToOne(optional = **false**) //**Runtime evaluation**  
 @JoinColumn(name = **"label\_id"**, referencedColumnName = **"id"**) *//name е при нашата таблица, referencedColumnName е отсрещната таблица* //**Column name in table shampoos, referencedColumnName in table labels****private** BasicLabel **label**;

main

BasicLabel label = **new** BasicLabel(**"blue"**);  
BasicShampoo shampoo = **new** BasicShampoo(**"shower"**, label);  
  
entityManager.persist(label); //първо създаваме label  
entityManager.persist(shampoo); //след това създаваме shampoo

### One-To-One – Bidirectional – двупосочна

**Шампоанът знае към кой етикет се отнася, но и етикетът знае към кой шампоан е**



@Entity  
@Table(name = **"shampoos"**)  
**public class** BasicShampoo {  
 @Id  
 @GeneratedValue(strategy = GenerationType.***IDENTITY***)  
 **private int id**;  
  
 **private** String **name**;  
  
 @OneToOne(optional = **false**)  
 @JoinColumn(name = **"label\_id"**, referencedColumnName = **"id"**) *//name е при нашата таблица, referencedColumnName е отсрещната таблица* **private** BasicLabel **label**;

@Entity  
@Table(name = **"labels"**)  
**public class** BasicLabel {  
 @Id  
 @GeneratedValue(strategy = GenerationType.***IDENTITY***)  
 **private int id**;  
 **private** String **color**;  
  
 @OneToOne(mappedBy = **"label"**, //**Field in entity BasicShampoo (не е колона от базата данни)**

targetEntity = BasicShampoo.**class**) // **Entity for the mapping**  
 **private** BasicShampoo **shampoo**;

main

BasicLabel label = **new** BasicLabel(**"blue"**);  
BasicShampoo shampoo = **new** BasicShampoo(**"shower"**, label);  
  
entityManager.persist(label); //първо създаваме label  
entityManager.persist(shampoo); //след това създаваме shampoo

Създава се същите таблици, но вече имаме двустранна връзка

### Many-To-One – Unidirectional



@Entity  
@Table(name = **"batches"**)  
**public class** ProductionBatch {  
 @Id  
 @GeneratedValue(strategy = GenerationType.***IDENTITY***)  
 **private int id**;  
  
 @Column(name = **"created\_at"**)  
 **private** LocalDate **createdAt**;

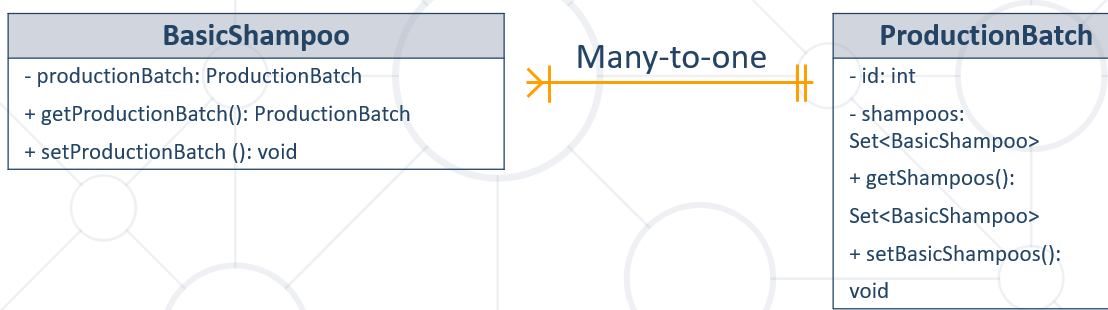
@Entity  
@Table(name = **"shampoos"**)  
**public class** BasicShampoo {  
 @Id  
 @GeneratedValue(strategy = GenerationType.***IDENTITY***)  
 **private int id**;  
  
 **private** String **name**;  
  
 @OneToOne(optional = **false**) **//имам ли право да има NULL стойност в базата, false няма право(NOT NULL в SQL)**  
 @JoinColumn(name = **"label\_id"**, referencedColumnName = **"id"**) *//name е при нашата таблица, referencedColumnName е отсрещната таблица* **private** BasicLabel **label**;  
  
 *//много шампони да принадлеждат на една партида* @ManyToOne(optional = **false**) **//имам ли право да има NULL стойност в базата, false няма право (NOT NULL в SQL)**  
 @JoinColumn(name = **"batch\_id"**, referencedColumnName = **"id"**) // **Column name in   
table shampoos, Column name in table batches**

**private** ProductionBatch **batch**;

**main**

ProductionBatch batch = **new** ProductionBatch(LocalDate.*now*());  
BasicLabel label = **new** BasicLabel(**"blue"**);  
BasicShampoo shampoo = **new** BasicShampoo(**"shower"**, label, batch);  
  
entityManager.persist(batch);  
entityManager.persist(label);  
entityManager.persist(shampoo);

### One-To-Many – Bidirectional



@Entity  
@Table(name = **"batches"**)  
**public class** ProductionBatch {  
 @Id  
 @GeneratedValue(strategy = GenerationType.***IDENTITY***)  
 **private int id**;  
  
 @Column(name = **"created\_at"**)  
 **private** LocalDate **createdAt**;  
  
 *//един batch да знае, че има много шампоани* @OneToMany (mappedBy = **"batch"**, targetEntity = BasicShampoo.**class**, **//поле на класа и кой е класа** fetch = FetchType.***LAZY***, *//не ги вземай/зареждай всички шампоани всеки път //FetchType.EAGER е обратното* cascade = CascadeType.***ALL***) *//ако изтриеш batch какво да прави със свързаните неща/записи* **private** Set<BasicShampoo> **shampoos**;

**main**

ProductionBatch batch = **new** ProductionBatch(LocalDate.*now*());  
BasicLabel label = **new** BasicLabel(**"blue"**);  
BasicShampoo shampoo = **new** BasicShampoo(**"shower"**, label, batch);  
  
entityManager.persist(batch);  
entityManager.persist(label);  
entityManager.persist(shampoo);  
  
ProductionBatch productionBatch = entityManager.find(ProductionBatch.**class**, 1);  
Set<BasicShampoo> shampoos = productionBatch.getShampoos();  
  
System.***out***.println(shampoos);

### Many-To-Many – Unidirectional

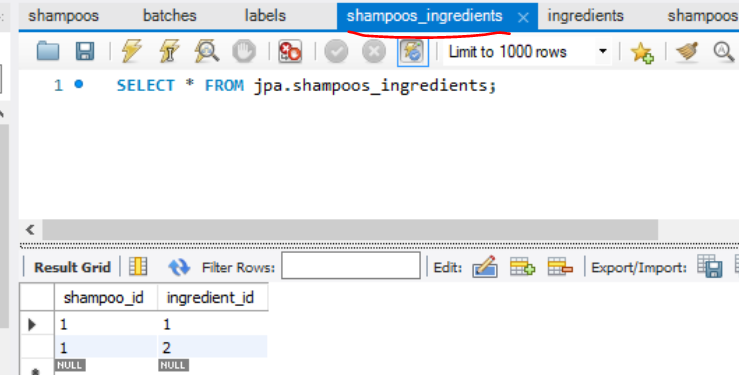
@Entity  
@Table(name = **"ingredients"**)  
**public class** BasicIngredient {  
 @Id  
 @GeneratedValue(strategy = GenerationType.***IDENTITY***)  
 **private int id**;  
  
 **private double quantity**;  
  
 @Column(name = **"chemical\_name"**, nullable = **false**)  
 **private** String **chemicalName**;  
  
 **public** BasicIngredient() {  
 }  
  
 **public** BasicIngredient(**double** quantity, String chemicalName) {  
 **this**.**quantity** = quantity;  
 **this**.**chemicalName** = chemicalName;  
 }

@Entity  
@Table(name = **"shampoos"**)  
**public class** BasicShampoo {  
 @Id  
 @GeneratedValue(strategy = GenerationType.***IDENTITY***)  
 **private int id**;  
  
 **private** String **name**;  
  
 @OneToOne(optional = **false**)  
 @JoinColumn(name = **"label\_id"**, referencedColumnName = **"id"**)  
 *//name е при нашата таблица, referencedColumnName е отсрещната таблица* **private** BasicLabel **label**;  
  
 *//много шампони да принадлеждат на една партида* @ManyToOne(optional = **false**)  
 @JoinColumn(name = **"batch\_id"**, referencedColumnName = **"id"**)  
 **private** ProductionBatch **batch**;  
  
 @ManyToMany  
 @JoinTable(name = **"shampoos\_ingredients"**,  
 joinColumns = @JoinColumn(name = **"shampoo\_id"**, referencedColumnName = **"id"**), //текущия клас - //работи с SQL колоните от базата, id реално не е анотирано с Column и с различно име в базата  
 inverseJoinColumns = @JoinColumn(name = **"ingredient\_id"**, referencedColumnName = **"id"**)) //класът, който реферираме, а именно BasicIngredient  
 **private** Set<BasicIngredient> **ingredients**;  
  
 **public** BasicShampoo() {  
 }  
  
 **public** BasicShampoo(String name, BasicLabel label, ProductionBatch batch) {  
 **this**.**name** = name;  
 **this**.**label** = label;  
 **this**.**batch** = batch;  
 **this**.**ingredients** = **new** HashSet<>();  
 }  
  
 **public void** addIngredient(BasicIngredient basicIngredient){  
 **this**.**ingredients**.add(basicIngredient);  
 }  
  
 **public** Set<BasicIngredient> getIngredients() {  
 **return** Collections.*unmodifiableSet*(**this**.**ingredients**); //да не можем да го модифицираме  
 }

main

ProductionBatch batch = **new** ProductionBatch(LocalDate.*now*());  
BasicLabel label = **new** BasicLabel(**"blue"**);  
BasicShampoo shampoo = **new** BasicShampoo(**"shower"**, label, batch);  
  
BasicIngredient ingredient = **new** BasicIngredient(100, **"B12"**);  
BasicIngredient ingredient2 = **new** BasicIngredient(2, **"Violet"**);  
  
shampoo.addIngredient(ingredient);  
shampoo.addIngredient(ingredient2);  
entityManager.persist(ingredient);  
entityManager.persist(ingredient2);  
  
entityManager.persist(batch);  
entityManager.persist(label);  
entityManager.persist(shampoo);

Mapping таблицата



### Many-To-Many – Bidirectional

Ако искаме да знаем със съставка лавандула в колко вида шампоана е използвана, може да направим и обратна връзка.

@Entity  
@Table(name = **"ingredients"**)  
**public class** BasicIngredient {  
 @Id  
 @GeneratedValue(strategy = GenerationType.***IDENTITY***)  
 **private int id**;  
  
 **private double quantity**;  
  
 @Column(name = **"chemical\_name"**, nullable = **false**)  
 **private** String **chemicalName**;

//Обратната връзка  
 @ManyToMany(mappedBy = **"ingredients"**, targetEntity = BasicShampoo.**class**) **//поле на класа и кой е класа**  
 **private** Set<BasicShampoo> **shampoos**;

@Entity  
@Table(name = **"shampoos"**)  
**public class** BasicShampoo {  
 @Id  
 @GeneratedValue(strategy = GenerationType.***IDENTITY***)  
 **private int id**;  
  
 **private** String **name**;  
  
 @OneToOne(optional = **false**)  
 @JoinColumn(name = **"label\_id"**, referencedColumnName = **"id"**) //работи с SQL колоните от базата, id реално не е анотирано с Column и с различно име в базата  
 *//name е при нашата таблица, referencedColumnName е отсрещната таблица* **private** BasicLabel **label**;  
  
 *//много шампони да принадлеждат на една партида* @ManyToOne(optional = **false**)  
 @JoinColumn(name = **"batch\_id"**, referencedColumnName = **"id"**)  
 **private** ProductionBatch **batch**;  
  
 @ManyToMany  
 @JoinTable(name = **"shampoos\_ingredients"**,  
 joinColumns = @JoinColumn(name = **"shampoo\_id"**, referencedColumnName = **"id"**), //от текущия клас  
 inverseJoinColumns = @JoinColumn(name = **"ingredient\_id"**, referencedColumnName = **"id"**)) //от класа BasicIngredient  
 **private** Set<BasicIngredient> **ingredients**;

### Self-Reference

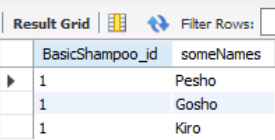
@Entity  
@Table(name = **"employees"**)  
**public class** Employee {  
 **private** Integer **id**;  
 **private** String **firstName**;  
 **private** String **lastName**;  
 **private** Employee **manager**;  
  
  
 @Id  
 @GeneratedValue(strategy = GenerationType.***IDENTITY***)  
 @Column(name = **"employee\_id"**)  
 **public** Integer getId() {  
 **return id**;  
 }  
  
 **public void** setId(Integer id) {  
 **this**.**id** = id;  
 }  
  
 @Column(name = **"first\_name"**)  
 **public** String getFirstName() {  
 **return firstName**;  
 }  
  
 **public void** setFirstName(String firstName) {  
 **this**.**firstName** = firstName;  
 }  
  
 @Column(name = **"last\_name"**)  
 **public** String getLastName() {  
 **return lastName**;  
 }  
  
 **public void** setLastName(String lastName) {  
 **this**.**lastName** = lastName;  
 }  
  
  
  **@ManyToOne**  
 **@JoinColumn**(name = **"manager\_id"**, referencedColumnName = **"employee\_id"**) **//работи с SQL колоните от базата, id реално тук Е анотирано с Column анотация и с различно име в базата (“employee\_id”) спрямо полето на текущия клас** **public** Employee getManager() {  
 **return manager**;  
 }

### Използване на @ElementCollection

@Entity  
@Table(name = **"shampoos"**)  
**public class** BasicShampoo {  
 @Id  
 @GeneratedValue(strategy = GenerationType.***IDENTITY***)  
 **private int id**;  
  
 **private** String **name**;  
  
………………  
 *//Ако ни трябват група от елементи, които нямат отсрещна таблица* @ElementCollection  
 **private** List<String> **someNames**;  
  
 **public void** setSomeNames(List<String> someNames) {  
 **this**.**someNames** = someNames;  
 }

main

ProductionBatch batch = **new** ProductionBatch(LocalDate.*now*());  
BasicLabel label = **new** BasicLabel(**"blue"**);  
BasicShampoo shampoo = **new** BasicShampoo(**"shower"**, label, batch);  
  
BasicIngredient ingredient = **new** BasicIngredient(100, **"B12"**);  
BasicIngredient ingredient2 = **new** BasicIngredient(2, **"Violet"**);  
  
shampoo.addIngredient(ingredient);  
shampoo.addIngredient(ingredient2);  
entityManager.persist(ingredient);  
entityManager.persist(ingredient2);  
  
entityManager.persist(batch);  
entityManager.persist(label);  
entityManager.persist(shampoo);  
  
ProductionBatch productionBatch = entityManager.find(ProductionBatch.**class**, 1);  
Set<BasicShampoo> shampoos = productionBatch.getShampoos();  
  
List<String> names = Arrays.*asList*(**"Pesho"**, **"Gosho"**, **"Kiro"**);  
shampoo.**setSomeNames**(names);



### Lazy Loading – Fetch Types

* Fetching – retrieve objects from the database
  + Fetched entities are stored in the **Persistence Context** as cache
* Retrieval of an entity object might cause automatic retrieval of **additional** entity objects

### Fetching Strategies

* Fetching Strategies
  + EAGER – retrieves all entity objects reachable through fetched entity
    - Can cause **slowdown** when used with a big data source
  + **LAZY** – retrieves all reachable entity objects **only when fetched entity's getter method is called**

University **university** = em.find((**long**) 1); *// collection students = null  
  
// The collection holding the students is populated when the getter is called*university.getStudents();

Когато почнем да принтираме например, чак тогава ги вика всичките!!!

### Cascading

* JPA translates **entity state transitions** to database **DML** statements
  + This behavior is configured through the **CascadeType** mappings
* **CascadeType.PERSIST**: means that save() or persist() operations cascade to related entities
* **CascadeType.MERGE**: means that related entities are merged into managed state when the owning entity is merged
* **CascadeType.REFRESH**: does the same thing for the refresh() operation
* **CascadeType.REMOVE**: removes all related entities association with this setting when the owning entity is deleted
* **CascadeType.DETACH**: detaches all related entities if a "manual detach" occurs
* **CascadeType.ALL**: is shorthand for all of the above cascade operations

# Other

Zero-days vulnarabilities – не се знае от колко време е бил пробивът и колко хора са имали достъп/изтривали нещо/добавяли нещо

Не трябва да се разминават схемите между Java И базата!!!

По спецификация винаги трябва да имаме празен конструктор. Операцията persist не го изисква. Но съгласно custom-orm, при търсене примерно find създаваме getDeclaredConstructor().newInstance(); и тук търси празен конструктор, който ние трябва да сме декларирали ръчно също.