1. Database Access with JDBC (Java DataBase Connectivity)

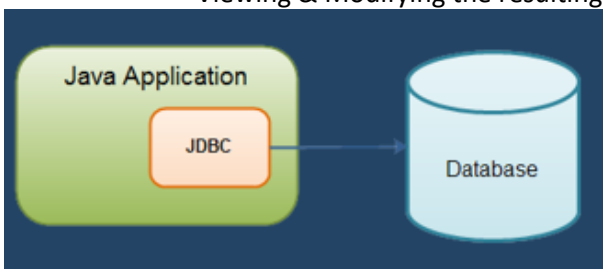
JDBC дава възможност да се свързваме към всякакъв вид бази

1.1. Intro

JDBC is a standard interface for connecting to relational databases from Java

- The JDBC Core API package is **java.sql**
- JDBC 2.0 Optional Package API is **javax.sql**
- JDBC 3.0 API includes the Core API and Optional Package API
- Latest version is JDBC 4.3

- 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

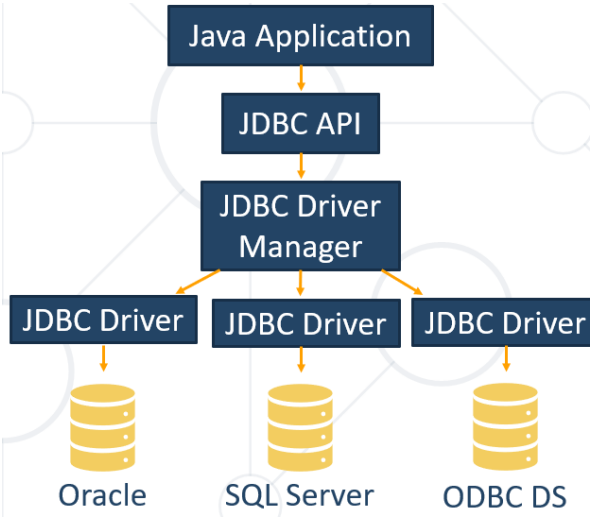


Java bin концепция/конвенция

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

1.2. JDBC Client Access to a Database

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**

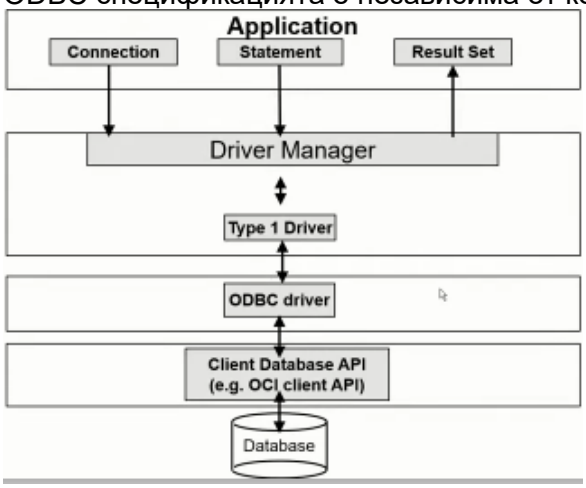
JDBC drivers

Four types of JDBC drivers:

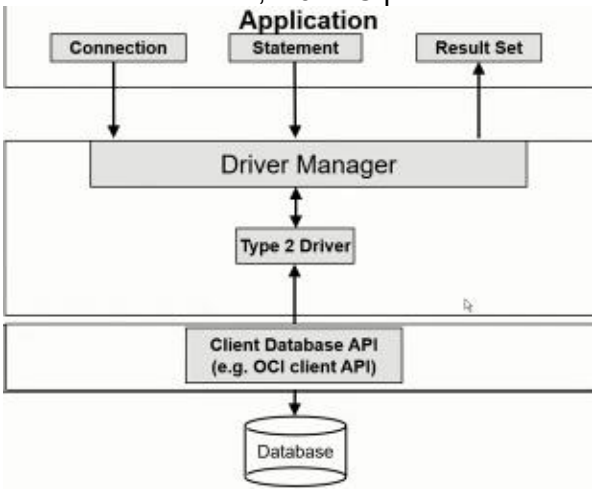
От 1 към 4 се намалява нивото на абстракция.

- Type 1: JDBC:**ODBC** bridge driver - използва се Object Database Connectivity API **protocol** (спецификация) - имплементирано от повечето големи бази данни като MySQL, PostgreSQL, Oracle.

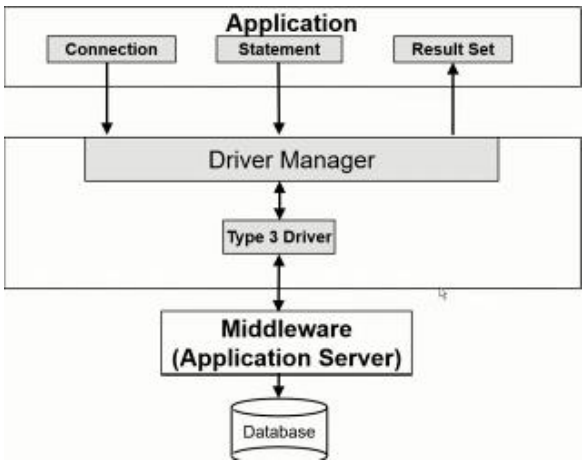
ODBC спецификацията е независима от конкретния програмен език - било то Java, Python, etc.



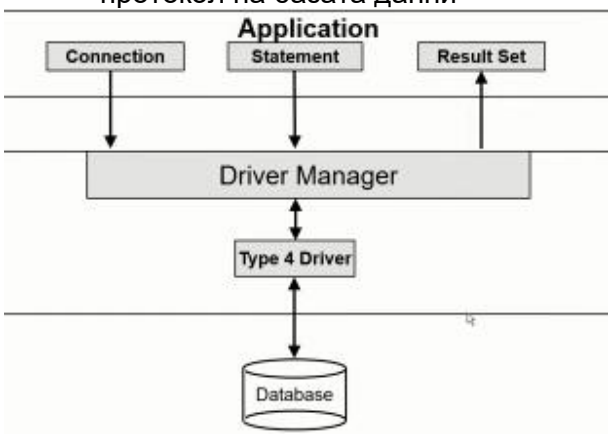
- Type 2: **Native API driver** (native client library provided for database) - конкретна клиентска библиотека специфична за дадената база данни. Самата база данни си има специфична библиотека, и JDBC работи с тази специфична библиотека.



- Type 3: **Network protocol driver** (driver talks to middleware) - на мрежово ниво, самия JDBC driver комуникира с това междинно ниво, а не директно с базата данни



- Type 4: **Database protocol driver** - имплементация на/от JDBC, която работи директно с протокол на базата данни



JDBC API

- JDBC API provides several interfaces and classes:
  - **DriverManager** – matches requests from the application with the proper DB driver
  - **JDBC 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**

Driver Manager - loads database drivers, and manages the connection between application & driver
JDBC Driver - translates API calls to operations for a specific data source (database abstraction)
Connection - a session between an application and the relational database
Statement - a representation of an SQL statement
Metadata - information about the returned data, driver and database
Result Set - logical set of columns and rows returned by executing a statement

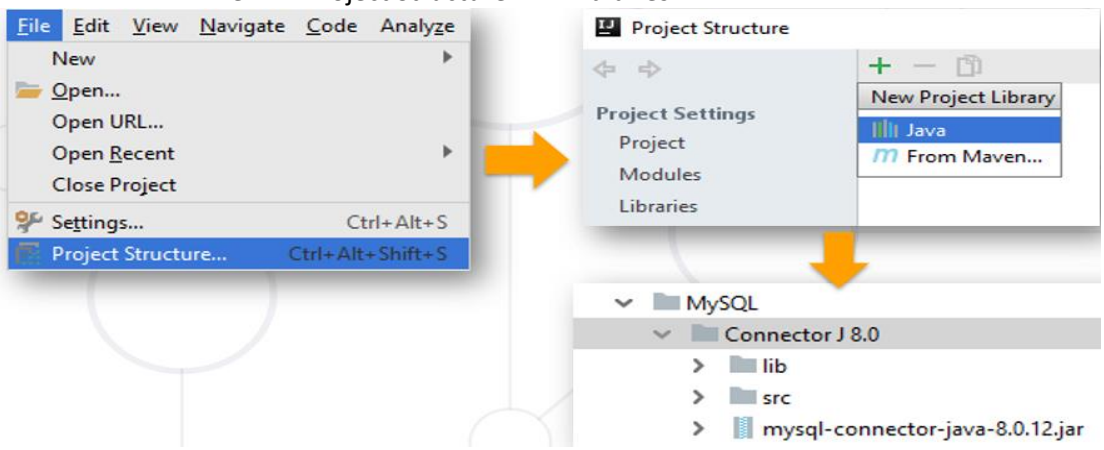
Java.sql* and MySQL Driver

- 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/

Setting Up the Driver in IntelliJ IDEA

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



1.3. JDBC API - Querying the Database

Stage 1: loading the JDBC driver

Loading the JDBC driver is done by a single line of code:
Class.forName("<jdbc driver class>");

Your driver documentation will give you the class name to use. Loading MySQL and Oracle driver (we can google it to see the right version):

```
Class.forName("com.mysql.jdbc.Driver");
Class.forName("oracle.jdbc.driver.OracleDriver");
```

Optional!!!!

Stage 2: establishing a connection

Main info

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

```
Connection c = DriverManager.getConnection(
"jdbc:mysql://localhost:3306/soft_uni", props);
```

Database name

Credentials

```
Connection connection =
        DriverManager.getConnection( url: "jdbc:mysql://localhost:3306/soft_uni?useSSL=false", props);

System.out.println
```

```
getConnection(String url, Properties info)                    Connection
getConnection(String url)                                     Connection
getConnection(String url, String user, String password)       Connection
```

JDBC **connection string** components
Jdbc : <driver protocol> : <connection details>

The following line of code illustrates the process:
Connection connection = DriverManager.getConnection("url", “user”, “pass”);

If you are using a JDBC-ODBC bridge driver, the JDBC URL will start with `jdbc:odbc:`

- ODBC data source is called “`Library`”
- DBMS login name is “admin”
- The password is “secret”

```
Connection connection = DriverManager
       .getConnection("jdbc:odbc:Library",
               "admin",
               "secret");
```

If you are using a JDBC driver developed by a third party, the documentation will tell you what sub-protocol to use.
You can also use this website for help - https://www.connectionstrings.com/

If you want to connect to MySQL

- The db server is MySQL 5.6, 5.7 or 8.0, locally installed
- MySQL database schema is called “`soft_uni`”
- The user and password is “hr”

```
Connection connection = DriverManager
       .getConnection("jdbc:mysql://localhost:3306/soft_uni?” +
“user=hr&password=hr");
```

If you want to connect to Oracle database

- The db server is 10g Express Edition, locally installed
- Oracle database schema is called “`HR`”
- The user and password is “hr”

```
Connection connection = DriverManager
       .getConnection("jdbc:oracle:thin:@localhost:1521/HR?”, “hr”, “hr");
```

A JDBC connection string is **used to load the driver and to indicate the settings that are required to establish a connection to the data source**. These settings are referred to as connection properties.

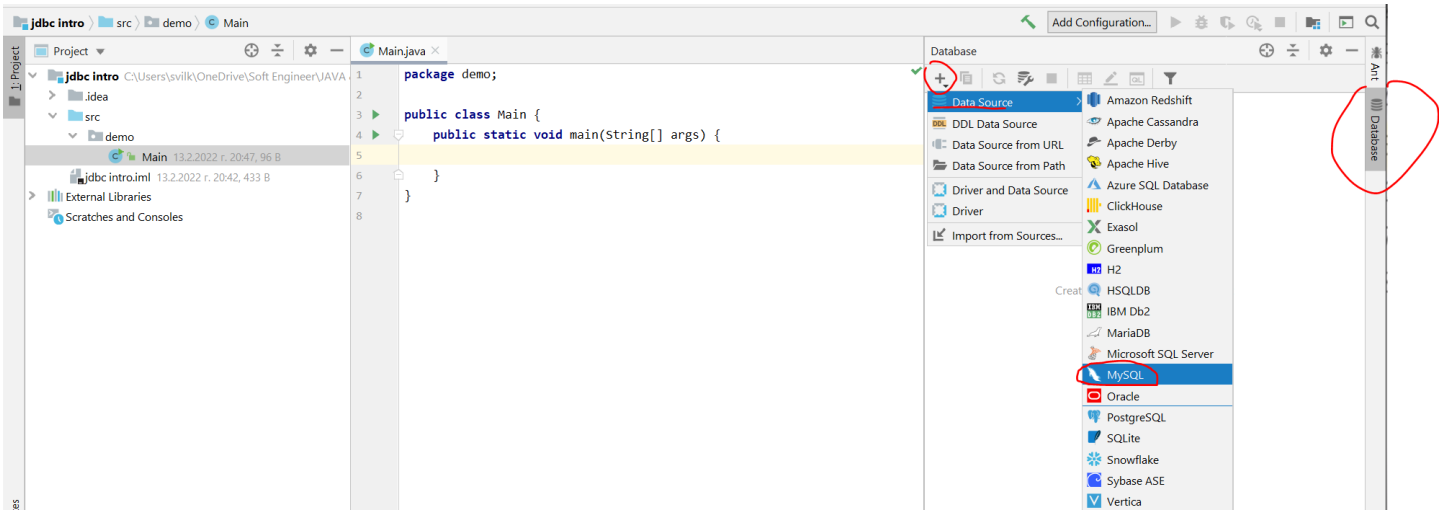
What is a JDBC connection string?

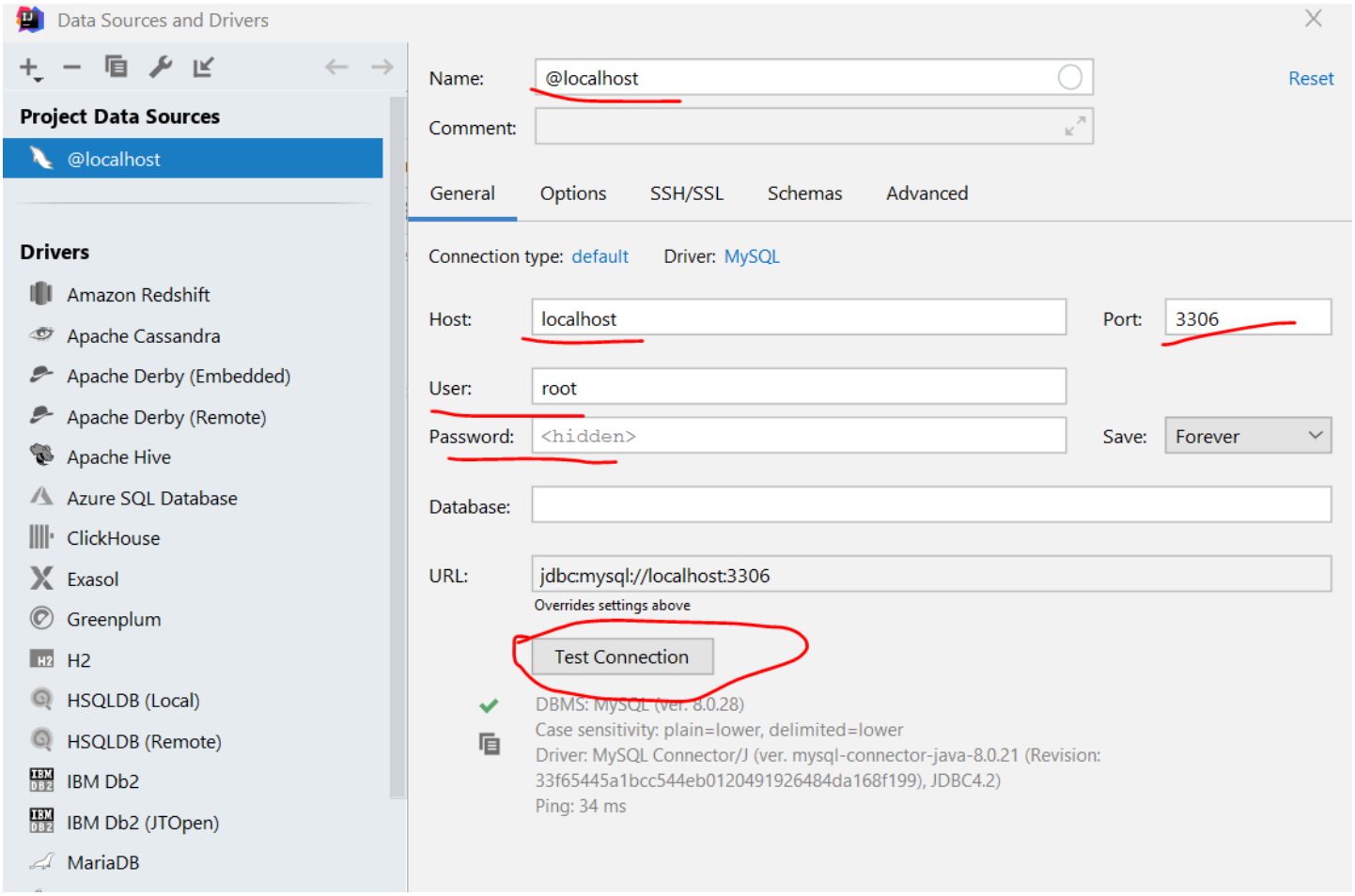
a. A string containing database connection settings
b. A string containing database host and port
c. A string containing database type, host and port
d. A connection string having all database connection settings

IntelliJ IDE

Connection to DB Via Java App Demo – part 1

Import-ваме базата данни чрез Plug-in -а Database отдясно отстрани

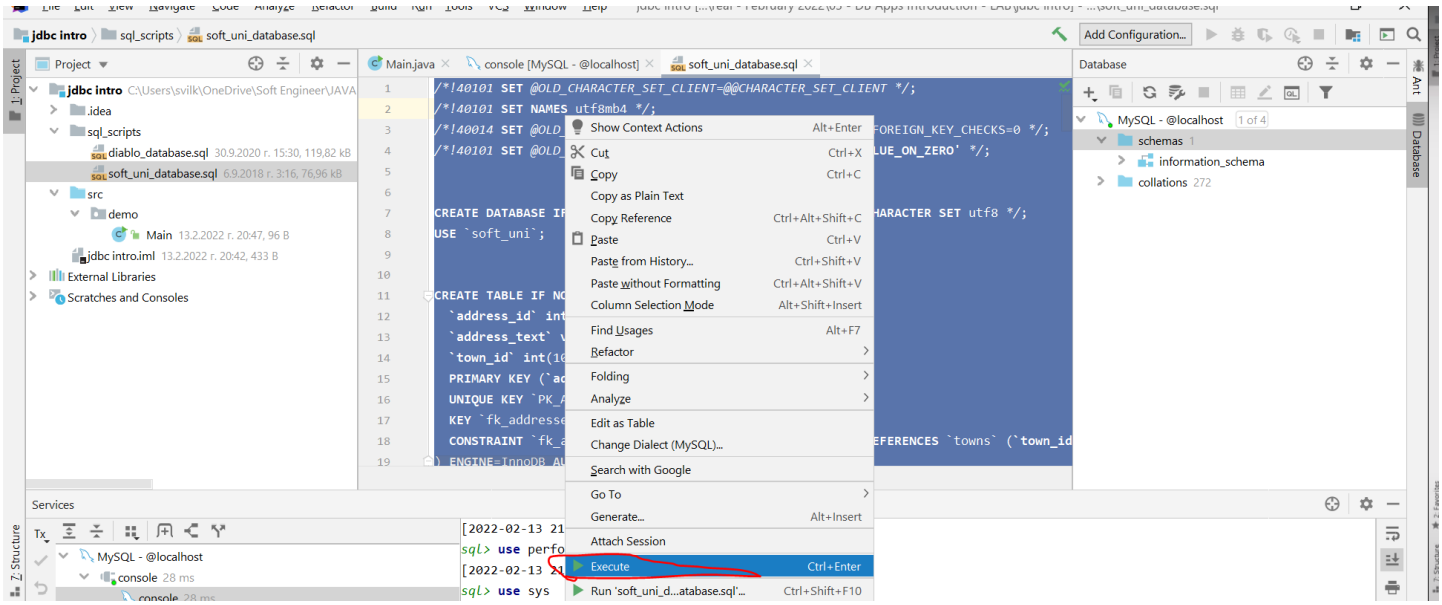




Избираме диалекта – MySQL / Maria DB match best

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.

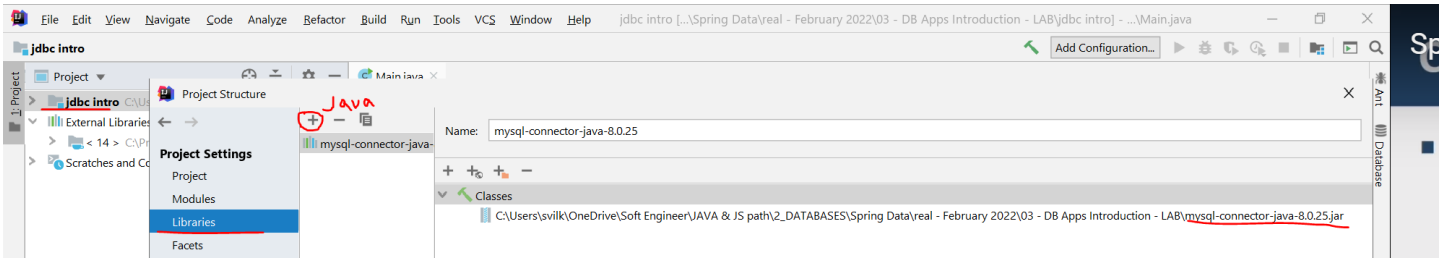
Новият и старият driver:

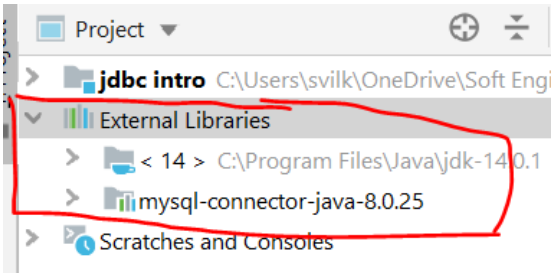
– за MySQL 8 е `"com.mysql.cj.jdbc.Driver"`

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

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

File -> Project structure …..






```
//2. Connect to DB - Пишем конекцията във формата даден в презентацията
Connection connection =
        DriverManager.getConnection( url: "jdbc:mysql://localhost:3306/soft_uni?useSSL=false", props);

System.out.println
```

```
getConnection(String url, Properties info)          Connection
getConnection(String url)                           Connection
getConnection(String url, String user, String password)  Connection
Pressing Ctrl+Space twice without a class qualifier would show all accessible static methods  Next Tip
```

Properties object

You can pass database specific information to the database by using Properties object. Example for oracle database:

```
java.util.Properties props = new java.util.Properties();
props.put("user", "Scott");
props.put("password", "tiger123");
props.put("defaultRowPrefetch", "30");
props.put("defaultBatchValue", "5");
Connection dbConn = DriverManager.getConnection("jdbc:oracle:thin:@hoststring", props);
```

Stage 3: creating a statement

A statement object sends the SQL commands to the DBMS.

- executeQuery() is used for SELECT statements
- executeUpdate() is used for statements that create/modify tables

An instance of active Connection is used to create a Statement object

```
Connection conn = DriverManager
       .getConnection("jdbc:odbc:Library", "admin", "secret");
Statement stmt = conn.createStatement(); //изпраща заявката към базата данни
```

Stage 4: executing a statement

executeQuery() executes SQL command through a previously created statement
Returns the results in a ResultSet object

```
Connection conn = DriverManager
       .getConnection("jdbc:odbc:Library", "admin", "secret");
```

```
Statement stmt = conn.createStatement(); //изпраща заявката към базата данни

//never returns null
ResultSet rs = stmt.executeQuery("SELECT first_name FROM employees");


//either (1) the row count for SQL Data Manipulation Language (DML) statements or (2)
0 //for SQL statements that return nothing
```

executeUpdate() is used to submit DML/DDL SQL statements

- DML is used to manipulate existing data in objects (using UPDATE, INSERT, DELETE statements)
- DDL is used to manipulate database objects (CREATE, ALTER, DROP)

```
Connection conn = DriverManager
       .getConnection("jdbc:odbc:Library", "admin", "secret");
Statement stmt = conn.createStatement(); //изпраща заявката към базата данни
```

При executeUpdate връща дали е успешно или не (0 или 1)

```
int rowsAffected = stmt.executeUpdate("UPDATE employees SET salary = salary * 1.2");
```

Stage 5: processing the result

The ResultSet object

- 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

```
//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

```
ResultSet resultSet = stmt.executeQuery("SELECT last_name, salary FROM
employees");
while (resultSet.next()){  //Returns: true if the new current row is valid;
false if there are no more rows. Moves the cursor to the first row of the
result
   String lastName = resultSet.getString("last_name"); //column label
   double salary = resultSet.getDouble("salary");
   System.out.println(lastName+ " " + salary);
}
```

- 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);
```

Stage 6: closing the connection

Explicitly close a Connection, Statement and ResultSet to release resources that are no longer needed.
Ако не затворим връзката, може да имаме memory leak.

```
try {
   Connection conn = ...;
   Statement stmt = ...;
   ResultSet resultSet = ...;
} finally {
   // clean up
   resultSet.close();
   stmt.close();
   conn.close();
}
```

SQLException

SQL statements can throw java.sql.SQLEception during their execution.

```
try {
   Connection conn = ...;
   Statement stmt = ...;
   ResultSet resultSet = ...;
} catch (SQLException sqlex) {
   ... //Handle SQL errors here
} finally {
   // clean up all used resources
   try {
       if (resultSet != null) resultSet.close();
       if (stmt != null) stmt.close();
       if (conn != null) conn.close();
   } catch (SQLException sqlex){
       ... // Ignore closing errors
   }
}
```

1.4. JDBC API - Modifying the Database

To execute a DML statement

Delete заявка

```
Statement stmt = conn.createStatement(); //изпраща заявката към базата данни
```

// either (1) the row count for SQL Data Manipulation Language (DML) statements or (2) 0 for SQL statements that return nothing

```
int rowsDeleted = stmt.executeUpdate("DELETE FROM order_items WHERE order_id =
2354");
```

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;
}
```

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.");
```

To execute a DDL statement

```
Statement stmt = conn.createStatement(); //изпраща заявката към базата данни

int rowsDeleted = stmt.executeUpdate("CREATE TABLE temp(
col1 NUMBER(5,2), col2 VARCHAR(30))");
```

To execute an unknown statement

```
Statement stmt = conn.createStatement(); //изпраща заявката към базата данни
```

// true if the first result is a ResultSet object; false if it is an update count or there are no results

```
boolean result = stmt.execute(SQLstatement);

if (result) { // was a query - process results
   // Returns: the current result as a ResultSet object or null if the result is an
update count or there are no more results
   ResultSet resultSet = stmt.getResultSet();
} else { // was an update or DDL - process result
   // Returns: the current result as an update count; -1 if the current result is a
ResultSet object or there are no more results
   int count = stmt.getUpdateCount();
}
```

Само `stmt.execute()` изпълняваме и при stored procedures

Prepared statement

Used to:

- Execute a statement that takes parameters
- Execute a given statement many times

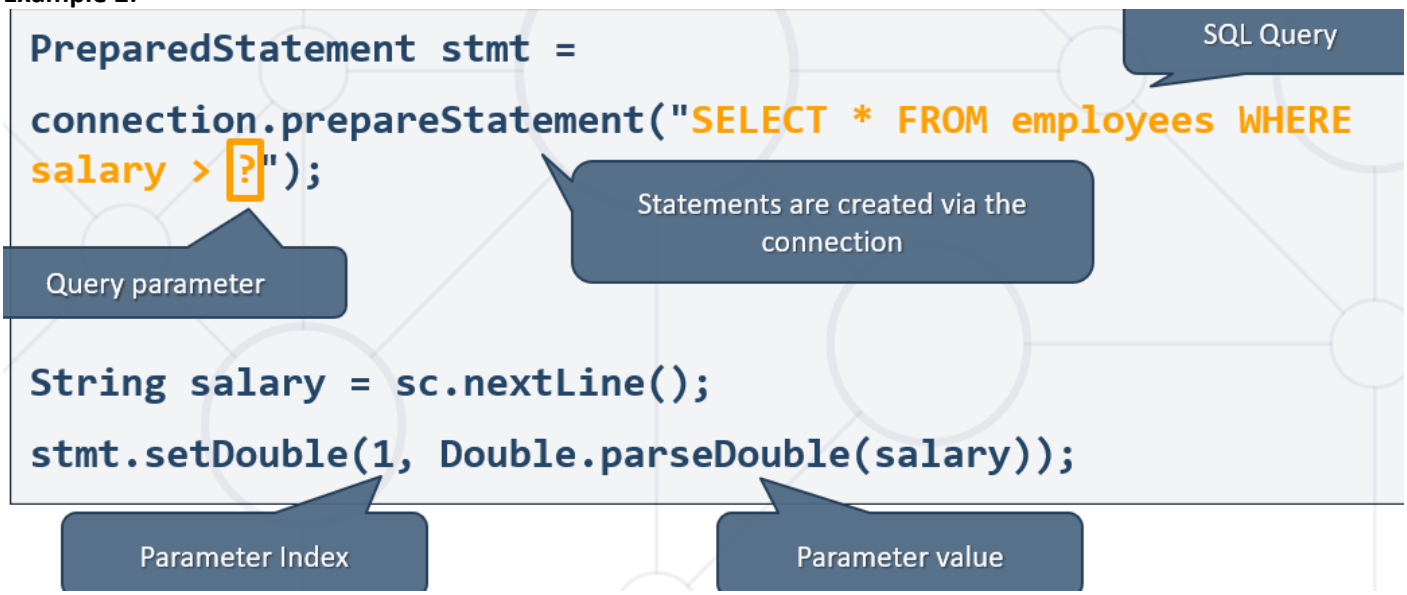
Така наречената шаблонна заявка. Т.е. създаваме типизирани заявки.

Example 1:

```
Connection conn = DriverManager.getConnection("jdbc:odbc:Library", "admin", "secret");
String insertSQL = "INSERT INTO employees (first_name, last_name, salary) VALUES(?,?,?)";
PreparedStatement prepStatement = conn.prepareStatement(insertSQL);
prepStatement.setString(1, "Martin"); //parameter index 1
prepStatement.setString(2, "Toshev"); //parameter index 2
prepStatement.setDouble(3, 1000.0); //parameter index 3
prepStatement.executeUpdate(); при модификация insert
```

Предпазваме се от SQL injection чрез използването на PreparedStatement!

Example 2:




```
//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(); //при SELECT sql заявка

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`
```

Auto increment primary key column

Many databases support “auto increment” primary key columns.

- E.g. MS SQL Server, MS Access. MySQL, PostgreSQL
- JDBC can retrieve auto generated keys

```
Connection conn = DriverManager.getConnection("jdbc:odbc:Library", "admin", "secret");
Statement stmt = conn.createStatement();
// Insert row and return PK
int rowsCount = stmt.executeUpdate("INSERT INTO Messages(Msg) VALUES('Test')",
       Statement.RETURN_GENERATED_KEYS); //The constant indicating that generated keys
should be made available for retrieval.

// Get the auto generated PK
ResultSet rs = stmt.getGeneratedKeys();
rs.next(); //Moves the cursor forward one row from its current position. A ResultSet
cursor is initially positioned before the first row; the first call to the method next
makes the first row the current row;
long primaryKeyFromColumnIndex = rs.getLong(1);
long primaryKeyFromColumnLabel = rs.getLong("Msg");
```

Oracle does not support “auto increment” primary key. Use sequences to generate unique values.

```
Statement stmtSeq = dbConnection.createStatement();
ResultSet resultSetNextId = stmtSeq.executeQuery("SELECT <some_sequence>.nextval FROM
dual");
resultSetNextId.next();
long nextId = resultSetNextId.getLong(1); //column index

PreparedStatement psInsert = dbConnection.prepareStatement("INSERT INTO Table(id, name)
VALUES (?,?)");
psInsert.setLong(1, nextId);
psInsert.setString(2, "Svilen");
psInsert.execute();
```

Calling stored procedures

CallableStatement interface

- Is used for executing stored procedures
- Can pass input parameters
- Can retrieve output parameters

Demo 1

```
Connection dbConn = DriverManager.getConnection("jdbc:odbc:Library", "admin", "secret");
```

```
CallableStatement callableStmt = dbConn.prepareCall("call SP_Insert_Msg(?,?)");
callableStmt.setString(1, msgText);
callableStmt.registerOutParameter(2, Types.BIGINT); //ако базата данни го поддържа
callableStmt.executeUpdate(); за модификация insert
long id = callableStmt.getLong(2);
```

Demo 2

```
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(); за get sql заявка!!

        //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);

    }
}
```

1.5. JDBC Statement, PreparedStatement, CallableStatement

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

Interfaces	Recommended use
Statement	For general-purpose access to your database and static SQL statements at runtime. Cannot accept parameters.
PreparedStatement	For SQL statements used many times. Accepts parameters.
CallableStatement	Used for stored procedures. Accepts parameters.

1.6. JDBC executing queries in JDBC

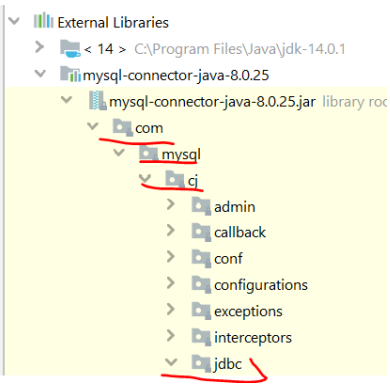
executeQuery()	executeUpdate()	execute()
This method is used to execute the SQL statements which retrieve some data from the database.	This method is used to execute the SQL statements which update or modify the database.	This method can be used for any kind of SQL statements.
This method returns a ResultSet object which contains the results returned by the query.	This method returns an int value which represents the number of rows affected by the query. This value will be the 0 for the statements which return nothing.	This method returns a boolean value. TRUE indicates that query returned a ResultSet object and FALSE indicates that query returned an int value or returned nothing.
This method is used to execute only select queries.	This method is used to execute only non-select queries.	This method can be used for both select and non-select queries.
Ex : SELECT	Ex : DML → INSERT, UPDATE and DELETE DDL → CREATE, ALTER	This method can be used for any type of SQL statements.

1.7. DEMOS

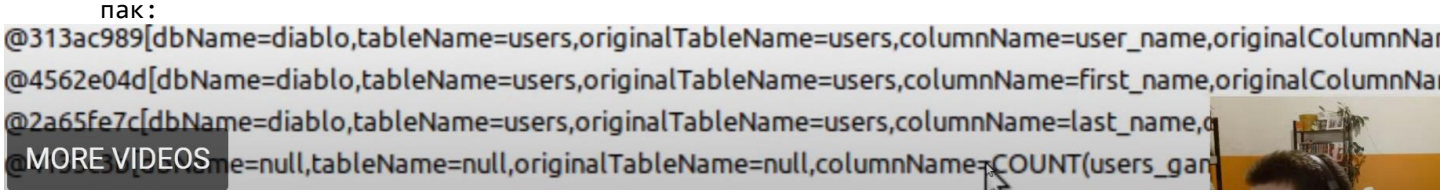
The whole Demo itself

```
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-а 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

```
Enter minimal salary (default 20 000):
| Roberto         | Tamburello      |   43300,00
| Rob             | Walters         |   29800,00
| Thierry         | D'Hers          |   25000,00
```

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-a Connection е 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";
```

Main2.java | db.properties

```
user=root
password=
salary=40000
```

```
        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-a Connection е 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

1.8. 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**

Example

Когато външен потребител зададе грешна заявка и нещата в базата могат да се объркат ако не ползваме `PreparedStatement`. При целенасочена атака, обикновено се връща GET заявка с чувствителна информация за даден потребител.
`PreparedStatement` един вид санитизира да не може да се подаде някаква различна или вложена заявка от рода на “WHERE 1 = 1” и така да се върнат всички записи например.

- `Do not build a query using the “+” operator`
- `Prefer named parameters`

```
String insertSQL = "INSERT INTO employees (first_name, last_name) VALUES(@fn,@ln)";
PreparedStatement prepStatement = conn.prepareStatement(insertSQL);
prepStatement.setString("@fn", "Martin"); //named parameter fn
prepStatement.setString(“@ln”, "Toshev"); //named parameter ln
prepStatement.executeUpdate();
```

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:

```
SELECT id FROM users
WHERE username = 'example_user' AND password = '12345';
```

- 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:

```
SELECT id FROM users
WHERE username = 'pesho'
AND password = '1' OR username = 'admin';
```

- 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.)

1.9. Retrieving database metadata

- Each RDBMS provides a way to retrieve database metadata (i.e. list of tables in the database or list of columns in table). In Oracle this is the database dictionary.
- JDBC provides a way to retrieve RDBMS metadata independent of the particular RDBMS implementation

- To retrieve metadata from the connection (e.g. user schemas, schema tables)

```
Connection dbConn = DriverManager.getConnection(.............);
DatabaseMetaData metaData = dbConn.getMetaData();
ResultSet hrmTables = metaData.getTables(null, "HRM", null, null);
```

- To retrieve metadata from the result set (e.g. table column names and types)

```
Connection dbConn = DriverManager.getConnection(.............);
DatabaseMetaData metaData = dbConn.getMetaData();
ResultSet hrmTables = metaData.getTables(null, "HRM", null, null);
ResultSetMetaData resultMetadata = hrmTables.getMetaData(); //мета данни на метаданни!!!
int columnCount = resultMetadata.getColumnCount();

ResultSet procedures = metaData.getProcedures(........);
```

ResultSetMetaData class
Used to get information about the types and properties of the columns in a ResultSet object

```
Statement stmt = conn.createStatement();
ResultSet resultSet = stmt.executeQuery("SELECT * FROM employees");
ResultSetMetaData resultSetMetaData = resultSet.getMetaData(); //мета данни на реални
данни
int columnCount = resultSetMetaData.getColumnCount();
for (int i = 0; i < columnCount; i++) {
   System.out.println(resultSetMetaData.getColumnName(i));
      System.out.println(resultSetMetaData.getColumnTypeName(i));
}
```

1.10. Handling 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();
}
```

A database transaction is a set of database operations that must be either entirely completed or aborted.
A simple transaction is usually in this form:

1. Begin the transaction
2. Execute several SQL **DML** statements
3. Commit the transaction

If one of the SQL statements fails, rollback the entire transaction.

JDBC transaction mode:

- Auto-commit by default
- Can be turned off by calling `connection.setAutoCommit(false);`

In auto-commit mode each statement is treated as a separate transaction.
If the auto-commit mode is off, no changes will be committed until the commit() is invoked.
Auto-commit mode can be turned back on by calling connection.setAutoCommit(true).

По подразбиране auto commit е true. Ако искаме да използваме трансакции, то се налага да сменим тази настройка!

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. JDBC Advanced features

Connection pool

Кънекциите към базата са скъпи обекти/ изискват ресурси. Затова можем да използваме connection pool.
Connection pool contains a number of open database connections.
There are a few choices when using connection pool:

1. Depends on application server - т.е. когато не използваме директно JDBC за връзка с базата данни, а връзката се менъджира от т.наречения application server, който използва pools
2. Use JDBC 2.0 interfaces (`javax.sql.ConnectionPoolDataSource; javax.sql.PooledConnection;`)
3. Create your own connection pool

Transaction isolation level

You can set the transaction isolation level by calling:
`dbConn.setTransactionIsolation(level);`

level – one of the following Connection constants:
Connection.TRANSACTION_READ_UNCOMMITTED,
Connection.TRANSACTION_READ_COMMITTED,
Connection.TRANSACTION_REPEATABLE_READ,
or Connection.TRANSACTION_SERIALIZABLE.
(Note that Connection.TRANSACTION_NONE cannot be used because it specifies that transactions are not supported.)

До каква степен трансакциите виждат промяната от текущата трансакция. Аналогия може да се каже - синхронизация на трансакциите.

TRANSACTION_SERIALIZABLE е с най-високо ниво на изолация - невлияещо се от нито една друга трансакция докато не приключи текущата трансакция. Но пък и е по-бавно това ниво.
Нивото зависи и от конкретната база данни, която използваме.

Transaction Level	Permitted Phenomena			Impact
	Dirty Reads	Non-Repeatable Reads	Phantom Reads	
TRANSACTION_NONE	-	-	-	FASTEST
TRANSACTION_READ_UNCOMMITED	YES	YES	YES	FASTEST
TRANSACTION_READ_COMMITED	NO	YES	YES	FAST
TRANSACTION_REPEATABLE_READ	NO	NO	YES	MEDIU
TRANSACTION_SERIALIZABLE	NO	NO	NO	SLOW

1.12. 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();
```

1.13 JDBC best practices

Close your connection

- Closing connections, statements and result sets explicitly allows garbage collector to recollect memory and resources as early as possible
- Close statement object as soon as you finish working with them
- Use **try-finally** statement to guarantee that resources will be freed even in case of exception
- Possible to be managed also via the connection pool

Use proper statements

- Use *PreparedStatement* when you execute the same statement more than once
- Use *CallableStatement* when you want to result from multiple and complex statements for a single request

Batch your queries

- Send multiple queries to reduce the number of JDBC calls and improve performance - изпращаме накуп няколко заявки

```
connection.setAutoCommit(false);

Statement stmt = conn.createStatement();
stmt.addBatch("sql_query1");
stmt.addBatch("sql_query2");
stmt.addBatch("sql_query3");
stmt.executeBatch();
```

- **You can improve performance by increasing number of rows to be fetched at a time**

```
Statement stmt = conn.createStatement();
stmt.setFetchSize(30);
```

- **Get only necessary columns, not all columns of the entity.**

```
Statement stmt = conn.createStatement();
BAD
ResultSet rs = stmt.executeQuery("SELECT * FROM employees WHERE ID=1");

GOOD
ResultSet rs = stmt.executeQuery("SELECT NAME, SALARY FROM employees WHERE ID=1");
```

1.14. Demo with batches

```
public class App {
   public static void main(String[] args) {
        System.out.println("Hello World!");
        java.sql.Connection connection = null;
        try {
            connection =
java.sql.DriverManager.getConnection("jdbc:h2:file:./sample;DB_CLOSE_DELAY=-
1;AUTO_SERVER=TRUE", "", "");
//            connection = java.sql.DriverManager.getConnection("jdbc:h2:mem:sample");
            createLogsTable(connection, 10);
            System.out.println(countLogs(connection));
        } catch (Exception ex) {
//            if (connection != null) {
//                jdbc.test.JDBCUtil.closeConnection(connection);
//            }
            System.out.println("I AM EXCEPTION: " + ex.getMessage());
        }
   }

   public static void createLogsTable(java.sql.Connection connection, int entriesCount)
throws java.sql.SQLException {
        connection.setAutoCommit(false);

        java.sql.Statement stmt = connection.createStatement();
        stmt.executeUpdate("CREATE TABLE Logs(Id INT, Message VARCHAR(1000))");
```

//batches and PreparedStatement

```
        java.sql.PreparedStatement prepStatement = connection.prepareStatement("INSERT
INTO Logs(Id, Message) VALUES(?,?)");
        for (int i = 1; i <= entriesCount; i++) {
            prepStatement.setInt(1, i);
            prepStatement.setString(2, "");
            prepStatement.addBatch();
        }
        prepStatement.executeBatch();

        connection.commit();
```

```
    }

    public static void createLogsTable(java.sql.Connection connection, int entriesCount)
throws java.sql.SQLException {
        java.sql.Statement stmt = connection.createStatement();
        stmt.executeUpdate("CREATE TABLE Logs(Id INT, Message VARCHAR(1000))");

        String insertSQLString = "INSERT INTO Logs(Id, Message) VALUES(?, ?)";
        java.sql.PreparedStatement prepStatement =
connection.prepareStatement(insertSQLString);
        for (int i = 1; i <= entriesCount; i++) {
            prepStatement.setInt(1, i);
            prepStatement.setString(2, "");
            prepStatement.executeUpdate();
            prepStatement.clearParameters();
        }
    }

    public static void createLogsTable(java.sql.Connection connection, int entriesCount)
throws java.sql.SQLException {
        connection.setAutoCommit(false);

        java.sql.Statement stmt = connection.createStatement();
        stmt.addBatch("CREATE TABLE Logs(Id INTEGER, Message VARCHAR(1000))");
```

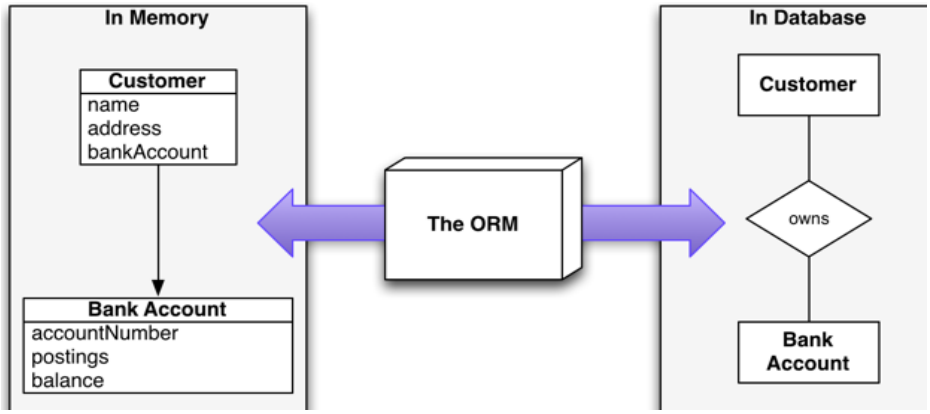
//batches and Statement

```
        String insertSQLString = "INSERT INTO Logs(Id, Message) VALUES(%d, '')";
        for (int i = 1; i <= entriesCount; i++) {
            stmt.addBatch(insertSQLString.formatted(i));
        }
        stmt.executeBatch();

        connection.commit();
        connection.setAutoCommit(true);
    }


public static void createLogsTable(java.sql.Connection connection, int entriesCount)
{
    try {
        java.sql.Statement stmt = connection.createStatement();
        stmt.executeUpdate("CREATE TABLE IF NOT EXISTS Logs(Id INTEGER, Message
VARCHAR(1000))");

        stmt.executeUpdate("DELETE FROM Logs");

        String insertSQLString = "INSERT INTO Logs(Id, Message) VALUES(%d, '')";
        for (int i = 1; i <= entriesCount; i++) {
            stmt.executeUpdate(insertSQLString.formatted(i));
        }
    } catch (java.sql.SQLException e) {

    }
}

    public static int countLogs(java.sql.Connection connection) throws
java.sql.SQLException {
        java.sql.Statement stmt = connection.createStatement();
```

```
        java.sql.ResultSet rs = stmt.executeQuery("SELECT count(*) FROM Logs");
        rs.next();
        return rs.getInt(1);
    }
}
```

2. ORM Fundamentals

2.1. ORM Introduction - Object-Relational Mapping



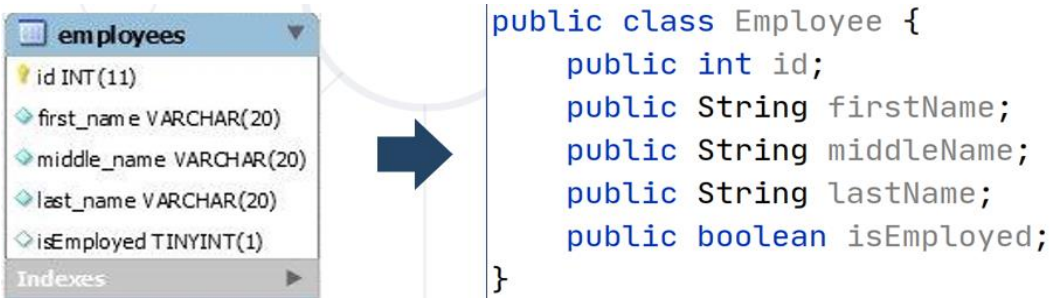
ORM (Object relational mapping) Frameworks Overview

- 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)...

What is ORM?

- **Technique** for **converting data** between incompatible 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**



- In relational databases, business entities are represented as tables
- In object-oriented languages, business entities are represented as classes
- Object relational mapping frameworks are used for moving business entities from one medium to the other

Key points of an ORM:

- Object persistence
  - Create, Retrieve, Update, Delete (CRUD)
  - Find (<criteria>)
- Session management – when connecting to a database, we are in a session
- Transaction management
  - Automatic / implicit transactions
  - Concurrency control
- Lazy loading
- Soft deletion
- Data versioning – версии на редовете записи в таблиците
- Data caching – ORM is caching data
- Data validation
  - Content
  - Security
- Audit logging
- Cascade delete / update / create
- Entity inheritance

- In Java SE / Java EE (Jakarta EE) the JPA (Java Persistence API) is a programming interface specification for working with relational databases from Java code
  - Entities are Java POJOs (Plain Old Java Objects) – simple Java classes that correspond to relational tables
  - **JPQL** is the query language used to perform queries against the database in Java **by means of the entities** (JPQL is the equivalent of SQL) – работим с ентититата на нашето приложение (т.е. с ентити класовете)

- The **JPA specification is derived** from features used already in existing ORM frameworks such as Hibernate and Eclipse TopLink. I.e. Hibernate existed before JPA specification.
- **JPA** is supported in latest versions of most widely used ORM libraries (such as Hibernate and Eclipse TopLink)

Why do we need ORM?

- In OOP, data-management tasks act on **objects** that are almost always **non-scalar** values

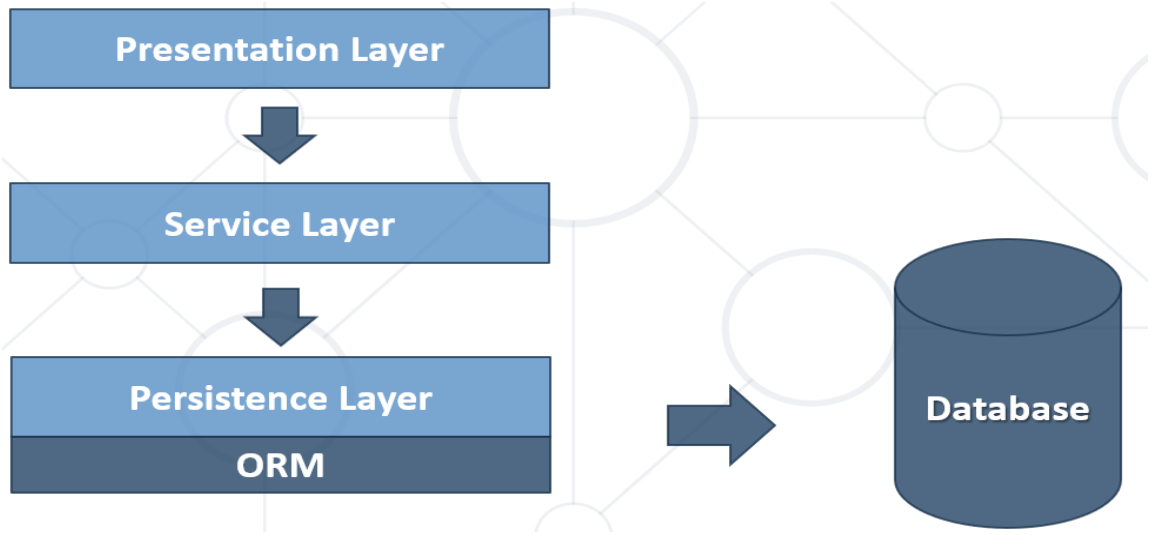
- Many **database** can only store and manipulate **scalar** values, organized within **tables – scalar са обикновените данни, а non-scalar са релационни данни примерно**
- We must **manually** convert values into groups of simpler values to store in DB and convert them back when we retrieve data

- Benefits in using ORM:
  - Increased developer productivity – use objects with associations instead of tables and SQL
  - Portability – database vendor independence
  - Abstraction – the relational database is represented as **Java object model**
    - Complexity hidden within ORM
    - Developer works with objects (however possibility to also work with SQL queries)
  - Fewer bugs
    - Less code
    - Code reuse and code generation
  - Improved design
    - Decoupling / separation of concerns

JDBC vs. 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



DB First model vs. Code First model

- **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**)

Code First model preferred!!!

- **Models** the database after the entity classes are created (класове в java) -> които да бъдат съхранявани след това в база данни. Логиката да е същата.

Perform data operations with ORM

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

- Save entity to DB

```
Student student = new
Student('George', 'Brown');
session.save(student);
```

```
INSERT INTO students
(firstName, lastName)
VALUES
('George', 'Brown')
```

- Retrieve data from DB

```
Student student = (Student)
session.get(Student.class, 1);
```

```
SELECT * FROM students
WHERE id=1;
```

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

Hybernate QueryLanguage = HQL

- Using HQL

```
List<Student> studentList =
session.createQuery("FROM Student").toList();
```

- Using SQL

```
String sql = "SELECT * FROM Employee";
SQLQuery query = session.createSQLQuery(sql);
query.addEntity(Employee.class);
List<Employee> results = query.list();
```

Как става в mapping-а - 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;
}
```

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**

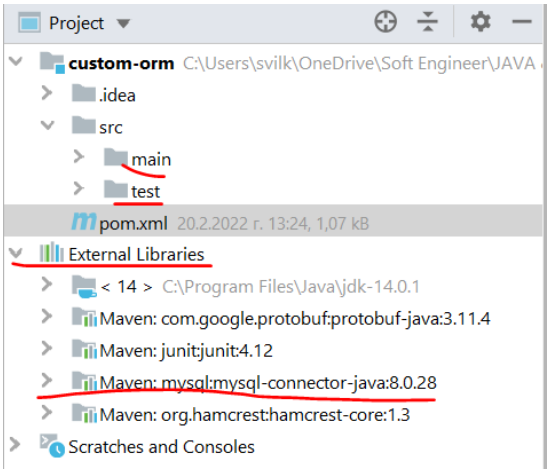
- What the code is actually doing - the developer is more in control using SQL

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

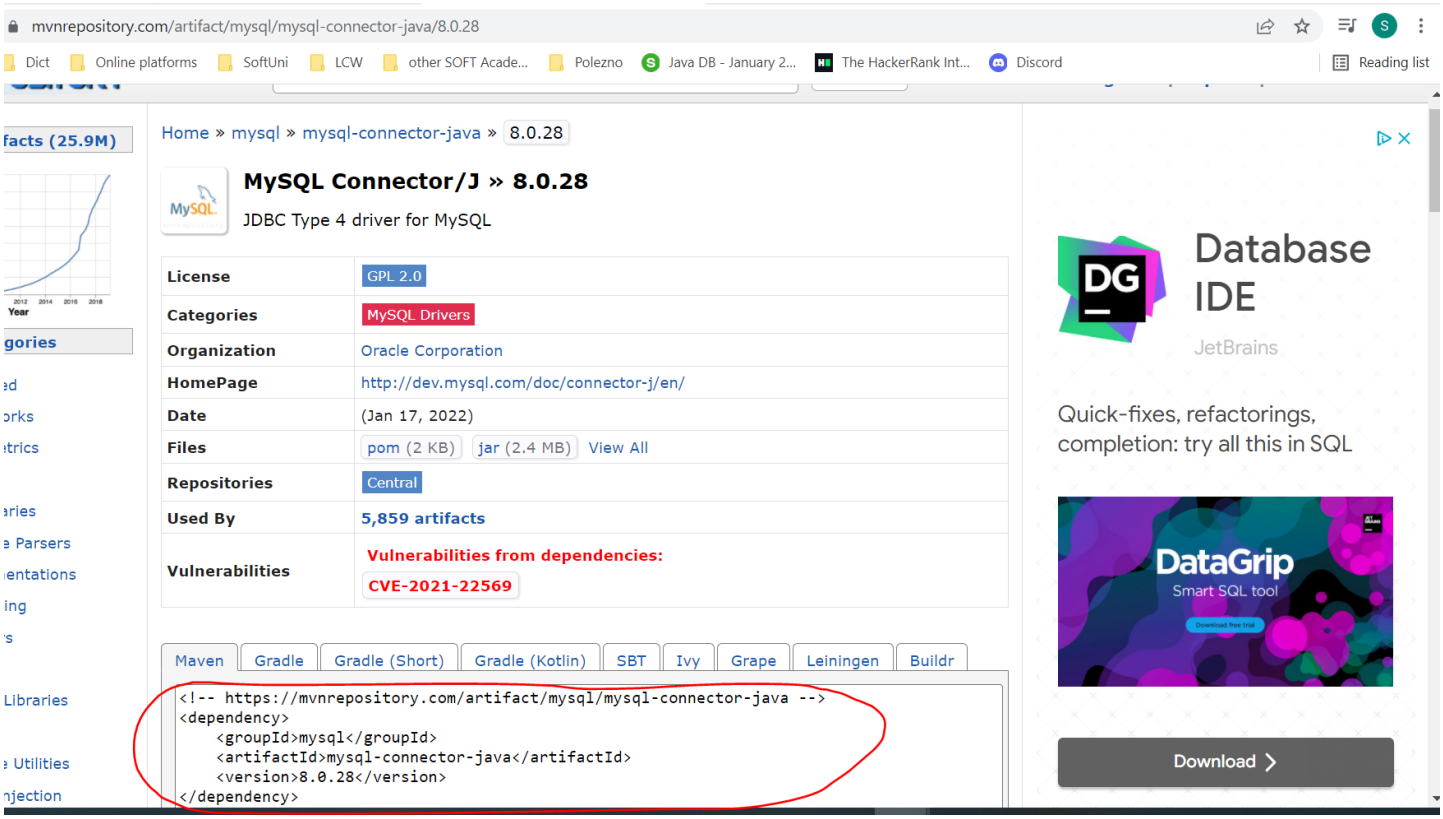
Имената на елементите на анотациите са имената на променливите в базата данни!!!
Следната задача не е рефактурирана с цел – че няма да мога да се ориентирам ако изнесем в много методи

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 подразбиране 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 – грешно, трябва да използваме анотацията
@Table за да сменим името на таблицата в базата @Table(name = " users")
public class User {
    @Id
    @Column(name = "id")
    private long id;
```

```
    @Column(name = "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 addresses - грешно, трябва да използваме
анотацията @Table за да сменим името на таблицата в базата @Table(name = "addresses")
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 е инстанция на table
            //declared field е едно от полетата на table
```

```
            //fieldName е 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(); //мези
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

3.1. Info/ Intro

https://hibernate.org/orm/documentation/6.4/

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!!!

- Hibernate is an ORM library for Java (**since 3.2 provides implementation of JPA**)
    - Entities are Java POJOs (Plain Old Java Objects) – simple Java classes that correspond to relational tables
    - **HQL** is the query language used to perform queries against the database in Java by means of the entities (**HQL** is equivalent to SQL) - работим с ентититата на нашето приложение (т.е. с ентити класовете)
    - In Hibernate only, **Criteria queries are** an object-oriented **alternative to HQL**

- Both JPA and Hibernate support configuration via two different mechanisms:
    - XML files
    - Java annotations

- Hibernate е разработван под шапката на RedHat

3.2. Hibernate features

Same features as the ORM that we already mentioned in previous chapter.

- Open-source, free (LGPL license)
- Based on modern OOP methodologies
- Transparently persists and retrieves POJO objects
- Stable, well established product
- Maps Java POJO classes to DB tables
- Maps Java data types to SQL data types
- Provides powerful data query and retrieval facilities:
    - HQL queries
    - Criteria queries
    - and pure SQL queries – bypassing the ORM framework in a way/ better not to use them
- Lazy loading
- Powerful caching mechanism
- Session management – **session is 1st level cache**
- Transaction management
- Removes the need to mention all these references to database connections, JDBC, SQL, etc.
- **Makes your code database independent!!!**
- Makes your business objects cleaner
- Mappings
    - Table row maps to an object automatically
    - Set of rows maps to a collection automatically

- Concurrency management – възможност да реализираме многонишков аспект/връзка с базата данни
    - **Да избягваме да ползваме и да споделяме една и съща Hibernate сесия от много нишки, а да ползва само 1 нишка за сесията!!! По-скоро да си създаваме отделни Hibernate сесии за отделните нишки/операции.**

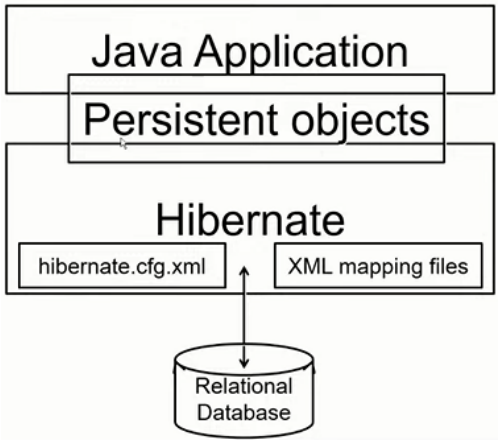
- Automatically traces and detects changed objects when saving or updating DB
- Transactional write-behind
    - No database access until needed – this allows more efficiency in queries, etc.
    - Keeps transactions much smaller in time/scope – avoids long transactions models

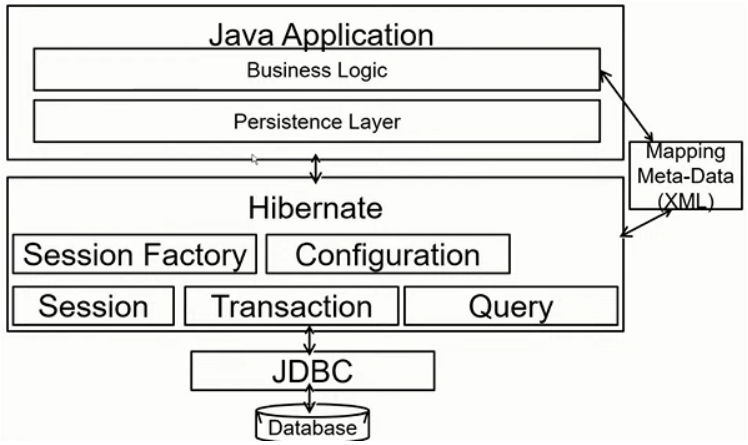
- Simplified navigation model
    - customer.getOrders() instead of SQL joins
    - navigate through child rows as nested collection of objects
- Code in your application is simpler

```
for (Order order : customer.getOrders()) {
      if (order.getItems().size() > 2) { … }
}
```

- Hibernate supports all popular databases:
  - Oracle
  - Microsoft SQL Server
  - DB2
  - PostgreSQL
  - MySQL
  - Interbase / Firebird
  - Informix
  - others

3.3. Hibernate Architecture





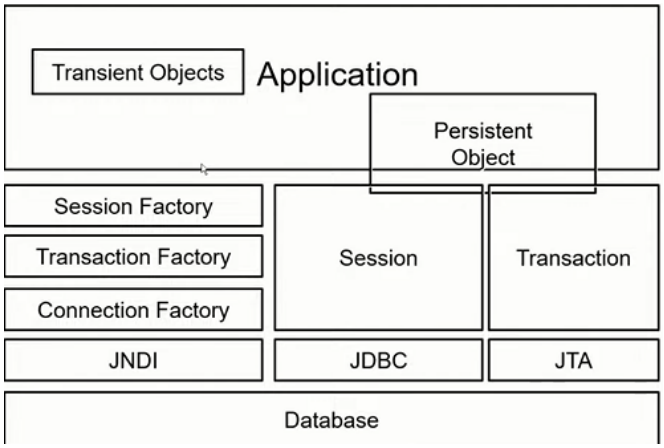
Ако използваме чистия Hibernate, то имаме класа **org.hibernate.**SessionFactory от <artifactId>hibernate-core.

Ако използваме JPA Hibernate, тогава имаме класа **javax.persistence.**EntityManager от <artifactId>javax.persistence-api.

JDBC – връзката към базата винаги се реализира под капака на Hibernate посредством JDBC

JNDI – Java Naming and Directory Interface used in JavaEE
В този сървис JNDI можем да декларираме например имплементация на ConnectionFactory, от което да създадем TransactionFactory и в последствие да имаме SessionFactory.

JTA – Java Transaction API – дава възможност да дефинираме трансакции в рамките на нашето приложение посредством анотацията @Transactional върху конкретен метод съдържащ в себе си много на брой операции, и без да използваме горните варианти/опции (org.hibernate.SessionFactory или javax.persistence.EntityManager).
QUARKUS фреймурка по дефаулт е в режим и на JTA винаги.
И съответно, ако нещо се прецака, да се rollback-нат всичките въпросни операции.



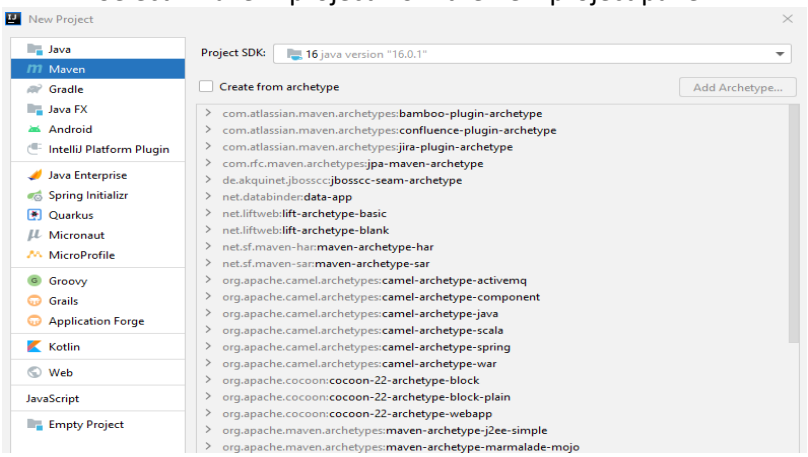
3.4. Maven - Project Management and Comprehension

Maven Overview

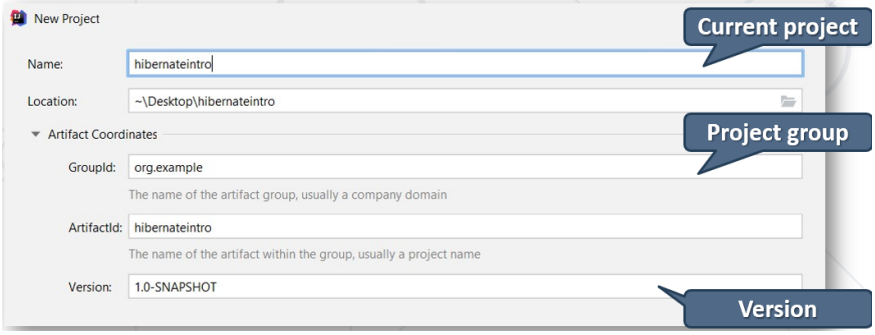
- Maven is a built automation tool.
  - Describes how software is built and its dependencies
  - Uses XML files
- Dynamically downloads **Java libraries** and **Maven plug-ins**
  - Projects are configured using a **P**roject **O**bject **M**odel, which is stored in a **pom.xml** file

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
```

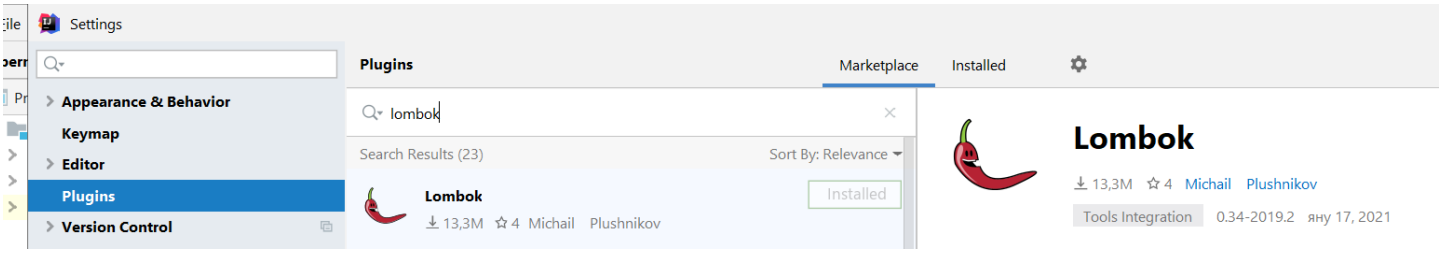
Properties обобщават долуизброените, и ако се наложи да се променя на много места, то можем да направим грешка. А тук само на едно място променяме версията

```
<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>
```

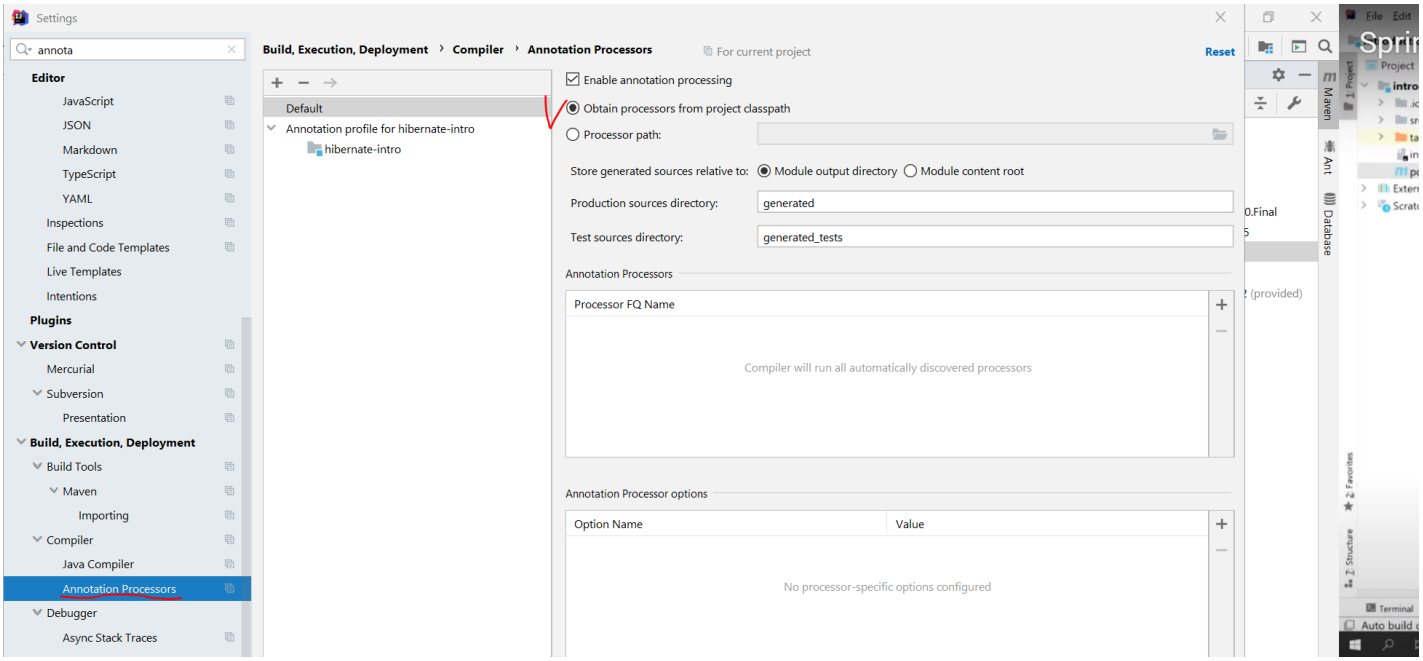
```
//това ни добавя driver за mysql конектора
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>${msql.version}</version>
</dependency>


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>
```

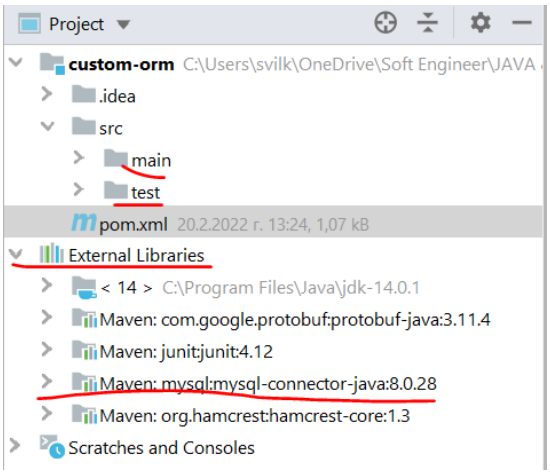
//Използване на lombok

```
//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>
```

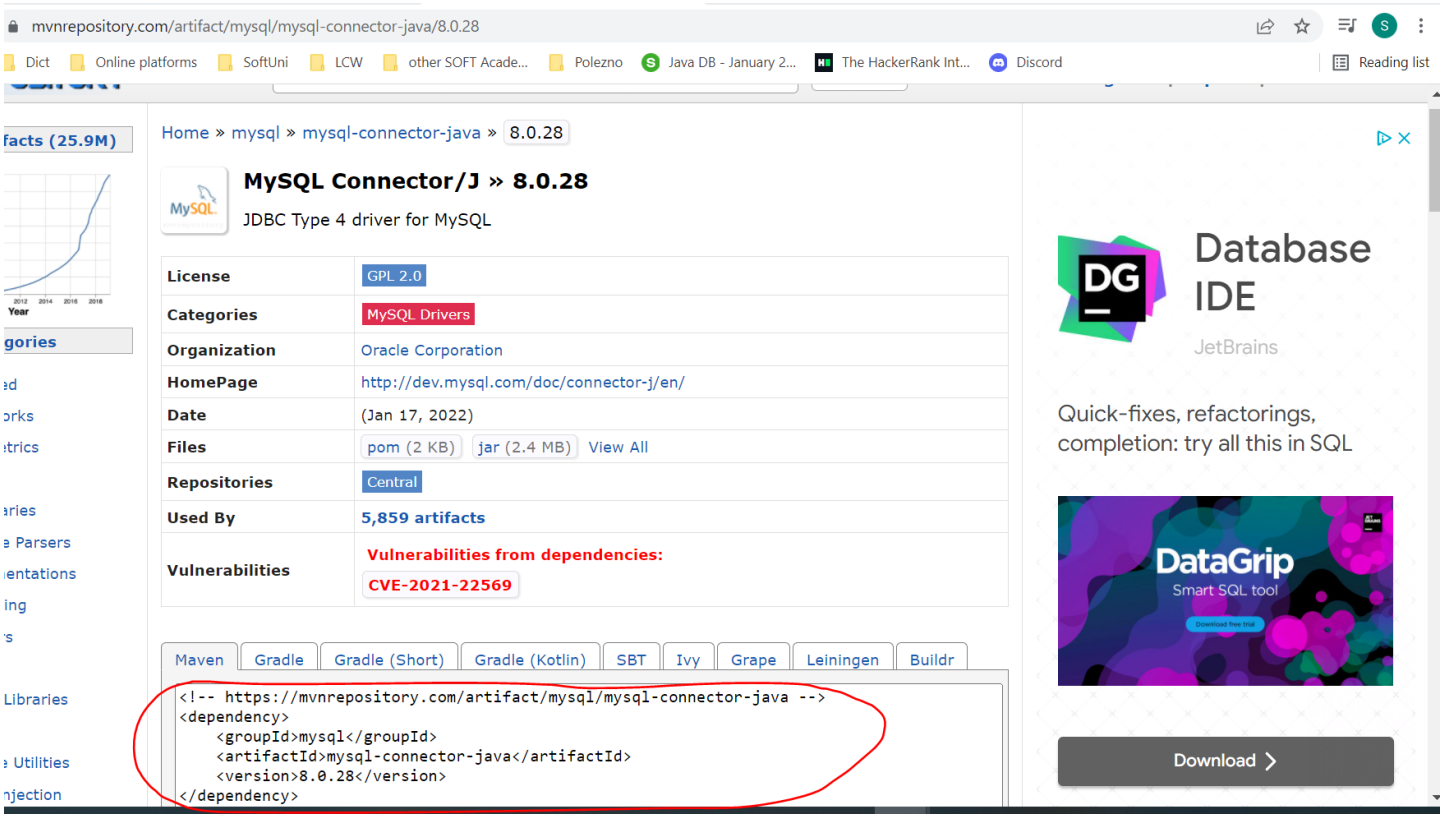


.....





Можем да заредим dependency и чрез сайта mvnrepository.com/
https://mvnrepository.com/artifact/mysql/mysql-connector-java



3.5. 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

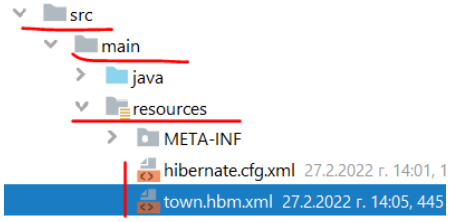
1) Classical Hibernate Native XML mappings using POJO (Plain Old Java Objects Without annotations)

Entity Class: Student

Hibernate generates proxies for entity classes so that child entities can be retrieved lazily when requested from the parent entity.

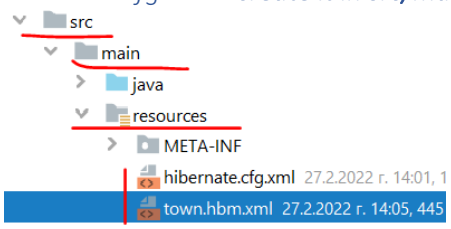
```
public class Student {

    private long id;

    private String name;

    // Getters and setters //мук си създаваме конструктори/getters/setters ръчно
}
```

*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, след това мапинг тип мапващ от Java към SQL


    </class>
</hibernate-mapping>
```

*hibernate.cfg.xml – **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 е 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

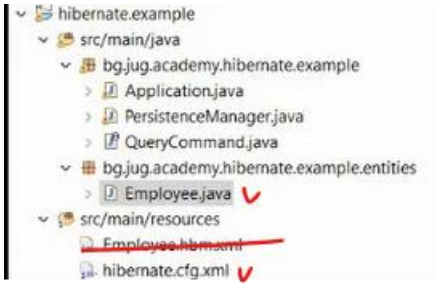
```
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;

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();
    }
}
```

Добре е да го сложим в try-catch-finally блок, и да rollback-нем ако нещо гръмне!!!

```
Session session = factory.openSession();
Transaction tx = null;
try {
   tx = session.beginTransaction();
   // do some work
   ...
   tx.commit();
}
catch (Exception e) {
   if (tx!=null) tx.rollback();
   e.printStackTrace();
}finally {
   session.close();
}
```

2) Hibernate Native Framework (before JPA) – using annotations (annotated POJO)

*hibernate.cfg.xml – **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  -->
        <mapping class="org.javaknights.hibernate.example.entities.Employee"/>
//mapping with annotations!!!
    </session-factory>
</hibernate-configuration>
```

3) JPA Annotated Java classes (annotated POJO) mapped to DB tables

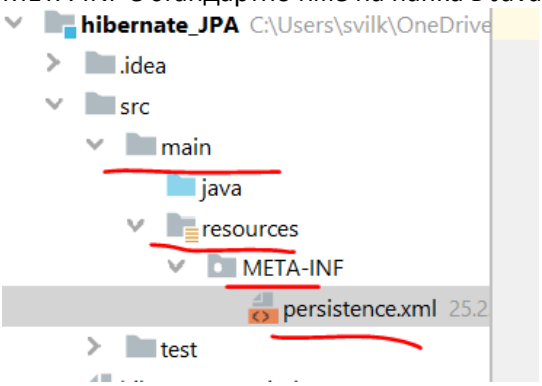
Entity Class: Student

Hibernate generates proxies for entity classes so that child entities can be retrieved lazily when requested from the parent entity.

```
@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
//Като я сложим тази анотация @GeneratedValue, това означава че id-то  се генерира от самата
база данни
    @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();
    }
}
```

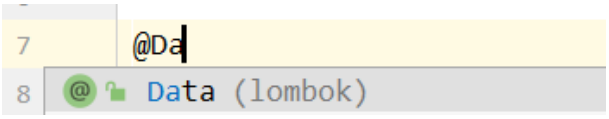
3.6. Hibernate Native Framework (before JPA) - Classical Hibernate XML mappings using POJO

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

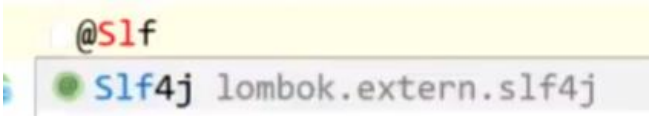
Използване на Lombok

```
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                //нтрави полето name като required argument на конструктора
    private String name;

    private Date registrationDate = new Date();
}
```

logger на Lombok - @Slf4j




```
@Override
public void run(String... args) throws Exception {
    SAMPLE_USERS.forEach(user -> this.userService.addUser(user));
    log.info("Created users: {}", userService.getUsers());
}
```

Exclude - If present, do not include this field in the generated toString.

```
@OneToMany(mappedBy = "author")
@ToString.Exclude
private Collection<Post> author = new ArrayList<Post>();
```

Hibernate Configuration

Entity Class: Student

Hibernate generates proxies for entity classes so that child entities can be retrieved lazily when requested from the parent entity.

```
public class Student {

    private long id;

    private String name;

    // Getters and setters //тук си създаваме конструктори/getters/setters ръчно
}
```

pom.xml

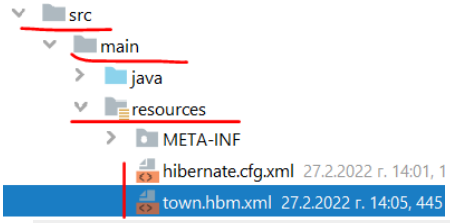
//това ни добавя hibernate-core

```
        <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.xml – **create it in src/main/resources***

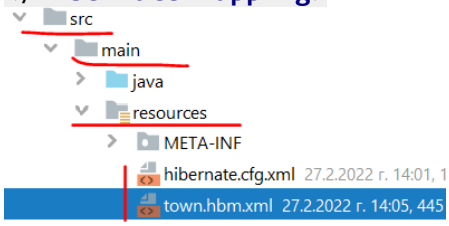



```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration                              //Configuration
        PUBLIC "-//Hibernate/Hibernate Configuration DTD//EN"
        "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">

<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();
    }
}
```

Добре е да го сложим в try-catch-finally блок, и да rollback-нем ако нещо гръмне!!!

```
Session session = factory.openSession();
Transaction tx = null;
try {
   tx = session.beginTransaction();
   // do some work
   ...
   tx.commit();
}
catch (Exception e) {
   if (tx!=null) tx.rollback();
   e.printStackTrace();
}finally {
   session.close();
}
```

Working with persistent objects

Info

- Changes to the entities (create, modify, delete, …) are not executed immediately
- Hibernate holds them in a list and executes them at once in a sequence
- To force posting pending changes, use: **session.flush();**
- Does not commit the active transaction – flushing does not commit the active transaction

- Sometimes entities or their associated entities hold old data and need to be refreshed (reloaded from the database)
- To refresh a persistent entity object, use: **session.refresh(object);**
- Use case:
    - We load an entity
    - Meanwhile another transaction updates it
    - We need to refresh

- To modify an entity:
    1. Obtain persistent entity object by **session.get()** or **Query.list()**
    2. Modify the persistent object
    3. Invoke **session.update(object);**

- Suppose we get some object from the database and close the Hibernate session:
    1. This object becomes detached
    2. If we try to get its child entities an exception will be thrown

```
Course course = (Course)
session.get(Course.class, 5L);
session.close();
List<Student> students = course.getStudents();
// org.hibernate.LazyInitializationException:
// no session or session was closed
```

Detached objects can be attached again with **session.buildLockRequest(LockMode.NONE).lock(object);**

```
Course course = (Course)
session.get(Course.class, 5);
session.close();
// Now the course object is "detached"
session = sessionFactory.getCurrentSession();
session.beginTransaction();
session.buildLockRequest(LockMode.NONE).lock(course);
// Attach "course" object to the new session
List<Student> students = course.getStudents();
```

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 Native Queries

- Native queries are just plain SQL queries written using the Hibernate API
- The results are returned as Object[]

```
private static List<Object[]>
getStudentsNames(Session hbSession) {
      SQLQuery studentsNamesQuery =
      hbSession.createSQLQuery(
      "select p.firstName, p.lastName " +
      "from students s inner join persons p
" + "on s.studentId = p.personId");
List<Object[]> studentsNames =
      studentsNamesQuery.list();
      return studentsNames;
}
```

- SQL queries can return results transformed to an arbitrary JavaBean class:

```
SQLQuery studentsNamesQuery =
   session.createSQLQuery(
      "select p.firstName as \"firstName\", " +
      "p.lastName as \"lastName\" " +
      "from students s inner join persons p " +
      "on s.studentId = p.personId");
studentsNamesQuery.setResultTransformer(
   Transformers.aliasToBean(NameBean.class));
List<NameBean> names = studentsNamesQuery.list();
```

Hibernate Retrieve Data by Get

```
session.get(<entity class>, <primary key>);
```

```
   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.

- HQL queries work on entities and their properties and is object-oriented (meaning it can handle object-oriented features of the Hibernate entities).
- Entity names and properties are case sensitive in HQL – everything else is case insensitive as in SQL
- In HQL you can omit a SELECT clause

```
from bg.jug.academy.hrm.entities.Employee
```

- In HQL you can omit the package of the entity

```
from Employee
```

- You can use aliases:

```
from Employee emp
```

```
from Employee as emp
```

- In HQL there is also a where clause

```
from Employee emp where emp.name like 'Ivan%';
```

- You can also use a SELECT clause

```
SELECT emp.name FROM Employee emp
WHERE emp.name like 'Ivan%';
```

- You can use an ORDER BY clause:

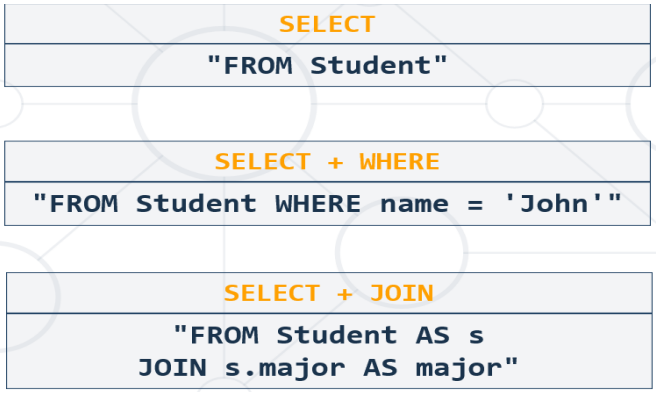
```
from Employee emp ORDER BY emp.name DESC;
```

- You can have nested queries

```
from Employee e
where e.salary = (select max(salary) from Employee);
```

- Navigating through many-to-one associations is performed directly

```
from Customer as cust
where cust.address.town = 'Sofia';
```



- HQL queries can have named and unnamed parameters
  - Unnamed parameters

```
from Student stud
where stud.lastName = ?
```

  - named parameters

```
from Student stud
where stud.firstName = :first_name
and stud.lastName = :last_name
and stud.courses.size = :count_of_courses
```

```
Query queryStudentByName = hbSession.createQuery(
      "from Student " +
      "where firstName = :fname " +
      "and lastName = :lname");
      queryStudentByName.setString("fname", "Ivan");
      queryStudentByName.setString("lname",
      "Ivanov");
List<Student> students = queryStudentByName.list();
if (students.size() == 0) {
      System.out.println("Student not found");
} else if (students.size() == 1) {
      Student stud = students.get(0);
      System.out.println("Id=" +
      stud.getPersonId());
} else {
      System.out.println("Too many students found");
}
```

- HQL queries are polymorphic (meaning that when a parent entity is retrieved, then child entities are also retrieved)
- Polymorphic behavior can be controlled via the Hibernate configuration
- When results are not entities, then the returned values are stored as Object[]

```
Query deptsStudsQuery = session.createQuery(
      "select dept.deptId, dept.name, count(*) " +
      "from Department dept join dept.Courses c " +
      "group by dept.deptId, dept.name");
List<Object[]> deptsAndStuds =
deptsStudsQuery.list();
```

- Several ways to process Query results:
  - list() – returns the results as a java.util.List instance
  - iterate() – returns the results as a java.util.iterator instance

```
Query studQuery =
      session.createQuery("from Student");
      Iterator<Student> studIter = studQuery.iterate();
      while (studIter.hasNext()) {
         Student stud = studIter.next();
         System.out.printf("FirstName=%s, LastName=%s",
            stud.getFirstName(), stud.getLastName());
}
```

  - uniqueResult() – **returns single object**, null or NonUniqueResultException

- You can specify additional join conditions using the with keyword:

```
from Employee as emp left join emp.skills as skill
with skill.name like '%Java%'
```

- You can also retrieve child records from joined entities using the fetch keyword:

```
from Cat as cat
inner join fetch cat.mate
left join fetch cat.kittens
```

- You can also create a new bean from an HQL query assuming it has appropriate constructor

```
select new EmployeeBean(emp.name, emp.phone)
from Employee emp
```

- HQL queries can even return the results of aggregate functions (such as sum, max, count and avg) on properties

- You can use indices to refer to particular elements – indexes can only be used in a WHERE clause

```
from Employee emp where emp.skills[0].name like
'%Java%'
```

Criteria API

- The Criteria API provides a mechanism for specifying WHERE conditions, joins, ordering, and basically most of the things supported in HQL (but not all of them)
- Criteria queries can also be generated at runtime

Examples:

```
List<Course> coursesByCriteria =
      session.createCriteria(Course.class)
      .add(Restrictions.like("name", "%Java%"))
      .list();
```

```
List<Course> coursesByCriteria =
      session.createCriteria(Course.class)
      .add(Restrictions.in("name",
      new String[] { "Java", "C#", "C++" } ) )
      .createAlias( "professor", "prof" )
      .add(Restrictions.eq("prof.lastName","Sali" ) )
      .addOrder( Order.asc("name") )
      .list();
```

```
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();
}
```

Handling transactions

- **Optimistic concurrency control** means that conflicts are resolved **at the end of a transaction** (optimistically) – supported in Hibernate via **automatic versioning** – таблицата има колона версия, и то е ясно ако се мъчи да запише и види, че версията на даден ред запис е вдигната, то да не записва
- **Pessimistic concurrency control** means that conflicts are resolved during transaction execution **via database locks** (pessimistically) – supported in Hibernate by means of the locking mechanism in the underlying database
    - Hibernate does not lock objects in memory – locking semantics are defined by RDBMS by means of the supported transaction isolation levels in case of pessimistic locking

- In addition to versioning for automatic concurrency control, Hibernate provides an API for pessimistic locking of rows via SELECT FOR UPDATE syntax – заключва определни записи/редове и по този начин конкурентна трансакция не може да пипа/променя по реда.
- It is possible to implement transactions in the application by using **explicit locks** on the database objects being manipulated – however this is a bad practice since the application does not scale to many users
- When multiple users access the database concurrently, it is recommended to use Hibernate's built-in utilities for versioning in order to use **optimistic locking** for the entities

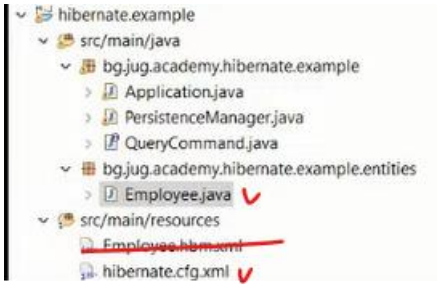
- Object identity != Entity identity отговаря на primary key от базата данни – гарантира се еднаквост само в рамките на дадена Hibernate сесия!!! Защото ако имаме 2 сесии – то се създават/взема/гетва данните за същия обект, но в рамките на двете различни сесии се създават нови/различни Java обекти
- In a single Hibernate session object identity is guaranteed to be equal to entity identity
- In concurrent sessions, object identities are different but entity identities for the same database record are equivalent

- Transactions can be commited/roll-backed explicitly when using Hibernate in a non-managed environment
- Transactions can be commited/roll-backed by the application container in a managed environment (such as a j2ee(JakartaEE) application server) when using Hibernate with JTA mode - (either with bean-managed transaction or with container-managed transaction)

- Hibernate wraps any underlying JDBC exceptions (such as an SQLException) that can be thrown in a HibernateException instance
- In case an exception occurs, then:
    - In a non-managed environment the application developer must ensure that the Hibernate session is closed
    - In a managed environment, the application developer may not need to explicitly commit/rollback the transaction or close the session when an exception is thrown – this is typically done by the container

3.7. Hibernate Native Framework (before JPA) – using annotations (annotated POJO)

*hibernate.cfg.xml – **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  -->
        <mapping class="org.javaknights.hibernate.example.entities.Employee"/>   //mapping with
annotations!!!
    </session-factory>
</hibernate-configuration>
```

3.8. 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 entity is 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

Hibernate generates proxies for entity classes so that child entities can be retrieved lazily when requested from the parent entity.

```
@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 id-то  се генерира от самата база данни
    @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

Hibernate generates proxies for entity classes so that child entities can be retrieved lazily when requested from the parent entity.

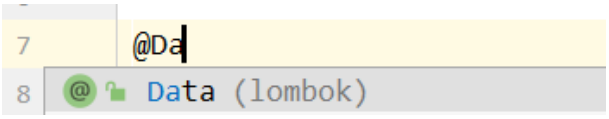
```
import javax.persistence.*;
import java.math.BigDecimal;
import java.time.LocalDateTime;
import java.util.Set;


//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

Hibernate generates proxies for entity classes so that child entities can be retrieved lazily when requested from the parent entity.

```
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, id-то  се генерира от
самата база данни
    @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

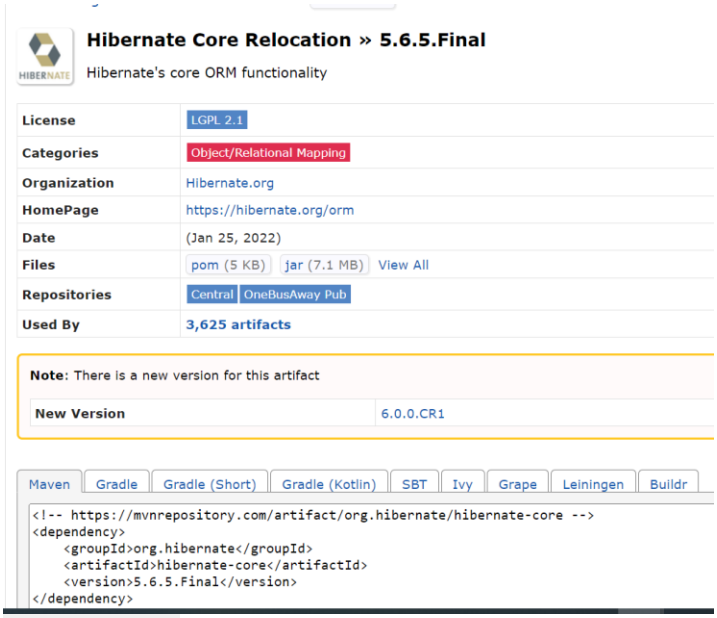
- **@Id** - Specifies the property, use for identity (primary key of a table) of the class
  - **@GeneratedValue** - specifies how the identity attribute can be initialized
    - Automatic, manual, or value taken from a sequence table
- **@Column** -Specifies the column attribute for the persistence property – без тази анотация взема автоматично името на полето. Ако я има – може да сложим специфично име на колоната в таблицата от базата данни.

JPA Configuration in Maven, in the pom.xml файла

pom.xml файла

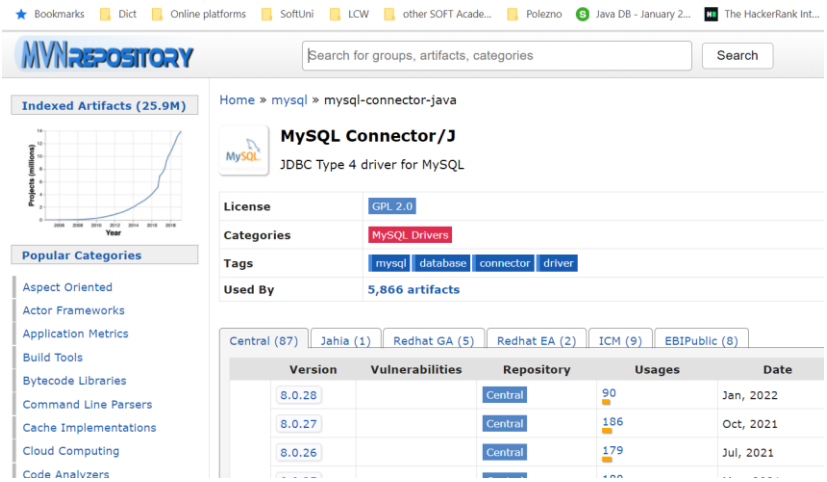
```
<dependency>
    <groupId>javax.persistence</groupId>
    <artifactId>javax.persistence-api</artifactId>
    <version>2.2</version>
</dependency>
```

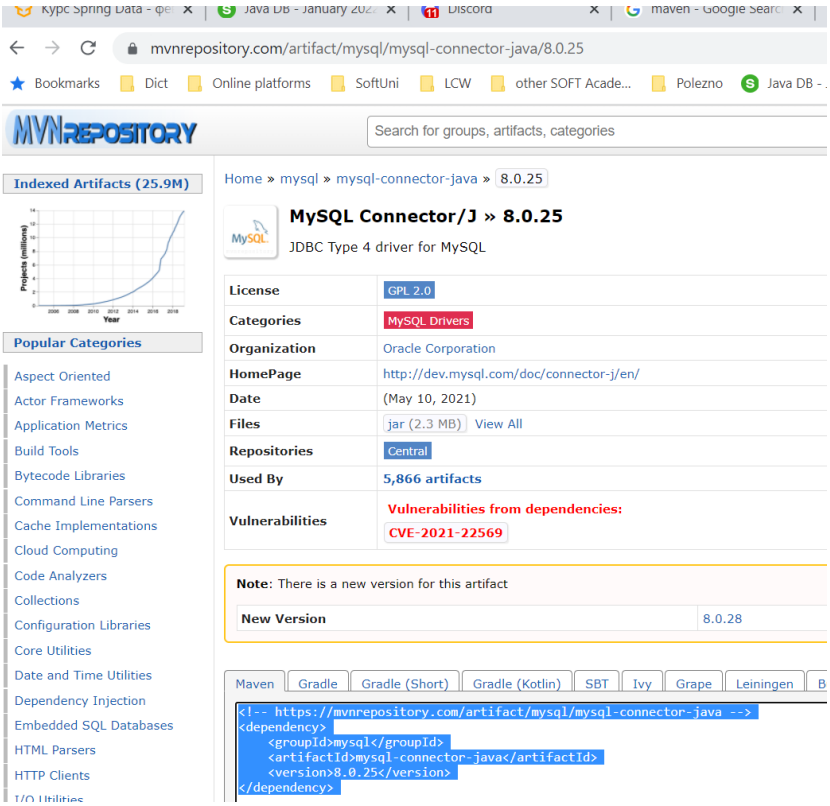
hibernate-core




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

mysql-connector-java

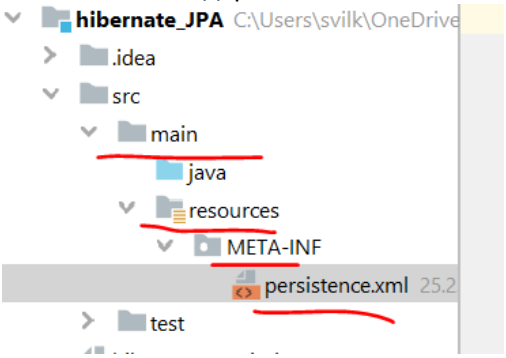





```xml
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>8.0.25</version>
</dependency>
```

*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



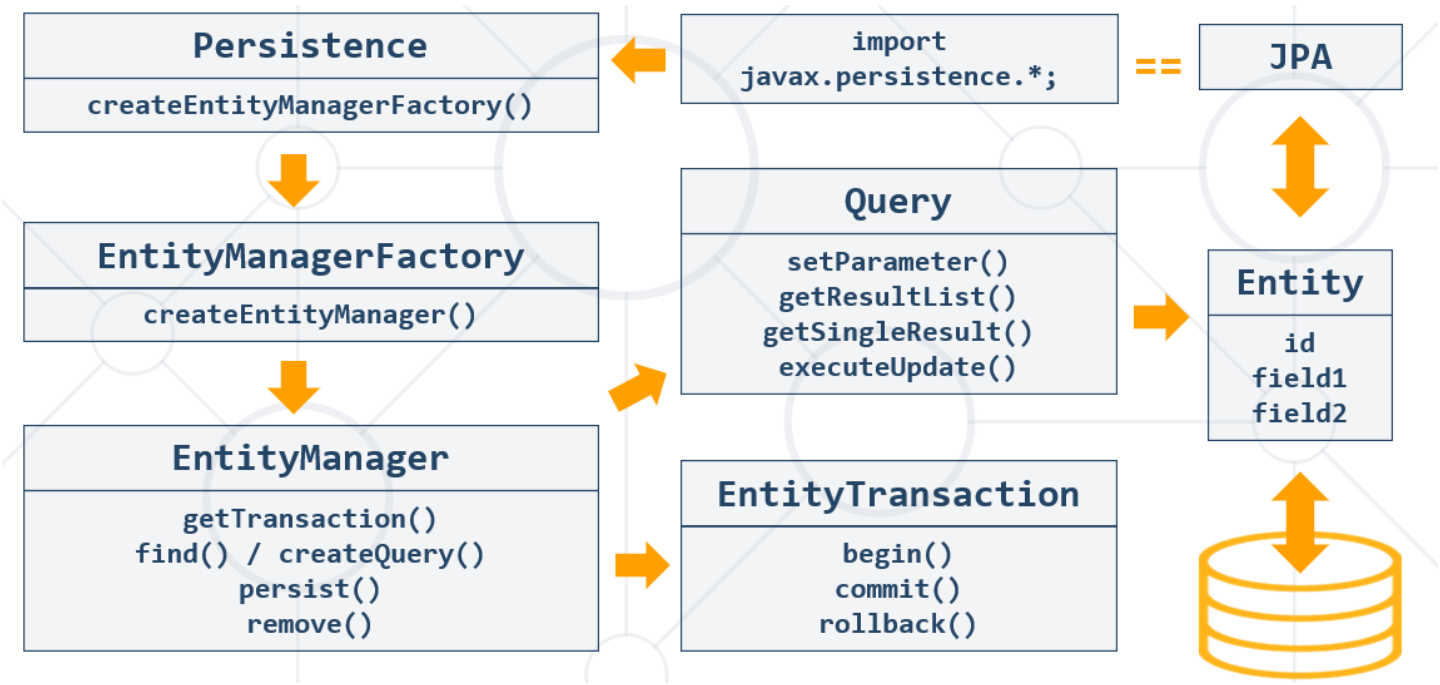

```xml
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence" version="2.0">
    <persistence-unit name="school"> //persistence unit name
        <properties>
            <property name="hibernate.connection.url"
                      value="jdbc:mysql://localhost:3306/school?createDatabaseIfNotExist=true"/>
            <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();
    }
}
```

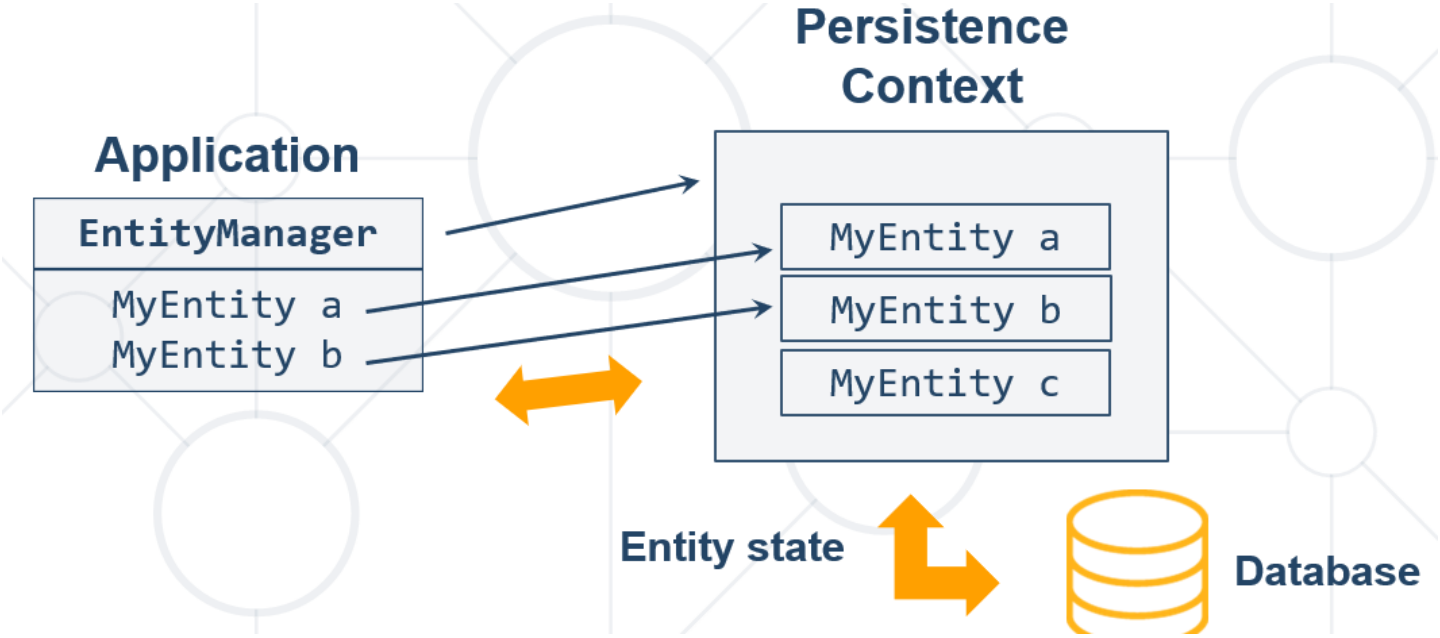
JPA schema – Java Persistence API



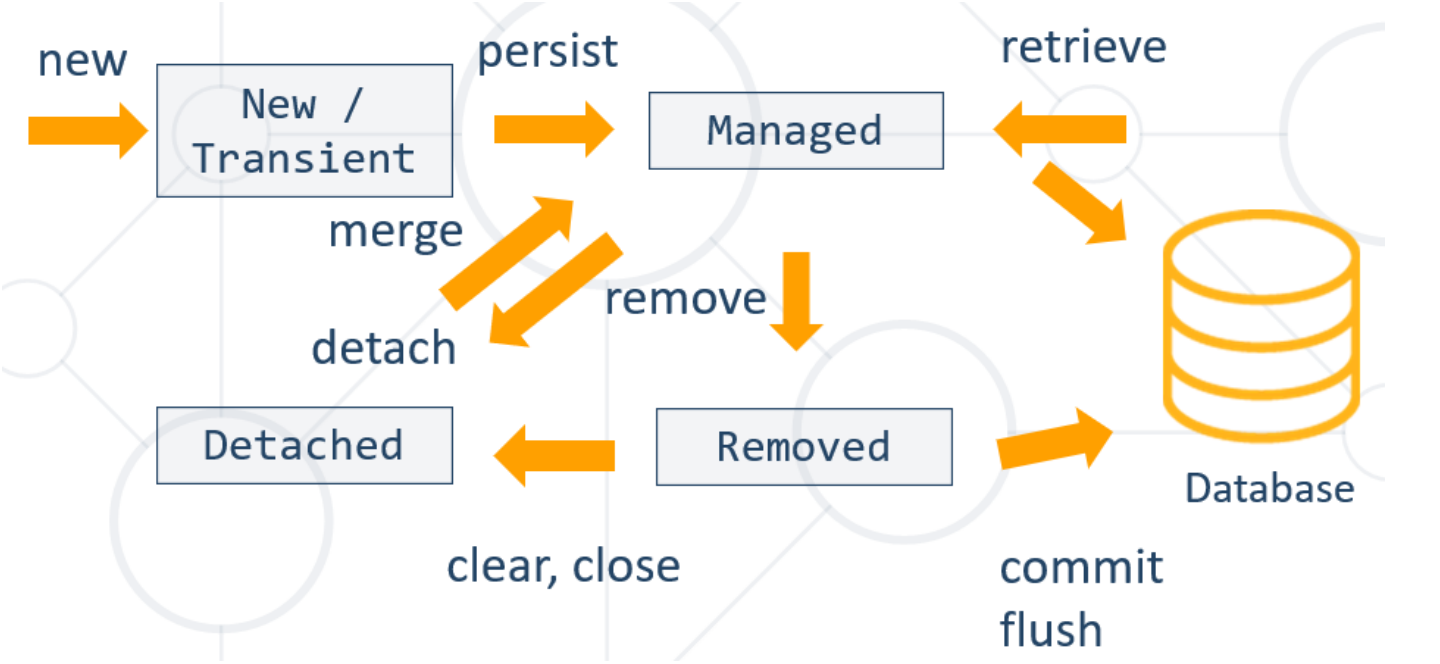
Java Persistence Query Language = JPQL

```
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 onl

```
import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;

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

```
import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;

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(не изтрили, а детачнали !!!) дадено entity (инстанция на класа Student примерно), то изтритият обект стои в паметта и можем да го върнем с 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);
}
```

Как съединяваме detach-нато entity извън persistence context/извън транзакция отново в статус Managed?
- с **@Transactional** анотация
- с **em.merge()**

JPA Retrieve Data by Criteria – more than one result

```
import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;


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

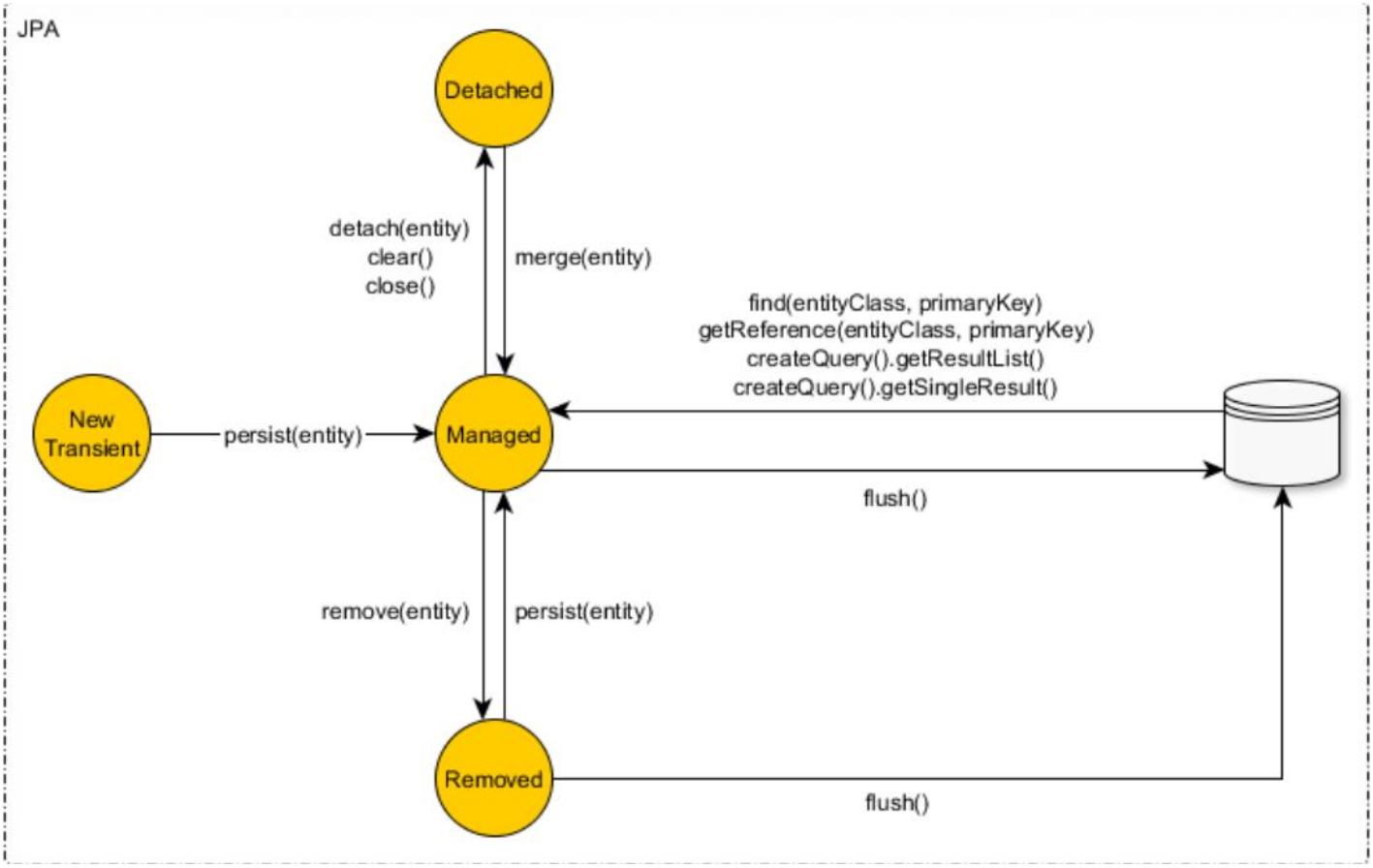
```
Query from_town = entityManager.createQuery("SELECT t FROM Town t", Town.class);
List<Town> resultList = from_town.getResultList();

for (Town town : resultList) {
    System.out.println(town);
}
```

3.9. Transaction Log and BackUp

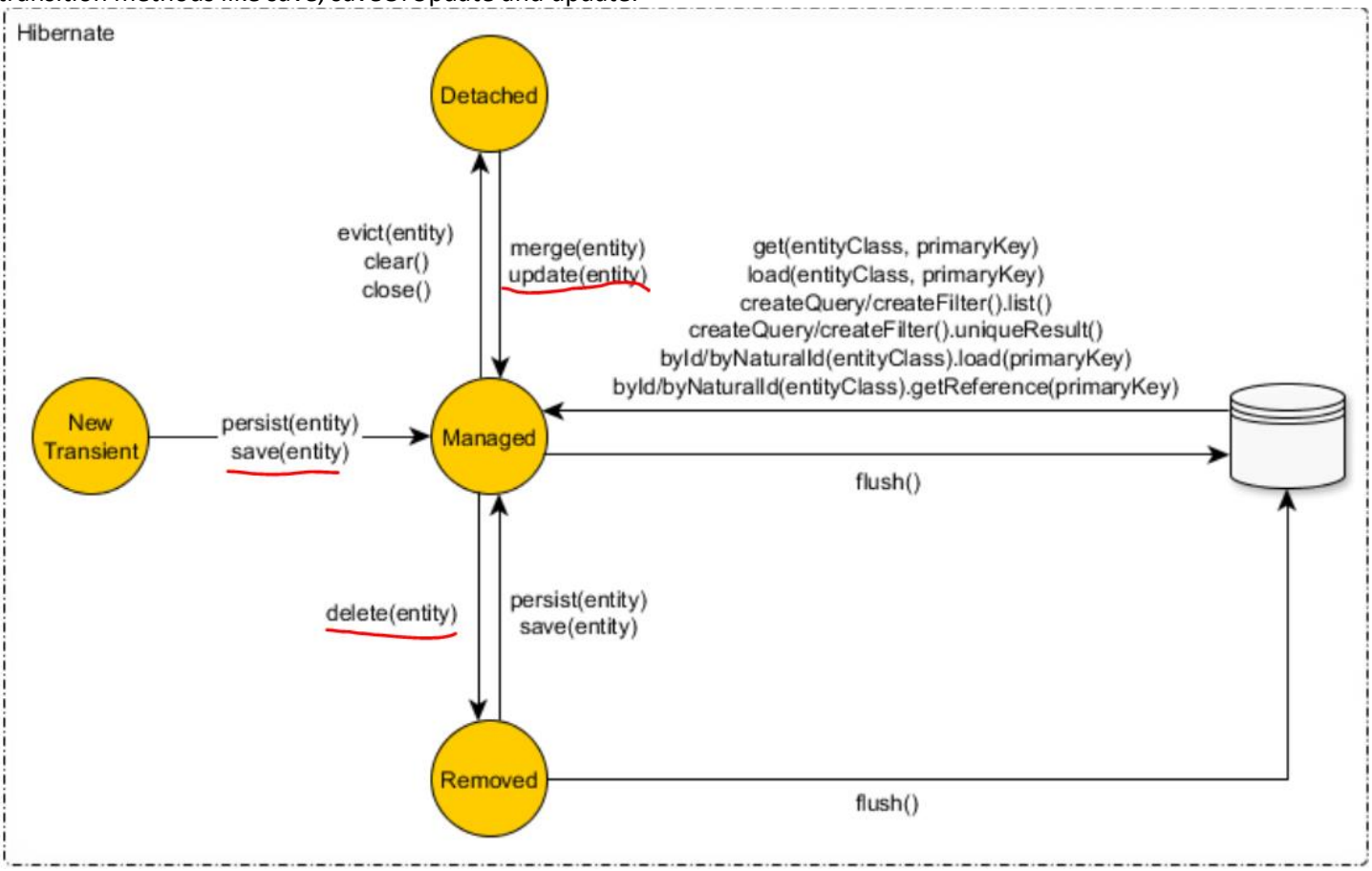
https://liquibase.org/
flyway

3.10. Differences between Hibernate without JPA (Hibernate existed before JPA) and only 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.

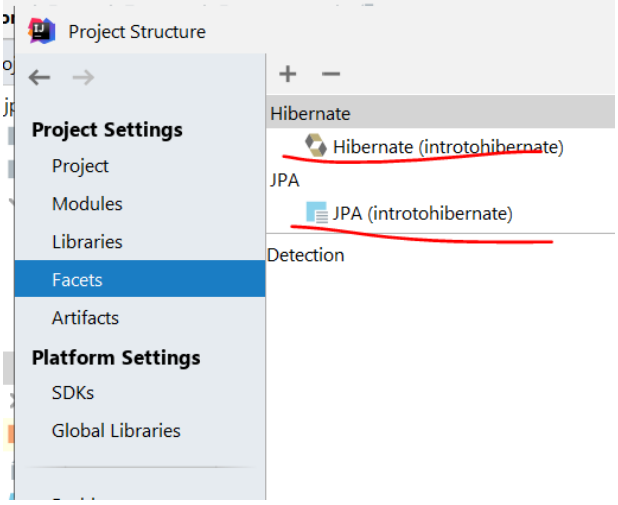
Можем ако сме на Hibernate with JPA да го извадим/преобразуваме само на Hibernate и да си ползваме оригиналните методи на Hibernate, които не са част от JPA изискуемата спецификация.

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 the JPA, we need a session object from javax.persistence.EntityManager. In this post, we will see How To get Hibernate Session From EntityManager in Spring Boot with Example.

```
import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;

import org.hibernate.Session;


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



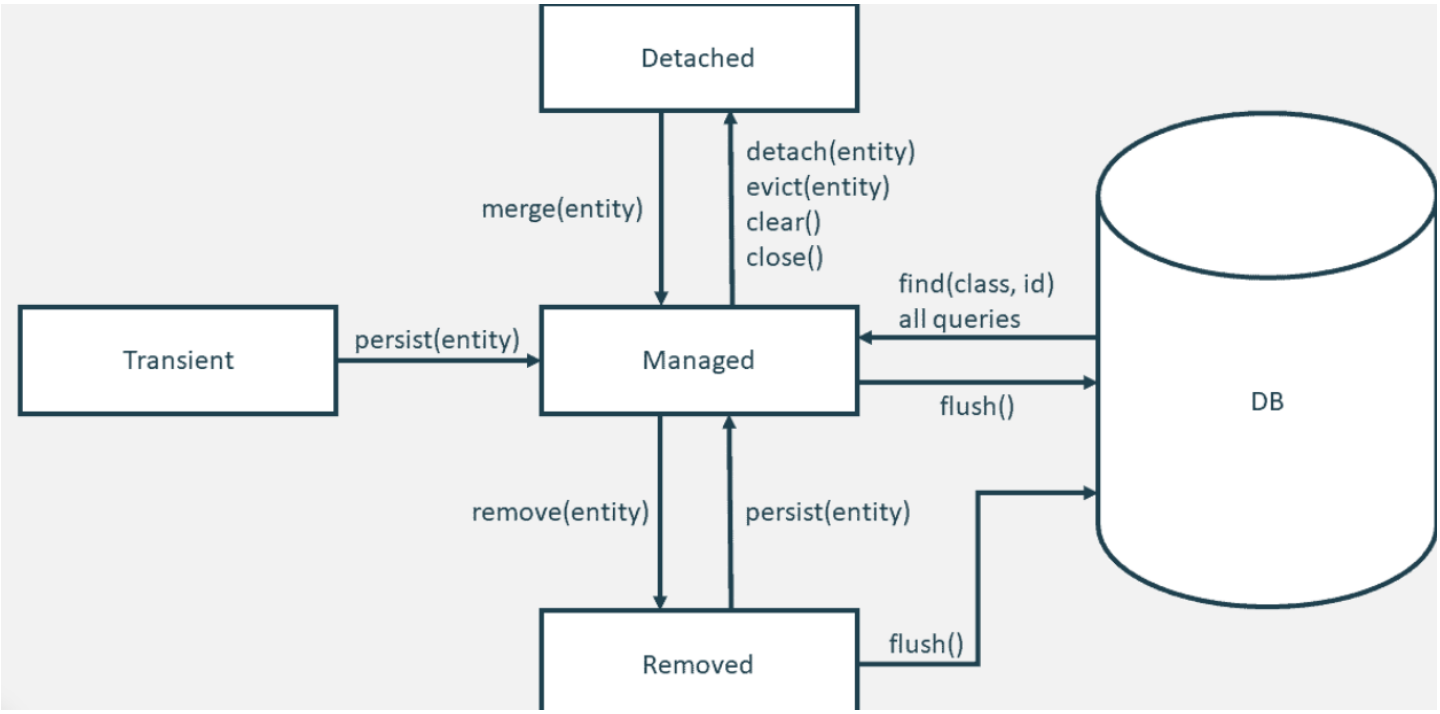
3.11. Hibernate pure 3 Lifecycle States

- **Persistent** – associated with an active Hibernate session
- **Detached** – associated to a Hubernate session, but the session was closed
- **Transient** – object was never associated with a session

3.12. JPA's 4 Lifecycle States

https://thorben-janssen.com/entity-lifecycle-model/?fbclid=IwAR3s6VaYrscygTxriYJwhA13Mw__4HFQvh20M7zKWhXSnM9VK9NQkIqCbgw

The lifecycle model consists of the 4 states –
Transient,
Managed,
Removed,
and Detached.



Transient

The lifecycle state of a newly instantiated entity object is called transient. The entity hasn't been persisted yet, so it doesn't represent any database record.

Your persistence context doesn't know about your newly instantiate object. Because of that, it doesn't automatically perform an SQL INSERT statement or tracks any changes. As long as your entity object is in the lifecycle state transient, you can think of it as a basic Java object without any connection to the database and any JPA-specific functionality.

```
1  Author author = new Author();
2  author.setFirstName("Thorben");
3  author.setLastName("Janssen");
```

That changes when you provide it to the EntityManager.find method. The entity object then changes its lifecycle state to managed and gets attached to the current persistence context.

Managed

All entity objects attached to the current persistence context are in the lifecycle state managed. That means that your persistence provider, e.g. Hibernate, will detect any changes on the objects and generate the required SQL INSERT or UPDATE statements when it flushes the persistence context.

There are different ways to get an entity to the lifecycle state managed:

1. You can call the EntityManager.persist *method with a new entity object.*

```
1  Author author = new Author();
2  author.setFirstName("Thorben");
3  author.setLastName("Janssen");
4  em.persist(author);
```

2. You can load an entity object from the database using the EntityManager.find *method, a JPQL query, a* CriteriaQuery*, or a native SQL query.*

```
1  Author author = em.find(Author.class, 1L);
```

3. You can merge a detached entity by calling the EntityManager.merge *method or update it by calling the* update *method on your Hibernate* Session*.*

```
1  em.merge(author);
```

Detached

An entity that was previously managed but is no longer attached to the current persistence context is in the lifecycle state detached.

An entity gets detached when you close the persistence context(not by removing it 😊). That typically happens after a request got processed. Then the database transaction gets committed, the persistence context gets closed, and the entity object gets returned to the caller. The caller then retrieves an entity object in the lifecycle state detached.

You can also programmatically detach an entity by calling the detach method on the EntityManager.

```
1  em.detach(author);
```

There are only very few performance tuning reasons to detach a managed entity. If you decide to detach an entity, you should first flush the persistence context to avoid losing any pending changes.

Reattaching an entity

You can reattach an entity by calling the update method on your Hibernate Session or the merge method on the EntityManager (or the @Transactional annotation). There are a few subtle differences between these operations that I explain in great detail in What's the difference between persist, save, merge and update? Which one should you use?

In both cases, the entity changes its lifecycle state from detached to managed.

Removed

When you call the remove method on your EntityManager, the mapped database record doesn't get removed immediately. The entity object only changes its lifecycle state to removed.

During the next flush operation, Hibernate will generate an SQL DELETE statement to remove the record from the database table.

```
em.remove(author);
```

Conclusion

All entity operations are based on JPA's lifecycle model. It consists of 4 states, which define how your persistence provider handles the entity object.

New entities that are not attached to the current persistence context are in the transient state.

If you call the persist method on the EntityManager with a new entity object or read an existing record from the database, the entity object is in the managed state. It's connected to the current persistence context. Your persistence context will generate the required SQL INSERT and UPDATE statement to persist the current state of the object.

Entities in the state removed are scheduled for removal. The persistence provider will generate and execute the required SQL DELETE statement during the next flush operation.

If a previously managed entity is no longer associated with an active persistence context, it has the lifecycle state detached. Changes to such an entity object will not be persisted in the database.

3.13. 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);
    }
}
```

3.14. 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(e -> e.setSalary(e.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",   //Deprecated, новия JPQL синтаксис е SIZE(a.employees),
COUNT(a.employees) става също
        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 – part of CriteriaAPI

```
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

Правилният вариант не е да го конкатенираме или пък да използваме Object[].

Правилният вараинт е да се направи с ръчно генериране на интерфейс или клас със съответните необходими полета, и този клас да го цитираме в JPQL заявката!!! В SpringData преподавателят така каза, че е правилния вариант / добрия вариант!!! Но не сме го имплементирали все още правилно!!!

```
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 Native with xml Table Relations, Cascade, Inheritance

Table relations

In **<class name>.hbm.xml** files

Unidirectional

- Unidirectional `Many-to-one` example:

```
<class name="Employees">
        <id name="id" column="id">
                <generator class="native"/>
        </id>
        <many-to-one name="department"
                        column="deparmentId"
                        not-null="true"/>
</class>
<class name="Departments">
        <id name="id" column="departmentId">
                <generator class="native"/>
        </id>
</class>
```

- Unidirectional `One-to-one` (using many-to-one and a unique constraint) example:

```
<class name="Employees">
        <id name="id" column="id">
                <generator class="native"/>
        </id>
        <many-to-one name="department"
                        column="deparmentId"
                         unique="true",
                        not-null="true"/>
</class>
<class name="Departments">
        <id name="id" column="departmentId">
                <generator class="native"/>
        </id>
</class>
```

- Unidirectional `One-to-many` example:

```
<class name="Employees">
        <id name="id" column="id">
                <generator class="native"/>
        </id>
</class>
<class name="Departments">
        <id name="id" column="departmentId">
                <generator class="native"/>
        </id>
        <set name="employees">
                <key column="departmentId" not-
null="true"/>
                <one-to-many class="Employee"/>
        </set>
</class>
```

- Unidirectional `Many-to-many` example:

```
<class name="Employee">
        <id name="id" column="id">
                <generator class="native"/>
        </id>
</class>
<class name="Department">
        <id name="id" column="departmentId">
                <generator class="native"/>
        </id>
        <set name="employees">
                <key column="departmentId" not-
null="true"/>
                <one-to-many class="Employee"/>
        </set>
</class>
```

- Unidirectional `One-to-many` (using a join table and a `many-to-many` with unique constraint) example:

```
<class name="Employees">
        <id name="id" column="id">
                <generator class="native"/>
        </id>
        <set name="skills"
        table="EmployeeSkills">
                <key column="employeeId"
                        not-null="true"/>
                <many-to-many class="Skills"
                        column="skillId"
                        unique="true"/>
        </set>
</class>
<class name="Skills">
        <id name="id" column="id">
                <generator class="native"/>
        </id>
</class>
```

Bidirectional

When the mapping is bidirectional (meaning that both the parent and the child entity define the mapping), you always have to mark one of the mappings with inverse = “true” so that Hibernate can determine which relationship endpoint to use for persistence.

Т.е. винаго едното entity е водещото, и маркираме ответното/отсрещното (това което не е водещото) entity с inverse=”true”.

Cascade

In **<class name>.hbm.xml** files

Only parent entities are saved by default with *save()* – to save also modifications to child entities specify a *save-update* cascade option.

```
<class name="model.Professor" table="PROFESSORS">
...
<set name="courses" table="COURSES"
      cascade="save-update">
      <key column="PROFESSORID"/>
      <one-to-many class="model.Course"/>
</set>
</class>
```

Advanced features

- In XML configuration file, the **<entity-name>** property could be specified additionally instead of using the <hibernate-mapping> …. <class>

```
<hibernate-mapping>
      <class entity-name="employee">
      …
      </class>
</hibernate-mapping>
```

    - The main benefit in using maps instead of POJOs is for creating quick prototypes
    - The main drawback is that compile-type type checking is lost

- Tupilizers can be used to customize mapping of particular fields (components) or entire entities in Hibernate
- Tupilizers are specified for the particular entity/field in the Hibernate mapping XML/ annotations configuration

- Hibernate does not differentiate between the usage of tables and views

- Some RDBMS do not support creation of views – in this case you can use Hibernate to link an entity to a query, thus simulating a view creation in Java

• Example (using the `@Subselect` annotation):

```
@Entity
@Subselect("select Name, Phone from Employees"
@Synchronize( {"Employees"} ) //tables impacted
public class Employee {
      @Id public String getId()
      {
            return id;
      }
... }
```

• Example (using XML configuration):

```
<class name="Employee">
      <subselect>select Name, Phone from Employees
      </subselect>
      <synchronize table="Employees"/>
      <id name="id"/>
      ...
</class>
```

- You can implement application-level **triggers** using **interceptors** (instances of ***org.hibernate.EmptyInterceptor***) – better they have named it NoOptInterceptor (да индикира, че не прави нищо)
- Interceptors can be used to inspect and manipulate properties of persistent entities once they are loaded, saved, updated or deleted – можем да наследим този EmptyInterceptor и да си override-нем поведение

```
/**
 * An interceptor that does nothing. May be used as a base class
 * for application-defined custom interceptors.
 *
 * @author Gavin King
 */
public class EmptyInterceptor implements Interceptor, Serializable {

    public static final Interceptor INSTANCE = new EmptyInterceptor();

    protected EmptyInterceptor() {}
```

```
    public boolean onLoad(
            Object entity,
            Serializable id,
            Object[] state,
            String[] propertyNames,
            Type[] types) {
        return false;
    }

    public boolean onSave(
            Object entity,
            Serializable id,
```

- Hibernate supports batch processing of queries by using the JDBC batch mechanisms
- Batch size is set using the *hibernate.jdbc.batch_size* property

- You can mark a column with the *@Version* annotation on a version number or date/timestamp column in order to enable optimistic locking
- Optimistic locking in that manner allows Hibernate to determine if an entity has been changed by another transaction in order to provide transaction consistency
- Version columns are typically generated automatically by Hibernate or the underlying database

- A property can also be a collection of simple types instead of a collection of entities – таблицата Nicknames служи като контейнер и всеки път като извикваме getNicknames() вземаме данни от таблицата Nicknames

```
@Entity
public class Employee
{
      @ElementCollection
      @CollectionTable(name="Nicknames", joinColumns
=@JoinColumn(name="user_id"))
      @Column(name="nickname")
      public Set<String> getNicknames() { ... }
}
```

- A property can also be a collection of embeddable types (components) instead of a collection of entities

- Example:

```
@Entity
public class Employee {
      @ElementCollection
      @CollectionTable(name="Addresses", joinColumns
     =@JoinColumn(name="user_id"))
      @AttributeOverrides({@AttributeOverride(name="
     city", column=
      @Column(name="col_city"))    })
      public Set<Address> getAddresses() { ... }
}
@Embeddable
public class Address {
      public String getCity() {...}  }
```

- Hibernate uses a fetching strategy to retrieve objects if the application needs to navigate the association – by default LAZY or EAGER depending on the configuration of the tables relations
- Fetch strategies can be declared in the O/R mapping metadata, **or can be over-ridden by a particular HQL or Criteria query**

- Common fetching strategies:
  - JOIN FETCH – associated instance/collection is retrieved as part of the SELECT that retrieves the parent instance **even when lazy is set to true**
  - SELECT fetching – a separate SELECT is used to retrieve child entities – executed only when children are requested **only when lazy = "true"** table relations (unless lazy = "false" is set)

Best practices

- Define ID fields for entities
- If using XML mapping files, define each entity class in a separate XML file
- Use named/positional bind variables in queries (именувани или неименувани параметри в заявките)

Design patterns

- Entity classes may also be called data transfer objects (DTO) – use a consistent naming schema for Hibernate entities
- Defining data access classes for the mapped entities with the following functionality:
  - Finding entities by primary key
  - Creating and updating entities
  - Deleting existing entities
  - Finding entities by criteria with support for paging and sorting
- Data Access classes are also called Data Access Object (DAO) classes – **класове, които реализират заявки към базата данни!**
- An **abstract** DAO **class or interface** can be defined to provide the common functionality of all DAOs (e.g. abstract method that must be implemented by all DAOs such *as find(Long id) or listAll()*) – в случая с Native Hibernate (а и с JPA Hibernate) самата сесия има доста от тези методи вече дефинирани и също така имплементирани за нас. Когато говорим за SpringData – там репозитори interface-те играят ролята/са дефакто тези DAO обекти.

5. JPA Hibernate Code First Entity Relations - Advanced Mapping with Annotations

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.

5.1. Info

- Entity annotations are two types:
  - Physical – describe the association/relation between entities
  - Logical – describe the database schema (tables, columns)

- Hibernate-specific annotations are in the package **org.hibernate.annotations**

- JPA-specific annotations are in the package **javax.persistence**

- Every non-static and not-transient property (field or method depending on the access type) of an entity is considered persistent, unless you annotate it as *@Transient*.
- By default the access type of a class hierarchy is defined by the position of the *@Id* or *@EmbeddedId* annotations
- If Hibernate annotations are on a field, then only fields are considered for persistence and the state is accessed via the field.
- If Hibernate annotations are on a getter, then only the getters are considered for persistence and the state is accessed via the getter/setter
- Access type can be enforced by means of the *@Access* annotation specified on the entity class (not recommended – should be used only if necessary)
- Fields can be derived by an entity by referenced classes (also called embedded objects or components)
- Component classes must be marked with an *@Embeddable* annotation
- Using components you can improve reusability of source code by extracting common entity fields to components (if applicable) – демек уж за удобство можем да преизползваме тези класове анотирани с *@Embeddable*

5.2. Когато слагаме много полета в клас, то по-добре е самите полета да не са анотирани, а да анотираме само getters – по четимо е така

```
@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;
    }
```

5.3. @Id анотация

- Hibernate provides different strategies for setting the @Id field:
    - IDENTITY – set by the RDBMS by means of an identity column (is supported by the RDBMS)
    - TABLE – uses a hi/lo algorithm to generate identifiers given a table and a column as a source of hi values - //пази уникални ключове измежду много таблици
    - SEQUENCE – uses a hi-/lo algorithm to generate identifiers using a named database sequence
    - AUTO (default one) – selects any of the previous based on the capabilities of the underlying database
    - Custom – custom ID generator

5.4. 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:
    - MappedSuperclass
    - 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

0. *@MappedSuperclass* Strategy

@MappedSuperclass and provides the shared attributes with their mapping annotations. As you can see, *Publication* has no *@Entity* annotation and will not be managed by the persistence provider.

```
@MappedSuperclass
public abstract class Publication {
```

```
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    @Column(name = “id”, updatable = false, nullable = false)
    protected Long id;

    @Column
    protected String title;

    @Version
    @Column(name = “version”)
    private int version;

    @Column
    @Temporal(TemporalType.DATE)
    private Date publishingDate;

    …
}
```

1. ***TABLE_PER_CLASS*** Strategy

The table per class strategy is similar to the mapped superclass strategy. The main difference is that the superclass is now also an entity.

This mapping allows you to use polymorphic queries and to define relationships to the superclass. But the table structure adds a lot of complexity to polymorphic queries, and you should, therefore, avoid them!

- **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  // auto_increment
    @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")
```

```java
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;
    }
}
```

```java
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;
    }
```

```java
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();
```

- Result:

bikes	
id	type
1	"BIKE"

cars	
id	type
2	"CAR"

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.

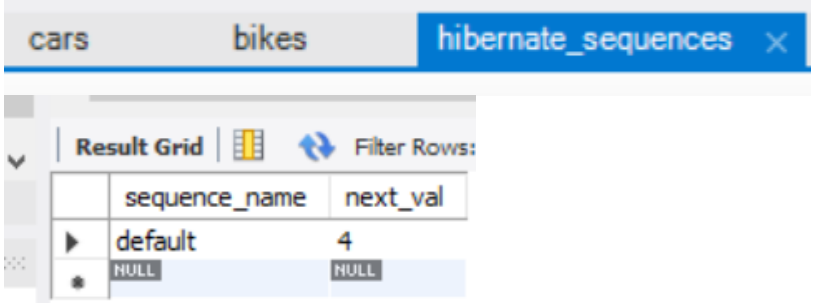


Table Per Class Strategy: Conclusion

- **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**

The base class is represented by a single table and each class in the hierarchy has its own table, which holds just the fields specific to that entity. Loading the entity almost always requires at least one database JOIN operation.

Example:

```
@Entity
@Table(name = "vehicles")
@Inheritance(strategy = InheritanceType.JOINED)
public abstract class Vehicle {
    @Id   // auto_increment
    @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, няма таблица за SQL от този клас да има
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

The joined table approach maps each class of the inheritance hierarchy to its own database table. This sounds similar to the table per class strategy. But this time, also the abstract superclass *Publication* gets mapped to a database table. This table contains columns for all shared entity attributes. The tables of the subclasses are much smaller than in the table per class strategy. They hold only the columns specific for the mapped entity class and a primary key with the same value as the record in the table of the superclass.

3. ***SINGLE_TABLE*** Strategy – най-използваната/предпочитаната

The single table strategy maps all entities of the inheritance structure to the same database table. This approach makes polymorphic queries very efficient and provides the best performance.

- **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!

When you persist all entities in the same table, Hibernate needs a way to determine the entity class each record represents. This is information is stored in a discriminator column which is not an entity attribute. You can either define the column name with a *@DiscriminatorColumn* annotation on the superclass or Hibernate will use *DTYPE* as its default name.

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  @GeneratedValue(strategy = GenerationType.IDENTITY)
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, няма таблица за SQL от този клас да има
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;
    }
```

You should also provide a *@DiscriminatorValue* annotation. It specifies the discriminator value for this specific entity class so that your persistence provider can map each database record to a concrete entity class.

As I explained at the beginning of this section, the single table strategy allows easy and efficient data access. All attributes of each entity are stored in one table, and the query doesn't require any join statements. The only thing that Hibernate needs to add to the SQL query to fetch a particular entity class is a comparison of the discriminator value.

```
@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);
    }
}
```

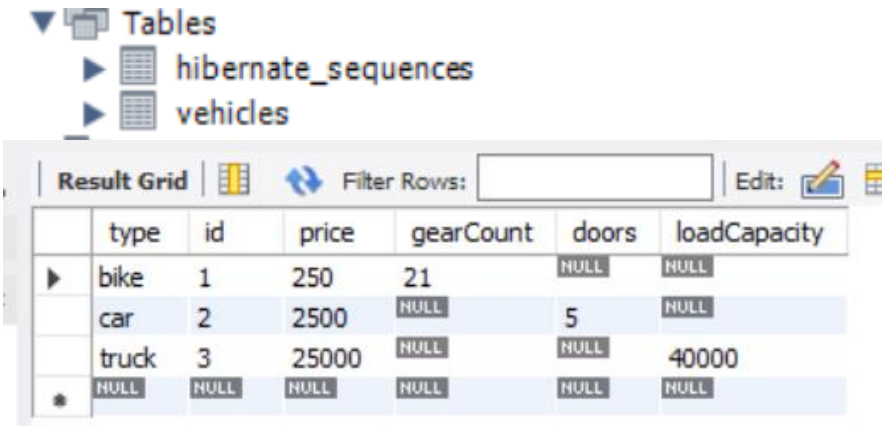
```java
@Entity
@Table(name = "bikes")
@DiscriminatorValue("bike") //какво да пише в дискриминационната колона type
public class Bike extends Vehicle {
    private static final String BIKE_TYPE = "Bike";

    private int gearCount;

    public Bike(int gearCount) {
        super(BIKE_TYPE, 250);
        this.gearCount = gearCount;
    }
```

```java
@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 – Single Table Strategy

Създава само една таблица **vehicles,** и hibernate_sequences както обикновено



Polymorphic queries no drama here. All entities of the inheritance hierarchy are mapped to the same table and can be selected with a simple query

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

Example DEFAULT discriminator column

```
@Entity
@Table(name = "Users")
public abstract class User {
        @Id
        @GeneratedValue(strategy = GenerationType.AUTO)  === @GeneratedValue(strategy = GenerationType.IDENTITY)
        private Long id;
        // Other fields and constructors
        public Long getId() {
        return id;
        }
        public void setId(Long id) {
        this.id = id;
        }
        // Other methods
}

@Entity
public class Manager extends User {
        // no id attribute, it is inheritted
}
@Entity
public class Author extends User {
        // no id attribute, it is inheritted
}
@Entity
public class Subscriber extends User {
        // no id attribute, it is inheritted
}
```

There is also a discriminator column (called by default DTYPE for Hibernate), which contains information about the entity to which the corresponding record belongs.
So, for example, if we insert a Manager record and we rely on the conventions - **DTYPE column will be set to Manager** for that record in the table Users.

4. Choosing strategy

- If you require the best performance and need to use polymorphic queries and relationships, you should choose the single table strategy. But be aware, that you can't use not null constraints on subclass attributes which increase the risk of data inconsistencies.
- If data consistency is more important than performance and you need polymorphic queries and relationships, the joined strategy is probably your best option.

- If you don't need polymorphic queries or relationships, the table per class strategy is most likely the best fit. It allows you to use constraints to ensure data consistency and provides an option of polymorphic queries. But keep in mind, that polymorphic queries are very complex for this table structure and that you should avoid them.

5.5. Table Relations - връзки

В който entity клас създадем допълнително поле, на което задаваме релация, това допълнително поле се добавя в базата данни.(т.н. foreign key).

Как разбираме ЛЕСНО кога да използваме mappedBy и кога не – пишем @OneToMany() и кликаме Alt+Enter и то ни излиза какво можем да добавим.

Ако не използваме mappedBy, то вариантът е да използваме в допълнение @JoinColumn

Винаги има вариант да използваме само и единствено една анотация за релация(@ManyToMany или другите) и то без параметри, и persistence се справя!

С mappedBy казваме, че връзката ни е Bidirectional

Реално имаме mappedBy само от едната страна на връзката.

Но за двупосочна връзка си пишем съответните анотации и от двете страни, и Hybernate си знае за връзката като сме задали от едната страна mappedBy.

Когато създадем унидиректионал еднопосочна връзка, то е добре да я направим bidirectional и в двете посоки. Не е ок да получаваме две отделни връзки за едно и също нещо….?

Database Relationships

https://thorben-janssen.com/complete-guide-inheritance-strategies-jpa-hibernate/?fbclid=IwAR0lDIMrbasNhcij4oKen4GjxLFpU0bpDoboEwdP1wGeAEcVdxpMSHMEenc

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

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.

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.

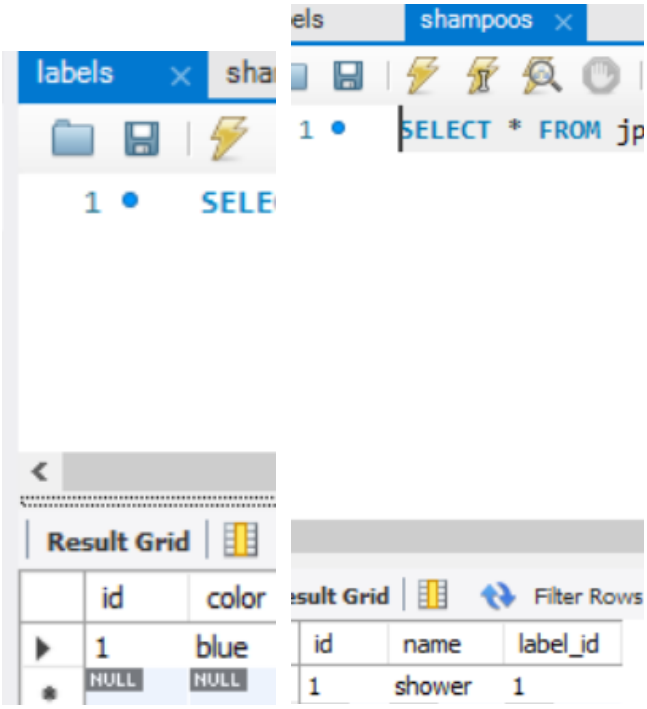
One-To-One – Unidirectional - Еднопосочно

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


```java
@Entity
@Table(name = "shampoos")
public class BasicShampoo {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private  int id;

    private String name;

    @OneToOne(optional = false) //Runtime evaluation false няма право(NOT NULL в SQL)
    @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

```java
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) 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;


@Entity
@Table(name = "labels")
public class BasicLabel {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;
    private String color;

С mappedBy казваме, че връзката ни е Bidirectional
    @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

Много шампоани като всеки от тях има production batch.

Полето ForeignKey е от Many страната на релацията.


```java
@Entity
@Table(name = "batches")
public class ProductionBatch {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;

    @Column(name = "created_at")
    private LocalDate createdAt;
```

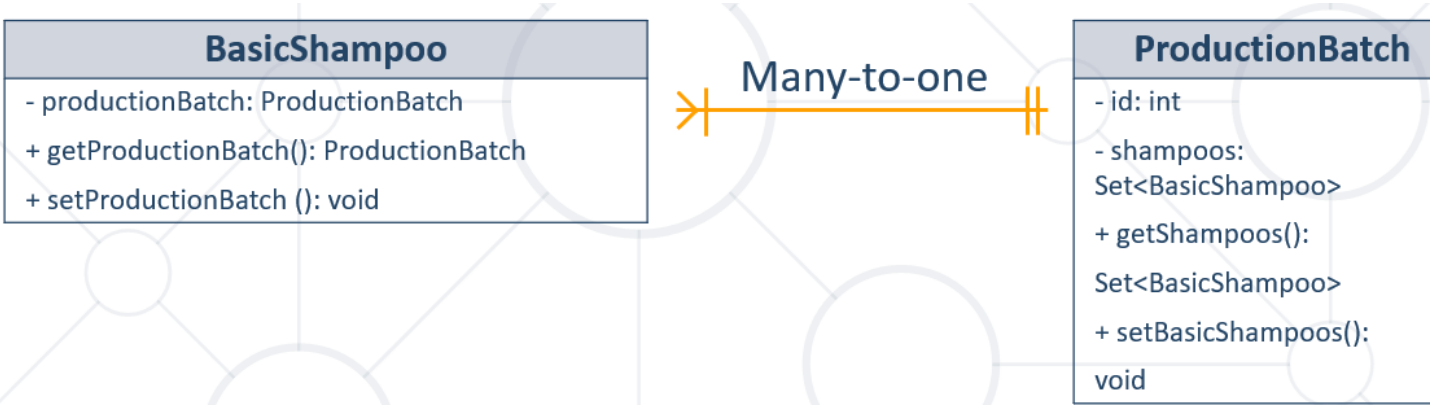
```java
@Entity
@Table(name = "shampoos")
public class BasicShampoo {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private  int id;

    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;


        //Все едно външен ключ foreign key създава
     //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

```java
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

Полето ForeignKey е от Many страната на релацията.




```
@Entity
@Table(name = "batches")
public class ProductionBatch {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;

    @Column(name = "created_at")
    private LocalDate createdAt;
```

С mappedBy казваме, че връзката ни е Bidirectional

```
    //един batch да знае, че има много шампоани
    @OneToMany (mappedBy = "batch", targetEntity = BasicShampoo.class,  //поле на класа и кой е клас
            fetch = FetchType.LAZY,        //не ги вземай/зареждай всички шампоани всеки път //FetchType.EAGER е обратното
            cascade = CascadeType.ALL)    //ако изтриеш batch да изтрие/прави със свързаните неща/записи
{CascadeType.REFRESH, CascadeType.PERSIST}   //можем да изброяваме няколко каскадни типа наведнъж
    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

Прави трета mapping таблица с primary composite key от primary keys на първоначалните две таблици.

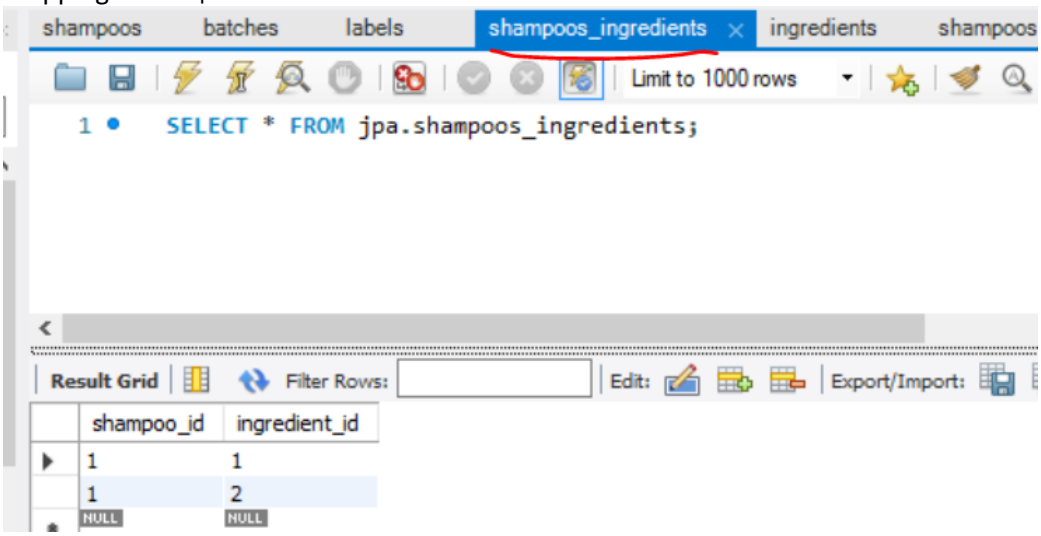
```
@Entity
@Table(name = "ingredients")
public class BasicIngredient {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;
```

```
    private double quantity;

    @Column(name = "chemical_name", nullable = false)  //NOT NULL, трябва да има стойност
    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

Прави трета mapping таблица с primary composite key от primary keys на първоначалните две таблици.

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)  //NOT NULL, трябва да има стойност
```

```
    private String chemicalName;


С mappedBy казваме, че връзката ни е Bidirectional
//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;
    }
```

По изчистен вариант само с Many-To-One връзки

Релацията One-To-Many прави допълнителна таблица.
Релацията Many-to-One не прави допълнителна помощна таблица.
Ако използваме само Many-To-One релацията, то таблицата откъм One страната реално не е свързана с таблицата откъм Many страната. Или с други думи ако искаме да вземем данни от таблицата откъм One страната заедно с данни от таблицата откъм Many страната, то правим две заявки към базата - за двете таблици. Или всяка таблица поотделно си вземаме данните.

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);
```

Result Grid | Filter Rows:

BasicShampoo_id	someNames
1	Pesho
1	Gosho
1	Kiro

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

```
@ManyToOne    e  default fetch type EAGER
```

  - **LAZY** – retrieves all reachable entity objects **only when fetched entity's getter method is called** (използва се при @One**ToMany**)

```
@OneToMany    e  default fetch type LAZY
@ManyToMany
```

```
University university = em.find((long) 1); // collection students = null

// The collection holding the students is populated when the getter is called
university.getStudents();
```

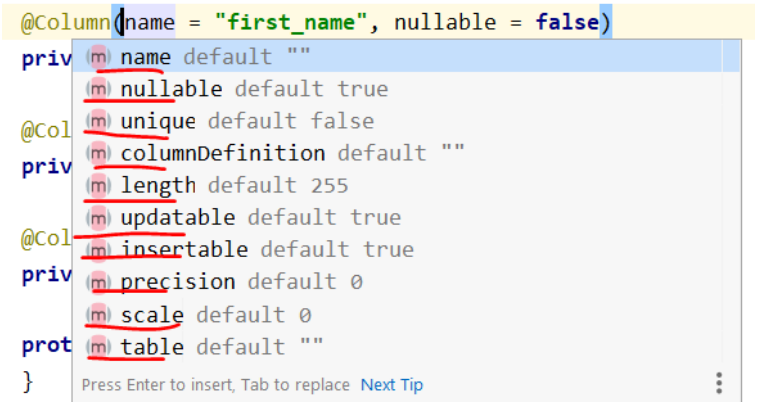
Когато почнем да принтираме например, чак тогава ги вика всичките!!!

Cascading

- JPA translates **entity state transitions** to database **DML (Data Manipulation Language)** statements
  - This behavior is configured through the **CascadeType** mappings
- **CascadeType.PERSIST**: means that save() or persist() operations cascade to related entities – когато се опитваме да save-нем обект, който в себе си има друг обект, то с CascadeType.PERSIST ще се запишат в базата както глобалния обект, така и вложения обект ако не съществува
- **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 – като изтривам обект, първи изтрива вложените обекти, и след това обекта, който изтривам
- **CascadeType.DETACH**: detaches all related entities if a "manual detach" occurs
- **CascadeType.ALL**: is shorthand for all of the above cascade operations – да не се налага да се записвам първо запис в едната таблица(обект да persist-нем в Java), след това определен друг запис(persist) в друга таблиц, и т.н. – с **CascadeType.ALL всичко върви без значение кое след кое записваме или изтриваме при вложени обекти.**

5.6. @Column анотация

```
@Entity
@Table(name = "_01_wizzards_deposits")
public class WizzardDeposits {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;

    @Column(name = "first_name", length = 50)
    private String firstName;

    /*
    @Lob
    @Column(name = "last_name", length = 60, nullable = false, columnDefinition = "BLOB NOT
NULL")
@Column(columnDefinition = "TEXT")

//    @Column(columnDefinition = "LONGTEXT")
@Lob  //работи и като longtext


    private String lastName;*/

    @Column(name = "last_name", length = 60, nullable = false)
    private String lastName;

    @Column(length = 1000)
    private String notes;

    private int age;  //равносилно на NOT NULL. Ако използваме Integer, тогавам можем да имаме
NULL, т.е. типа му е Optional един вид на бащиния Integer от Java към SQL-а.
```



5.7. Пример с Inheritance и с Relations strategies

Пример 1:

```
@Entity
@Table(name = "ingredients")
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name = "type", discriminatorType = DiscriminatorType.STRING)
public abstract class BasicIngredient implements Ingredient {
//…

    @ManyToMany(mappedBy = "ingredients", targetEntity = BasicShampoo.class)
    private Set<BasicShampoo> shampoos;
//…
}
```

Пример 2 - Bills Payment System:

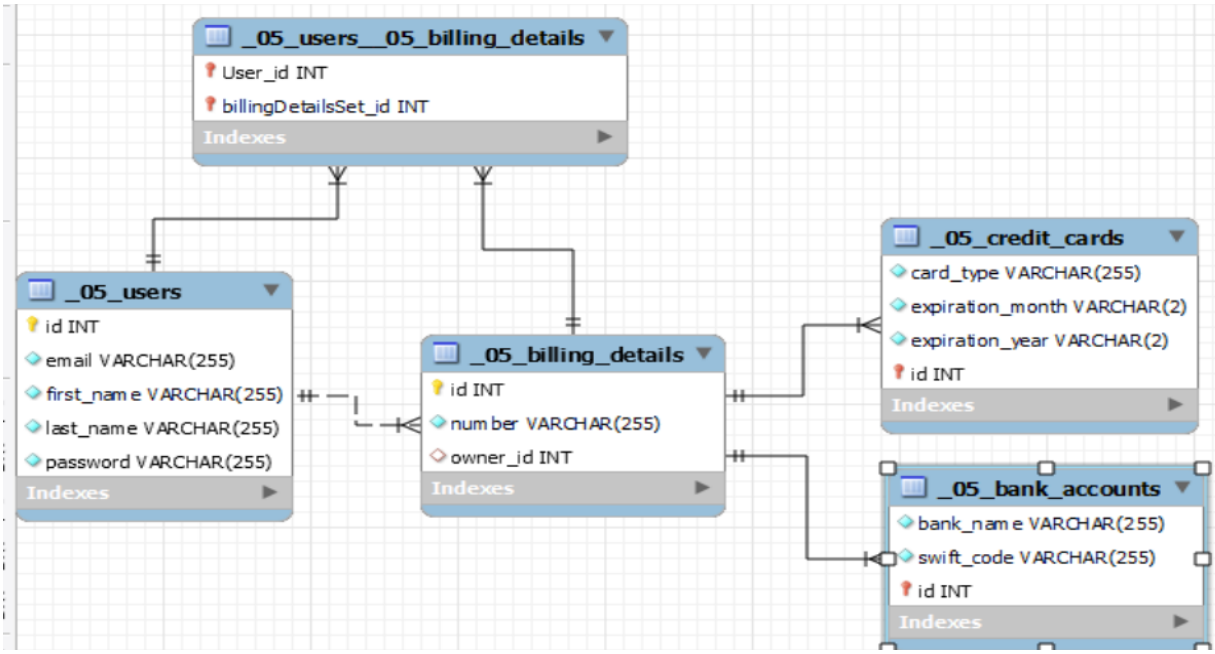
```
public class _05Main {
    public static void main(String[] args) {
        EntityManagerFactory emf = Persistence.createEntityManagerFactory("jpa_relations_exc");
        EntityManager entityManager = emf.createEntityManager();

        entityManager.getTransaction().begin();

        User user1 = new User("Bash", "Maistora", "mymail@abv.bg", "mypassword");
        User user2 = new User("Svilen", "Velikov", "svill@abv.bg", "mypasswordYee");

        //user 1
        BillingDetail billingDetail1 = new CreditCard("987654321", user1, "MASTERCARD",
                "08", "24");

        BillingDetail billingDetail2 = new BankAccount("111222333", user1, "OBB",
                "UBBS12D4");
        user1.addBillingDetail(billingDetail1);
        user1.addBillingDetail(billingDetail2);

        //user 2
        BillingDetail billingDetail3 = new CreditCard("12345678", user2, "VISA",
                "10", "23");
        BillingDetail billingDetail4 = new BankAccount("333222111", user2, "Bulbank",
                "BBBU12Z4");
```

```
        BillingDetail billingDetail5 = new BankAccount("999888777", user2, "Bulbank",
                "BBBU12Z4");
        user2.addBillingDetail(billingDetail3);
        user2.addBillingDetail(billingDetail4);
        user2.addBillingDetail(billingDetail5);

        entityManager.persist(user1);
        entityManager.persist(user2);

        entityManager.persist(billingDetail1);
        entityManager.persist(billingDetail2);
        entityManager.persist(billingDetail3);
        entityManager.persist(billingDetail4);
        entityManager.persist(billingDetail5);

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

@Entity
@Table(name = "_05_users")
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;

    @Column(name = "first_name", nullable = false)
    private String firstName;

    @Column(name = "last_name", nullable = false)
    private String lastName;

    @Column(nullable = false)
    private String email;

    @Column(nullable = false)
    private String password;

    @OneToMany //OneToMany пак може да има свързваща mapping таблица
    private Set<BillingDetail> billingDetailsSet = new HashSet<>();

    public Set<BillingDetail> getBillingDetailsSet() {
        return Collections.unmodifiableSet(billingDetailsSet);
    }

    public void addBillingDetail(BillingDetail bl) {
        this.billingDetailsSet.add(bl);
    }

    public User() {
    }

    public User(String firstName, String lastName, String email, String password) {
        this.firstName = firstName;
        this.lastName = lastName;
        this.email = email;
        this.password = password;
    }

    public int getId() {
```

```
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public String getFirstName() {
        return firstName;
    }

    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    public String getLastName() {
        return lastName;
    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
    }

    public String getEmail() {
        return email;
    }

    public void setEmail(String email) {
        this.email = email;
    }

    public String getPassword() {
        return password;
    }

    public void setPassword(String password) {
        this.password = password;
    }
}
//@MappedSuperclass //нещо междинно го анотира, няма таблица за SQL от този клас да има, will not be managed by the persistence provider
@Entity
@Table(name = "_05_billing_details")
@Inheritance(strategy = InheritanceType.JOINED)
public abstract class BillingDetail {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private  int id;

    @Column(nullable = false, unique = true)
    private String number;

    @ManyToOne
    private User owner;

    protected BillingDetail() {
    }

    protected BillingDetail(String number, User owner) {
        this.number = number;
        this.owner = owner;
```

```
    }

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public String getNumber() {
        return number;
    }

    public void setNumber(String number) {
        this.number = number;
    }

    public User getOwner() {
        return owner;
    }

    public void setOwner(User owner) {
        this.owner = owner;
    }
}

@Entity
@Table(name = "_05_bank_accounts")
public class BankAccount extends BillingDetail {
    @Column(name = "bank_name", nullable = false)
    private String bankName;

    @Column(name = "swift_code", nullable = false)
    private String swiftCode;

    public BankAccount() {
        super();
    }

    public BankAccount(String number, User owner, String bankName, String swiftCode) {
        super(number, owner);
        this.bankName = bankName;
        this.swiftCode = swiftCode;
    }

    public String getBankName() {
        return bankName;
    }

    public void setBankName(String bankName) {
        this.bankName = bankName;
    }

    public String getSwiftCode() {
        return swiftCode;
    }

    public void setSwiftCode(String swiftCode) {
        this.swiftCode = swiftCode;
```

```
    }
}


@Entity
@Table(name = "_05_credit_cards")
public class CreditCard extends BillingDetail {
    @Column(name = "card_type", nullable = false)
    private String cardType;

    @Column(name = "expiration_month", nullable = false, length = 2)
    private String expirationMonth;

    @Column(name = "expiration_year", nullable = false, length = 2)
    private String expirationYear;

    public CreditCard() {
        super();
    }

    public CreditCard(String number, User owner, String cardType, String expirationMonth, String
expirationYear) {
        super(number, owner);
        this.cardType = cardType;
        this.expirationMonth = expirationMonth;
        this.expirationYear = expirationYear;
    }

    public String getCardType() {
        return cardType;
    }

    public void setCardType(String cardType) {
        this.cardType = cardType;
    }

    public String getExpirationMonth() {
        return expirationMonth;
    }

    public void setExpirationMonth(String expirationMonth) {
        this.expirationMonth = expirationMonth;
    }

    public String getExpirationYear() {
        return expirationYear;
    }

    public void setExpirationYear(String expirationYear) {
        this.expirationYear = expirationYear;
    }
}
```



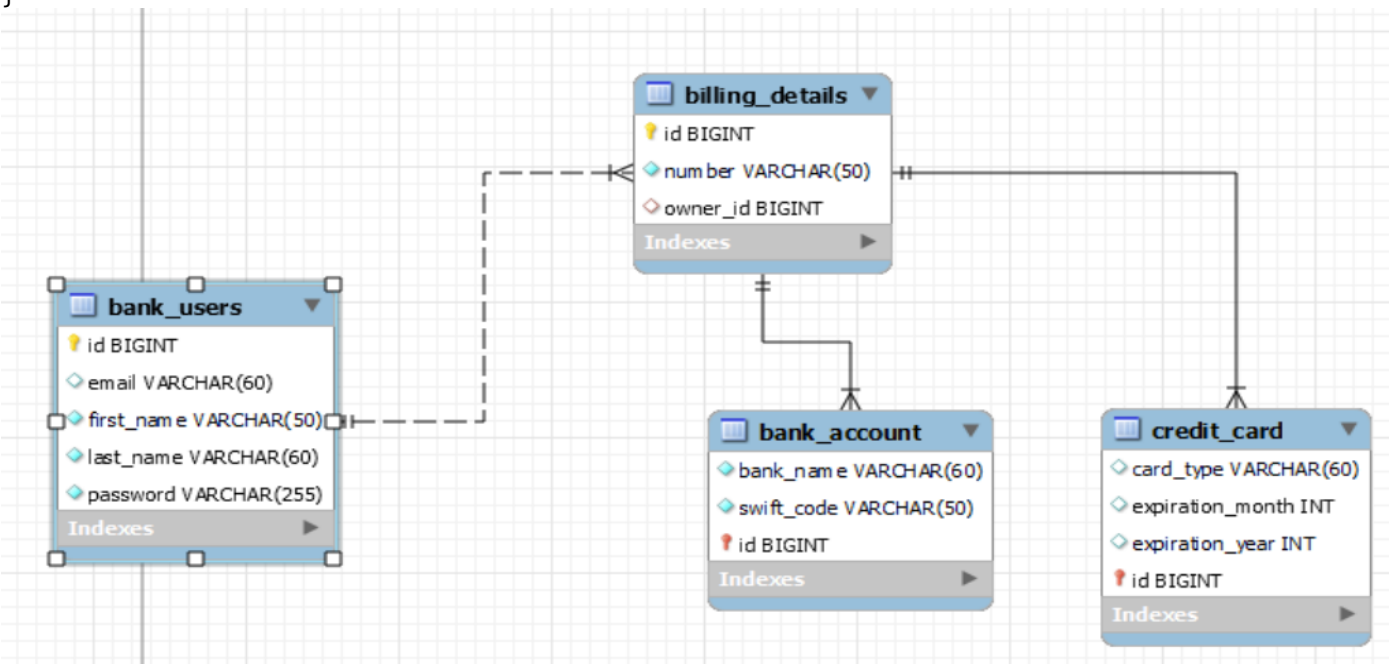
Пример 3 – отново Bills Payment System:

```
@MappedSuperclass //нещо междинно го анотира, няма таблица за SQL от този клас да има
public abstract class BaseEntity {
    private Long id;

    public BaseEntity() {
    }

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }
}


@Entity
@Table(name = "billing_details")
@Inheritance(strategy = InheritanceType.JOINED)
public abstract class BillingDetails extends BaseEntity {

    private String number;
    private BankUser owner;

    public BillingDetails() {
    }

    @Column(name = "number", length = 50, nullable = false, unique = true)
    public String getNumber() {
        return number;
    }
```

```
    public void setNumber(String number) {
        this.number = number;
    }

    @ManyToOne
    public BankUser getOwner() {
        return owner;
    }

    public void setOwner(BankUser owner) {
        this.owner = owner;
    }
}



@Entity
@Table(name = "credit_card")
public class CreditCard extends BillingDetails {
    private String cardType;
    private Integer expirationMonth;
    private Integer expirationYear;

    public CreditCard() {
        super();
    }

    @Column(name = "card_type", length = 60)
    public String getCardType() {
        return cardType;
    }

    public void setCardType(String cardType) {
        this.cardType = cardType;
    }

    @Column(name = "expiration_month")
    public Integer getExpirationMonth() {
        return expirationMonth;
    }

    public void setExpirationMonth(Integer expirationMonth) {
        this.expirationMonth = expirationMonth;
    }

    @Column(name = "expiration_year")
    public Integer getExpirationYear() {
        return expirationYear;
    }

    public void setExpirationYear(Integer expirationYear) {
        this.expirationYear = expirationYear;
    }
}


@Entity
@Table(name = "bank_account")
public class BankAccount extends BillingDetails{
    private String bankName;
    private String swiftCode;
```

```
    public BankAccount() {
        super();
    }

    @Column(name = "bank_name", length = 60, nullable = false)
    public String getBankName() {
        return bankName;
    }

    public void setBankName(String bankName) {
        this.bankName = bankName;
    }

    @Column(name = "swift_code", length = 50, nullable = false)
    public String getSwiftCode() {
        return swiftCode;
    }

    public void setSwiftCode(String swiftCode) {
        this.swiftCode = swiftCode;
    }
}


@Entity
@Table(name = "bank_users")
public class BankUser extends BaseEntity {

    private String firstName;
    private String lastName;
    private String email;
    private String password;

    public BankUser() {
    }

    @Column(name = "first_name", length = 50, nullable = false)
    public String getFirstName() {
        return firstName;
    }

    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    @Column(name = "last_name", length = 60, nullable = false)
    public String getLastName() {
        return lastName;
    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
    }

    @Column(name = "email", length = 60)
    public String getEmail() {
        return email;
    }

    public void setEmail(String email) {
```

```
        this.email = email;
    }

    @Column(name = "password", nullable = false)
    public String getPassword() {
        return password;
    }

    public void setPassword(String password) {
        this.password = password;
    }
}
```



5.8. Използване на Enum

```
package _00_EnumDemo;

public enum AccountType {
//ако тук разместим подредбата, да не ги взема по index, а по value!
    FREE("FREE"),
    SILVER("SILVER"),
    GOLD("GOLD"),
    TRIAL("TRIAL");

    private String value;

    AccountType(String value) {
        this.value = value;
    }

    public String getValue() {
        return this.value;
    }
}
```

```java
@Entity(name = "demos")
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;

    private String username;

//    @Enumerated(EnumType.ORDINAL) //по индекс
    @Enumerated(EnumType.STRING)  //по стойност value
    private AccountType type;

    public User() {
    }

public User(String username, AccountType type) {
    this.username = username;
    this.type = type;
}

public AccountType getType() {
    return type;
}
```

main

```java
User user = new User("Marina", AccountType.GOLD);
entityManager.persist(user);

User found = entityManager.find(User.class, 1);
System.out.println(found.getType().toString());
```

5.9. Default value

На wrapper класовете е null. – затова им задаваме тук `nullable = false`
На примитивния int е 0. – затова int никога не може да бъде NULL

6. HIBERNATE in more details + QUARKUS

https://quarkus.io/guides/hibernate-orm-panache
Quarkus support multiple DataSources, but Panache entities can be attached to only one persistence unit, so in order to be possible to use 2 databases (original and replica) we need to remove Panache and use JPA entities.

Common

Забелязах, че в курса java ee и тук в курса за Quarkus комбинираме валидации на колони в базата (JPA валидация) заедно с BeanValidation (javax.constraints). И пише, че са съвместими. В СофтУни ни учеха самия DTO обект да го валидираме с BeanValidation, а самото Entity с валидациите за базата данни. -
По принцип са съвместими, да. но препоръката е да се валидира на най-външния слой (в нашия случай DTO-то). Защото там, ако не мине валидацията, няма нужда да се навлиза по-навътре.

В базата (т.е. JPA) има смисъл да се слагат constraint-и, ако по някакъв начин стойностите, които се записват са резултат от изчисления идващи от две места (например ти се вика даден РЕСТ ресурс, ти в него викаш външен сървис, инпута от реста и резултата от външния сървис по някакъв начин ги комбинираш и ги пишеш в базата)!

Когато Update-ваме профил, как да го вкараме в Hibernate контекст

```
@Transactional
public void updateProfile(Profile profile) {
   Profile.getEntityManager().merge(profile);
}
```

Контролираме какво да ни върне. За разлика от namedQuery (and JPQL), които могат да ни върнат само цялото entity.

```
TypedQuery<Customer> typedQuery = em.createQuery(
"SELECT c FROM Customer c WHERE c.firstName = ?1", Customer.class);
typedQuery.setParameter(1, "Mike");
List<Customer> customers = typedQuery.getResultList();
```

JPQL и използване на entityManager

```
public List<String> getProfileIdsByBlocks(List<ProfileStatus> statuses) {
   EntityManager em = Profile.getEntityManager();
   TypedQuery<String> typedQuery = em.createQuery("SELECT cast(p.id as text) FROM Profile p WHERE
p.status IN (?1)", String.class);
   typedQuery.setParameter(1, statuses);
   return typedQuery.getResultList();
}
```

Прави същото като горе, но взема цялото entity

```
public static List<String> getProfileIdsByBlocks(List<ProfileStatus> statuses) {
   return Profile.<Profile>stream("status in (?1)", statuses).map(p ->
p.id.toString()).toList();
}
```

След като запишем в базата данни, то обекта musician автоматично приема и id-то от базата без да трябва да го присвояваме. А в Spring Data не е така - трябва да си вземем обекта от базата наново, за да го видим този обект и с id

```
Musician musician = new Musician();
musician.firstName = "Janis";
musician.lastName = "Joplin";
musician.dateOfBirth = LocalDate.of(1943, 01, 19);
musician.preferredInstrument = "Voice";

musician.persist(musician);

assertNotNull(musician.id);
```

No ONE-to-MANY direction, this is not a good direction to use

constructor - to have in mind which are the mandatory fields for initialization
json без дълги изречения на ключа
orElse веднага изпълнява
orElseGet -
dto - само на Rest слоя, entity само на service слоя

@Transactional е за сесията. Но когато четем от базата няма нужда от сесия. Има нужда от Hibernate сесия когато и от двете страни има комуникация.

Едно Dto с всички възможни варианти. Ако варианти липсват, то те не се дисплейват в Json-а и готово!
Единия вариант е да не ги сетваш, другия да не ги дърпаш от базата
Дисплейва каквото ти искаш, не каквото то реши

Dto-то задължително да има и Id (примерно това от базата) - голяма част от записите ги връщаме на други потребители и те трябва да знаят по какъв уникален код/цифра да ги достъпят - може да има Свилен Великов много потребители.

Третирай кода си с респект ако искаш и той да отвръща със същото!

https://vladmihalcea.com/the-best-way-to-use-the-manytomany-annotation-with-jpa-and-hibernate/

в @Transactional не е необходимо да persist-ваме вече активно managed bean entity
Представи си, че в един ресурс викаш 3 transactional операции, 2 се изпълняват, третата се чупи, и това което се случва е, че двете операции отиват в базата, а третата ревъртва, и си добавил данни в базата, които не искаш, ако имаш transactional на нивото на ресурса, счупи ли се една операция, всички се ревъртва.
Сега говорих с Ванката - каза ми, че може да оставя Service слоя - особено ако един и същи метод от Service слоя се вика на няколко Endpoint-а.

@Transactional не работи при @PostConstruct на Quarkus.
@PostConstruct търси inject-нат Bean за да работи

В общи линии дилемата е: 1. дали ще има повече връзки между таблиците, или 2. ще има по-малко връзки между таблиците, но с повече обхождания (join дето казваш)

Всички dto-а да са ми flat - без вложени обекти на dto-то - е flat object

SNAPSHOT vs RELATION

Това беше идеята - пазиш snapshot. когато става въпрос за snapshot, не правим релации, а пазим стойността в реферираща таблица.

Ако има само анотация @OneToMany (т.е. unidirectional), то се създава винаги нова join таблица с композитен ключ.

Ако има анотация само @ManyToOne (т.е. unidirectional), то не се създава нова помощна join таблица, а имаме foreign-key от към Many страната.

Ако връзката е Bi-directional - по дефаулт, то не се създава/ползва нова помощна join таблица

@ManyToOne - unidirectional

Demo code here - it is clear here

```
public class Student extends Person {
   public static final String GET_ALL_STUDENTS_FROM_A_KLASS_WITH_KLAS_INFO =
"getAllStudentsWithKlasInfo";
   public static final String GET_STUDENT_BY_EGN = "getStudentByEGN";


   @ManyToOne()
   private Klas klas;
```

@OneToMany - unidirectional

Прилагаме OneToFew все едно, т.е. когато имаме малко неща на many страната.
Прилагаме @OneToMany когато се очаква да го викаме/записваме от към One страната.
https://vladmihalcea.com/the-best-way-to-map-a-onetomany-association-with-jpa-and-hibernate/

В линка на горния ред елегантно се показва как можем да избегнем автоматичното генериране на трета таблица, но това важи само ако връзката е OneToMany. Т.е. ако едно ниво има 3 бонуса, а друго ниво има 5 бонуса като два от тях са като на предходното ниво, то тогава релацията е ManyToMany и генерирането на трета таблица е задължително (трета таблица значи ManyToMany).

OneToMany слага констраинт ако има трета таблица, то колоната от композитния ключ отговаряща за Many страната да бъде UNIQUE!!!

Също така когато имаме bidirectional и с анотация @OneToMany се бърка нещо JSONB при сериализирането на DTO-тата. И прави и втора табица

```
@Entity
@Table(name = "requests")
public class Request extends AbstractEntity {
    //Unidirectilan @OneToMany - the foreign key is in the target table RequestedMedicament using
@JoinColumn
    @OneToMany(cascade = CascadeType.ALL, orphanRemoval = true)
    @JoinColumn(name = "request_id")
    public List<RequestedMedicament> requestedMedicaments = new ArrayList<>();
```

Cascade - *The operations that must be cascaded to the **target** of the association. Т.е. Ако искаме всички операции - да записваме в базата или да трием, може да сетнем само на* CascadeType.*REFRESH*

```
@Entity
@Table(name = "Orders")
public class Order extends AbstractEntity {
    //Unidirectilan @OneToMany - the foreign key is in the target table RequestedMedicament using
@JoinColumn
    @OneToMany(cascade = CascadeType.ALL, orphanRemoval = true)
    @JoinColumn(name = "order_id")
    public List<RequestedMedicament> orderedMedicaments = new ArrayList<>();
```

```
@Entity
public class RequestedMedicament extends AbstractEntity {
    @ManyToOne(optional = false)
    public Medicament medicament;
```

WHERE ORDER BY id

	id 1	version	count	medicament_id	order_id	request_id
1	1	0	3	1	1	1
2	2	0	5	2	1	1
3	3	0	3	1	<null>	2
4	4	0	5	2	<null>	2
5	5	0	8	1	<null>	3

```
public Request(List<RequestedMedicament> requestedMedicaments, User user, Pharmacy pharmacy, String
patientMessage,
               LocalDateTime creationTime, LocalDateTime lastModified, RequestStatus status) {
    this.requestedMedicaments = requestedMedicaments;
    this.user = user;
    this.pharmacy = pharmacy;
    this.patientMessage = patientMessage;
    this.creationTime = creationTime;
    this.lastModified = lastModified;
    this.status = status;
}
```

@ManyToMany

https://vladmihalcea.com/the-best-way-to-use-the-manytomany-annotation-with-jpa-and-hibernate/

Без Cascade на mappedBy - от задната страна не знае какво да прави. Но ако сложим cascade и от задната страна, където е mappedBy, то можем да записваме и от двете страни - class.subects.add(newSubject)

Поради факта, че работим с колекция тип Set, то в нито една от междинните таблици CLAZZ_TEACHER, SUBJECT_TEACHER и SUBJECT_CLAZZ не можем да имаме повтаряща се двойка композитен primary key. Ако се опитаме да запишем повтаряща се двойка композитен primary key в междинната таблица CLAZZ_TEACHER, то няма да се породи нов запис с ново id, а първоначалната двойка ще си остане в базата (проверено).

```
clazz.teachers.add(teacher); //add if absent such relation
subject.teachers.add(teacher);//add if absent such relation
```

Когато имаме в заявките JOIN (LEFT JOIN), то таблицата се възпроизвежда x * y (x * y * z колкото join-а имаме) редове! Но като добавим клауза WHERE за всяка join-ата нормална таблица да намира един единствен нормален запис и поради факта, че работим със сетове и няма дублиращ се запис в междинните таблици, то винаги се връща списък от резултати, които не се дублират! (като математическа/логическа зависимост)
Т.е. няма нужда от DISTINCT!!!

```
@NamedQuery(query = "SELECT c FROM Clazz c LEFT JOIN c.teachers t WHERE t.id = ?1",
name = GET_CLAZZES_BY_TEACHER)

@NamedQuery(query = "SELECT c FROM Clazz c LEFT JOIN c.subjects sb LEFT JOIN
c.teachers t WHERE sb.id = ?1 AND t.id = ?2", name =
GET_CLAZZES_FROM_SUBJECT_BY_TEACHER)
```

Demo code here

```
@Entity
public class Subject extends AbstractEntity {
    @Column(unique = true)
    public String name;


    //!!! Merge-ваме добавяме елементи отттук, или с други думи:
    //добавяме на subject нови teachers
    //FRONT_SIDE of the BIDIRECTIONAL subjects <-> teachers
    @ManyToMany(cascade = {
            CascadeType.PERSIST,     добавяне на нови teachers оттук
            CascadeType.MERGE
    })
    @JoinTable(name = "subject_teacher",
            joinColumns = @JoinColumn(name = "subject_id", referencedColumnName = "id"),
            inverseJoinColumns = @JoinColumn(name = "teacher_id", referencedColumnName = "id")
    )
    private Set<Teacher> teachers = new LinkedHashSet<>();


    //!!! Merge-ваме добавяме елементи оттук, или с други думи:
    //добавяме на subject нови clazzes
    //FRONT_SIDE of the BIDIRECTIONAL subjects <-> clazzes
    @ManyToMany(cascade = {
            CascadeType.PERSIST,     добавяне на нови clazzes оттук
            CascadeType.MERGE
    })
    @JoinTable(name = "subject_clazz",
            joinColumns = @JoinColumn(name = "subject_id", referencedColumnName = "id"),
            inverseJoinColumns = @JoinColumn(name = "clazz_id", referencedColumnName = "id")
    )
    private Set<Clazz> clazzes = new LinkedHashSet<>();
```

```
}
```

```
@Entity
@Table(name = "clazz")
А ПО ТОЗИ НАЧИН ГЛАВНО
@NamedQuery(query = "SELECT DISTINCT c FROM Clazz c LEFT JOIN c.teachers t WHERE t.id = ?1", name =
GET_CLAZZES_BY_TEACHER)
@NamedQuery(query = "SELECT t FROM Teacher t LEFT JOIN t.clazzes c WHERE c.id = ?1 AND t.id = ?2",
name = GET_TEACHERS_FROM_CLAZZ_BY_TEACHER)
@NamedQuery(query = "SELECT c FROM Clazz c LEFT JOIN c.teachers t LEFT JOIN c.subjects s WHERE t.id
= ?1 AND s.id = ?2", name = GET_CLAZZES_BY_TEACHER_AND_SUBJECT)
public class Clazz extends AbstractEntity {
   public static final String GET_CLAZZES_BY_TEACHER = "Clazz.getClazzesByTeacher";
   public static final String GET_TEACHERS_FROM_CLAZZ_BY_TEACHER =
"Clazz.getTeachersFromClazzByTeacher";
   public static final String GET_CLAZZES_BY_TEACHER_AND_SUBJECT =
"Clazz.getClazzesByTeacherAndSubject";



   private String clazzNumber;
   private String subclazzInitial;


   //BACK_SIDE of the BIDIRECTIONAL subjects <-> clazzes
   @ManyToMany(mappedBy = "clazzes")
   private Set<Subject> subjects = new LinkedHashSet<>();



   //!!! Където декларираме така, оттам merge-ваме добавяме елементи, или с други думи:
   //добавяме на clazz teacher-и
   //FRONT_SIDE of the BIDIRECTIONAL clazzes <-> teachers
   @ManyToMany(cascade = {
           CascadeType.PERSIST,
//            CascadeType.MERGE  //TODO: !!!! за да може да инициализираме първоначалните данни
   })
   @JoinTable(name = "clazz_teacher",
           joinColumns = @JoinColumn(name = "clazz_id", referencedColumnName = "id"),
           inverseJoinColumns = @JoinColumn(name = "teacher_id", referencedColumnName = "id")
   )
   private Set<Teacher> teachers = new LinkedHashSet<>();
}
```

```
@Entity
public class Teacher extends Person {

   //BACK_SIDE of the BIDIRECTIONAL subjects <-> teachers
   @ManyToMany(mappedBy = "teachers")
   private Set<Subject> subjects = new LinkedHashSet<>();

   //BACK_SIDE of the BIDIRECTIONAL clazzes <-> teachers
   @ManyToMany(mappedBy = "teachers")
   private Set<Clazz> clazzes = new LinkedHashSet<>();
}
```

```
@RequestScoped
@Path(RESOURCE_PATH)
public class ClazzResource {
   public static final String RESOURCE_PATH = "clazz";
```

```
    /*
    {
        "clazzNumber": "3",
        "subclazzInitial": "A"
    }
     */
    @POST
    @Transactional
    @Consumes(MediaType.APPLICATION_JSON)
    public Response createClazz(@Valid @NotNull ClazzDTO clazzDTO) {
        Clazz clazz = clazzDTO.toClazz();
        clazz.persist();


        //тук не връщаме body - реално като сме поствали сме знаели какво създаваме и няма смисъл да
го връщаме
        return Response.created(URI.create(String.format("/%s/%d", RESOURCE_PATH,
clazz.getId()))).build();
    }

    @GET
    @Produces(MediaType.APPLICATION_JSON)
    public List<ClazzDTO> getAllClasses(@QueryParam("page") @DefaultValue("1") int page,
                                        @QueryParam("size") @DefaultValue("4") int size) {
        return Clazz.<Clazz>findAll()
                .page(page - 1, size)
                .stream()
                .map(ClazzDTO::new)
                .collect(Collectors.toList());
    }

    @GET
    @Path("name/{clazzName}")
    public Response getClazzByName(@ClazzName @PathParam("clazzName") String clazzName) {
        Pattern clazzPattern = Pattern.compile("(\\d+)([A-Z])+");
        Matcher clazzMatcher = clazzPattern.matcher(clazzName);
        if (clazzMatcher.find()) {
            String clazzNumber = clazzMatcher.group(1);
            String subclazzInitial = clazzMatcher.group(2);
            return Clazz.<Clazz>find("clazzNumber = ?1 and subclazzInitial = ?2", clazzNumber,
subclazzInitial)
                    .firstResultOptional()
                    .map(ClazzDTO::new)
                    .map(cdto -> Response.ok(cdto).build())
                    .orElseGet(() -> Response.status(Response.Status.NOT_FOUND).build());
        }

        return Response.status(Response.Status.BAD_REQUEST).build();
    }

    @PATCH
    @Transactional
    @Path("/{clazzId}/student")
    public Response addStudentToClazz(@NotNull @Positive @PathParam("clazzId") Long classId,
                                      @NotNull @Positive @QueryParam("studentId") Long studentId) {
        Clazz clazz = Clazz.findById(classId);
        Student student = Student.findById(studentId);

        //В Transactional минава през всичко - затова използваме if-else
        if (clazz == null || student == null) {
            return Response.status(Response.Status.NOT_FOUND).build();
        } else if (student.getClazz() != null) {
            //Check if student already has a clazz
            return Response.status(Response.Status.CONFLICT).build();
        } else {
            student.setClazz(clazz);
```

```
            return Response.status(Response.Status.NO_CONTENT).build();
        }
    }

@PATCH
@Transactional
@Path("{clazzId}/assign")
public Response assignTeacherToClazzAndSubject(@NotNull @Positive @PathParam("clazzId") Long
clazzId,
                                               @NotNull @Positive @QueryParam("teacherId") Long
teacherId,
                                               @NotNull @Positive @QueryParam("subjectId") Long
subjectId) {
    Clazz clazz = Clazz.findById(clazzId);
    Teacher teacher = Teacher.findById(teacherId);
    Subject subject = Subject.findById(subjectId);

    //В Transactional минава през всичко и взема последния return - затова връщаме накрая отговора
    Response response = null;
    if (clazz == null || teacher == null || subject == null) {
        response = Response.status(Response.Status.NOT_FOUND).build();
    } else {
        //Check if teacher already exists for that clazz
        //clazz.getTeachers().contains(teacher)
        boolean clazzExistsForThisTeacher = Clazz.<Clazz>find("#" +
GET_TEACHERS_FROM_CLAZZ_BY_TEACHER, clazzId, teacherId).firstResult() != null;
        boolean subjectExistsForThisClazz = Subject.<Subject>find("#" +
GET_CLAZZES_FROM_SUBJECT_BY_TEACHER, subjectId, clazzId).firstResult() != null;
        boolean subjectExistsForThisTeacher = Subject.<Subject>find("#" +
GET_TEACHERS_FROM_SUBJECT_BY_CLAZZ, subjectId, teacherId).firstResult() != null;

        if (clazzExistsForThisTeacher && subjectExistsForThisClazz && subjectExistsForThisTeacher) {
            response = Response.status(Response.Status.CONFLICT).build();
        } else {
            if (!clazzExistsForThisTeacher) {
                clazz.getTeachers().add(teacher); //Hibernate си знаел тук, не се натоварвало
            }

            if (!subjectExistsForThisClazz) {
                subject.getClazzes().add(clazz);  //Hibernate си знаел тук, не се натоварвало
            }

            if (!subjectExistsForThisTeacher) {
                subject.getTeachers().add(teacher);  //Hibernate си знаел тук, не се натоварвало
            }

            response = Response.status(Response.Status.NO_CONTENT).build();
        }
    }

    return response;
}

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

@RequestScoped
@Path(TEACHER_RESOURCE_PATH)
public class TeacherResource {

    public static final String TEACHER_RESOURCE_PATH = "teacher";

    /*
    {
        "firstName": "Marieta",
            "lastName": "Tabakova",
            "identity": "1111111111"
```

```
    }
     */
    @POST
    @Transactional
    @Consumes(MediaType.APPLICATION_JSON)
    public Response createTeacher(@Valid @NotNull TeacherDTO teacherDTO) {
        Teacher teacher = teacherDTO.toTeacher();
        teacher.persist();

        return Response.created(URI.create(String.format("/%s/%d", TEACHER_RESOURCE_PATH,
teacher.getId()))).build();
    }

    @GET
    @Produces(MediaType.APPLICATION_JSON)
    public List<TeacherDTO> getAllTeachers(@QueryParam("page") @DefaultValue("1") int page,
                                           @QueryParam("size") @DefaultValue("4") int size) {
        return Teacher.<Teacher>findAll()
                .page(page - 1, size)
                .stream()
                .map(TeacherDTO::new)
                .collect(Collectors.toList());
    }

    @GET
    @Path("identity/{identity}")
    @Produces(MediaType.APPLICATION_JSON)
    public TeacherDTO getTeacheByIdentity(@PathParam("identity") @NotBlank @IdentityGetExisting
String identity) {
        return Teacher.<Teacher>find("identity", identity)
                .firstResultOptional()
                .map(TeacherDTO::new)
                .orElse(null);
    }

    @GET
    @Path("{id}/clazzes")
    @Produces(MediaType.APPLICATION_JSON)
    public Response getTeacherClazzes(@PathParam("id") @Positive Long teacherId) {
        Teacher teacher = Teacher.findById(teacherId);
        if (teacher != null) {
            return
Response.ok(teacher.getClazzes().stream().map(ClazzDTO::new).collect(Collectors.toList())).build();
        } else {
            return Response.status(Response.Status.NOT_FOUND).build();
        }
    }

    @GET
    @Path("{id}/subjects")
    @Produces(MediaType.APPLICATION_JSON)
    public Response getTeacherSubjects(@PathParam("id") @Positive Long teacherId) {
        Teacher teacher = Teacher.findById(teacherId);
        if (teacher != null) {
            return
Response.ok(teacher.getSubjects().stream().map(SubjectDTO::new).collect(Collectors.toList())).build
();
        } else {
            return Response.status(Response.Status.NOT_FOUND).build();
        }
    }

    @GET
    @Path("{teacherId}/clazzes")
    @Produces(MediaType.APPLICATION_JSON)
    public Response getClazzesByTeacherAndSubject(@PathParam("teacherId") @Positive Long teacherId,
                                                  @QueryParam("subjectId") @Positive Long subjectId) {
        Teacher teacher = Teacher.findById(teacherId);
```

```
        Subject subject = Subject.findById(subjectId);

        //TODO: check here with SQL requests
//        boolean teacherIsTeachingThisSubject = teacher.getSubjects().contains(subject);
        if (teacher != null && subject != null) {
            List<Clazz> resultList = Clazz.<Clazz>find("#" + GET_CLAZZES_BY_TEACHER_AND_SUBJECT,
teacher.getId(), subject.getId()).list();
НЕ ПО ТОЗИ НАЧИН
//            Set<Clazz> teacherClazzes = teacher.getClazzes();
//            Set<Clazz> subjectClazzes = subject.getClazzes();
//
//            List<Clazz> clazzesByTeacherAndSubject = new ArrayList<>();
//            for (Clazz teacherClazz : teacherClazzes) {
//                if (subjectClazzes.contains(teacherClazz)) {
//                    clazzesByTeacherAndSubject.add(teacherClazz);
//                }
//            }

            return
Response.ok(resultList.stream().map(ClazzDTO::new).collect(Collectors.toList())).build();
        } else {
            return Response.status(Response.Status.NOT_FOUND).build();
        }
    }
}
```

Когато работим с Flyway

https://quarkus.io/guides/hibernate-orm

Виж какъв SQL код ще ти генерира Hibernate, и го напиши след това в нов sql init файл.
От тази настройка в application.properties файла казваме дали Hibernate да се изпълни -
quarkus.hibernate-orm.database.generation=none
quarkus.hibernate-orm.database.generation=drop-and-create

```
create table loyalty_level_bonus
(
    loyalty_level_id int8 not null,
    bonus_id         int8 not null,
    primary key (loyalty_level_id, bonus_id)
);

alter table if exists loyalty_level_bonus
    add constraint FKhwj377r8a7yhqs4g0gb310s8p
        foreign key (bonus_id)
            references bonus;

alter table if exists loyalty_level_bonus
    add constraint FK7rqql7u2lihp1p0vtuoe0w91g
        foreign key (loyalty_level_id)
            references loyalty_level;
```

За да работи програмата след като се инициализира горния код, то трябва да зададем имена на таблицата и колоните - понеже Hibernate генерира с други имена като е автоматично, А ние искаме определени по-подходящи имена.

```
@ManyToMany(
        cascade = {CascadeType.REFRESH}
)
@JoinTable(name = "loyalty_level_bonus",
        joinColumns = @JoinColumn(name = "loyalty_level_id", referencedColumnName = "id"),
        inverseJoinColumns = @JoinColumn(name = "bonus_id", referencedColumnName = "id")
```

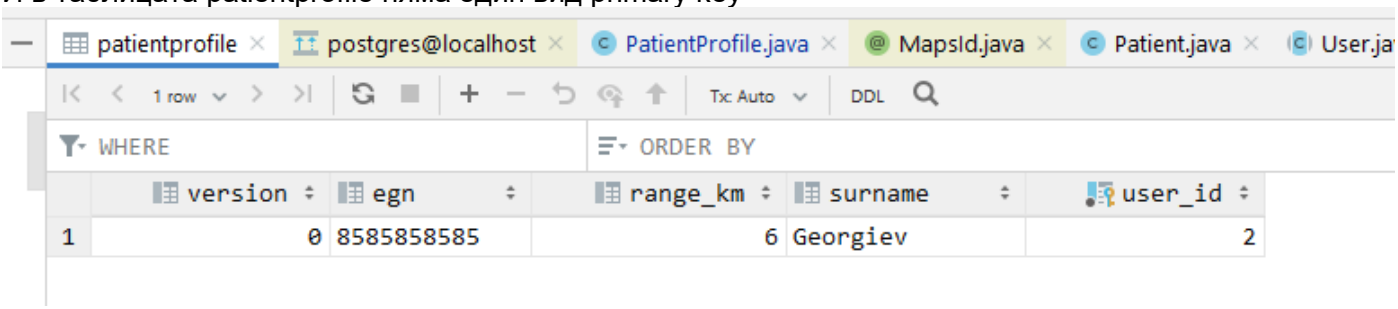
```
)
public Set<Bonus> bonuses = new LinkedHashSet<>();
```

@MapsId

https://vladmihalcea.com/the-best-way-to-map-a-onetoone-relationship-with-jpa-and-hibernate/

```
public @interface MapsId {

    /**
     * (Optional) The name of the attribute within the composite key
     * to which the relationship attribute corresponds.  If not
     * supplied, the relationship maps the entity's primary
     * key.
     */
  String value() default ""; }

@Entity
public class PatientProfile extends AbstractEntity {

    @MapsId    //PatientProfile primary key is the id (the primary key) of the Patient user
    @OneToOne
    public Patient user;

    //Range is a reserved value in MySQL
    @Column(name = "range_km")
    public Double range;

    public String surname;

    public String egn;
```

И в таблицата patientprofile няма един вид primary key



(Optional) The name of the attribute within the composite key to which the relationship attribute corresponds. If not supplied, the relationship maps the entity's primary key.

```
@MapsId(value = "email") //The field email of the Patient user
```

@ElementCollection

https://thorben-janssen.com/hibernate-tips-query-elementcollection/ - @ElementCollection

```
@Entity
@Table(name = "Users")
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
public abstract class User extends AbstractEntity {
    @Column(name = "first_name")
    public String firstName;

    @Column(name = "last_name")
```

```
    public String lastName;

    @Column(unique = true)
    public String email;
    public String password;
    public String salt;

    @Column(name = "preferred_locale")
    public String preferredLocale = "bg-BG";
```

Създава(използва/поддържа по-скоро вече съществуваща таблица за DML заявки) - вземи от таблицата user_roles всички роли за даден user.

```
    @ElementCollection(targetClass = UserRole.class)
    @JoinTable(name = "user_roles", joinColumns = @JoinColumn(name = "user_id"))
    @Column(name = "interest", nullable = false)
    @Enumerated(EnumType.STRING)
    public Set<UserRole> roles = new HashSet();

public enum UserRole {
    ADMIN,PATIENT,OPERATOR,PHARMACIST
```

@Convert

```
@Column(name = "payment_methods")
@Convert(converter = ListPaymentMethodsToStringConverter.class)
public List<PaymentMethod> paymentMethods;
```

```
/**
 * A class that implements this interface can be used to convert
 * entity attribute state into database column representation
 * and back again.
 * Note that the X and Y types may be the same Java type.
 *
 * @param <X>  the type of the entity attribute
 * @param <Y>  the type of the database column
 */
public interface AttributeConverter<X,Y> {


    /**
     * Converts the value stored in the entity attribute into the
     * data representation to be stored in the database.
     *
     * @param attribute  the entity attribute value to be converted
     * @return  the converted data to be stored in the database
     *          column
     */
    public Y convertToDatabaseColumn (X attribute);


    /**
     * Converts the data stored in the database column into the
     * value to be stored in the entity attribute.
     * Note that it is the responsibility of the converter writer to
     * specify the correct <code>dbData</code> type for the corresponding
     * column for use by the JDBC driver: i.e., persistence providers are
     * not expected to do such type conversion.
     *
     * @param dbData  the data from the database column to be
     *                converted
```

```
     * @return  the converted value to be stored in the entity
     *          attribute
     */
    public X convertToEntityAttribute (Y dbData);
```

```
public class ListPaymentMethodsToStringConverter implements AttributeConverter<List<PaymentMethod>,
String> {


    @Override
    public String convertToDatabaseColumn(List<PaymentMethod> list) {
        if (list == null || list.isEmpty()) {
            return null;
        }
        List<String> listOfStrings = list.stream().map(Enum::name).toList();
        return String.join(",", listOfStrings);
    }

    @Override
    public List<PaymentMethod> convertToEntityAttribute(String string) {
        if (string == null || string.isBlank()) {
            return Collections.emptyList();
        }
        return Stream.of(string.split(",")).map(PaymentMethod::valueOf).toList();
    }
}
```

@Column

Since, we have annotated the class with @Entity
If we remove @Column annotation from a field, then for String it takes the default value length 255.
If we use @Column annotation on a field without setting the length, then it again takes default length 255.
So, we should use @Column annotation and set the max length.
PostgresSQL type TEXT is 2^31-1, so we set the length of the above two columns to be Short.MAX_VALUE which is 2^15-1.

A columnDefinition="TEXT" няма смисъл да използваме - понеже SQL DefinitionL го прави ръчно.

@Column, though, is a JPA annotation that we use to control DDL statements only!!!

Това се използва само, когато разчиташ на Hibernate да създаде таблиците. Но когато таблиците са създадени (примерно от Flyway), тази анотация няма смисъл.

Аха. т.е. length за текстово поле взема default-на стойност (ако не е указано length) само за DDL операции. Т.е. при DML/ нов запис, няма да има проблеми

Точно така. ако искаш да имаш по време на нов запис, трябва да използваш bean validation анотациите

If we want to equalize the DDL with Hibernate to be exactly like the DDL with Flyway, then we can remove length, but use this: @Column(columnDefinition = "TEXT") and the default length will not be 255 characters but the POSTGRES length 2^31-1.

@PrePersist and @PreUpdate

https://nullbeans.com/how-to-use-prepersist-and-preupdate-in-jpa-hibernate/

```
@Entity
@Table(name = "customer")
```

```
public class Customer {

   private static final Logger log = LoggerFactory.getLogger(Customer.class.getName())
   private String type:

   ..... some code here ...

   @PrePersist
   private void prePersistFunction(){
  log.info("PrePersist method called");

      if(StringUtils.isEmpty(type)){
         type = "STANDARD_CUSTOMER";
      }
   }
```

@Index

Като заложим индекс на колона profile_id,

```
create index loyalty_achievement_profile_id_index on loyalty_achievement(profile_id);
```

То при заявка where profile_id = 10, самата база данни го търси по индекс

Можем да го направим и през Hibernate и Java кода:

```
@Entity
@Table(name = "loyalty_achievement",
indexes = @Index(name = "loyalty_achievement_profile_id_index", columnList =
"profile_id", unique = true))
```

index name
which column to be included in the index
is the index to be unique

BRIN in PostgreSQL

Подобно на B-Tree, GIN, GiST, Sp-GiST. Или как по-точно дадени заявки да работят по-бързо като придаваме определено желано поведение на индекса.
брин е някаква структура, в която се пазят данни.

create index if not exists game_record_brin_date_created_index on game_record **using BRIN** (date_created);
create index if not exists game_record_player_id_index on game_record (player_id);

@UniqueConstraint

Слагане на композитна уникалност ключ

```
alter table if exists Profile
   add constraint UKg5gaiumt unique (brand_id, email);
```

Можем да го направим и през Hibernate и Java кода:

```
@Entity
@Table(uniqueConstraints = {
       @UniqueConstraint(columnNames = {"brand_id", "email"})
})
```

@Embedded/@Embeddable

```
@Entity
@Table(name = "account_history")
public class AccountHistory extends PanacheEntity {
    @Embedded
    public Author author;

@Embeddable
@Access(AccessType.FIELD)  Could be added also
public class Author {
    @Column(name = "author_id")
    public UUID id;

    @Column(name = "author_username")
    public String username;
```

account_history

5 rows | Tx: Auto | DDL

WHERE | ORDER BY

		author_id	author_username	
1		3640cbd7-220f-426e-9ed3-c8a007c8ddb8	joseph.reid@example.org	<nu
2	NGE	9f859ee0-cf1e-4619-9ed5-85707077d53f	backofficeuser	Mar
3	NGE	<null>	"backofficeuser"	"Ma
4	NGE	9f859ee0-cf1e-4619-9ed5-85707077d53f	backofficeuser	"Ma
5	NGE	9f859ee0-cf1e-4619-9ed5-85707077d53f	backofficeuser	Mar

Cascading

```
@ManyToMany(
        cascade = {CascadeType.REFRESH}
)
@JoinTable(name = "loyalty_level_bonus",
        joinColumns = @JoinColumn(name = "loyalty_level_id", referencedColumnName = "id"),
        inverseJoinColumns = @JoinColumn(name = "bonus_id", referencedColumnName = "id")
)
public Set<Bonus> bonuses = new HashSet<>();
```

ВАЖНО!!!: таргета на cascade-а е таблицата `loyalty_level_bonus`. НЕ Е Bonus таблицата `bonus`

So EntityManager.refresh() is defined as :
Refresh the state of the instance from the database, overwriting changes made to the entity, if any.
So if entity A has a reference to entity B, and that reference is annotated with @CascadeType.REFRESH, and EntityManager.refresh(A) is called, then EntityManager.refresh(B) is implicitly called also.

I tested now, only the REFRESH cascading is doing almost everything - adding to the database, getting from the database, and also deleting from the database related entity records.

Merging detached entity into transaction

Направи ме активен/да бъда част от транзакцията

```
gameRecord.game = Game.getEntityManager().merge(gameRecord.game);
```

Projection in Quarkus with Hibernate

```
No @RegisterForReflection
public class ProfileConsents {
   public Long id;
   public boolean newsletterSubscribed;
   public CommunicationConsent communicationConsent;

   public ProfileConsents(Long id, boolean newsletterSubscribed, CommunicationConsent
communicationConsent) {
       this.id = id;
       this.newsletterSubscribed = newsletterSubscribed;
       this.communicationConsent = communicationConsent;
   }
}
```

```
io.quarkus.hibernate.orm.panache.PanacheQuery<Entity>.project()
```

```
public Optional<ProfileConsents> getProfileConsentsProjectedById(Long id) {
    return Profile.find("id", id).project(ProfileConsents.class).firstResultOptional();
}
```

`@RegisterForReflection` – ползва се когато пишем plugin-и например

Hmm, the annotation is used for native compilation of native applications. And we do not use such.

I thought the annotation helped preparing the SQL query as it takes static path `<project-name>.profile.model.projection.ProfileConsents` and also makes the ProfileConsentsProjected be able to be used for reflection. But all this seems to be done by Hibernate.

When removing the annotation, it works. :)

7. Spring Data Introduction

1. Inversion of Control принципа

Iversion of Control принципът се реализира по няколко начина в Spring без да пипаме source кода/компонентите. В контейнера има само interface-и на различните компоненти/:

Dependency Injection е първия подход

Annotation Driven

```
@Autowired   //Look up??? Не е -  прилагане на Inversion of control principle
```

With external XML file (Annotation Driven) Dependency Injection

Progamming (configuration class or with builder шаблон) Dependency Injection

Look up е втория подход

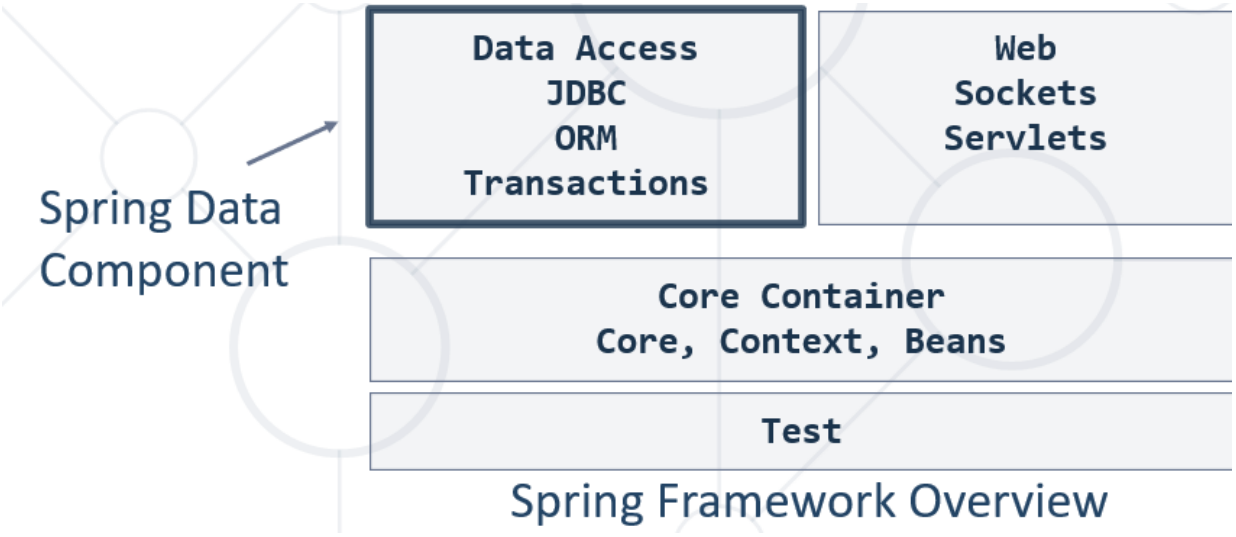
С някакво минимално API да намираме необходимите Beans по техния интерфейс.

2. Framework

- Platform for **developing software applications**
- Provides a **foundation** on which **software developers** can **build programs** for a **specific platform**
- Similar **to an API**
  - A Framework **includes an API**
- May include **code libraries**, a **compiler**, and other programs **used in the software development process**

3. Revision of Spring

- Spring Application framework for the Java Platform
  - Technology stack - includes several modules that provide a range of services



Beans and Context

Обикновени JAVA обекти/елементи, които са анотирани, за да могат да бъдат менъджирани от framework-а в техния жизнен цикъл.

Чрез scope анотация, можем да дефинираме типа на Bean-а или неговия context!

Singleton контекст на Bean-а – само по една инстанция, по default са Singleton обикновено

Prototype контекст на Bean-а – всеки път да се вика нова инстанция

В Web частта има още

Request контекст

Сами по себе си REST заявките са state-less

Session контекст – примера с пазаруването от сайта

Най-често чрез cookies

и

Application context

Web контекст – специфични за различни технологии

Servlet – класическата JAVA web архитектура е базирана върху тези Servlet технологии

GET, POST, PUT, DELETE, options, заявки

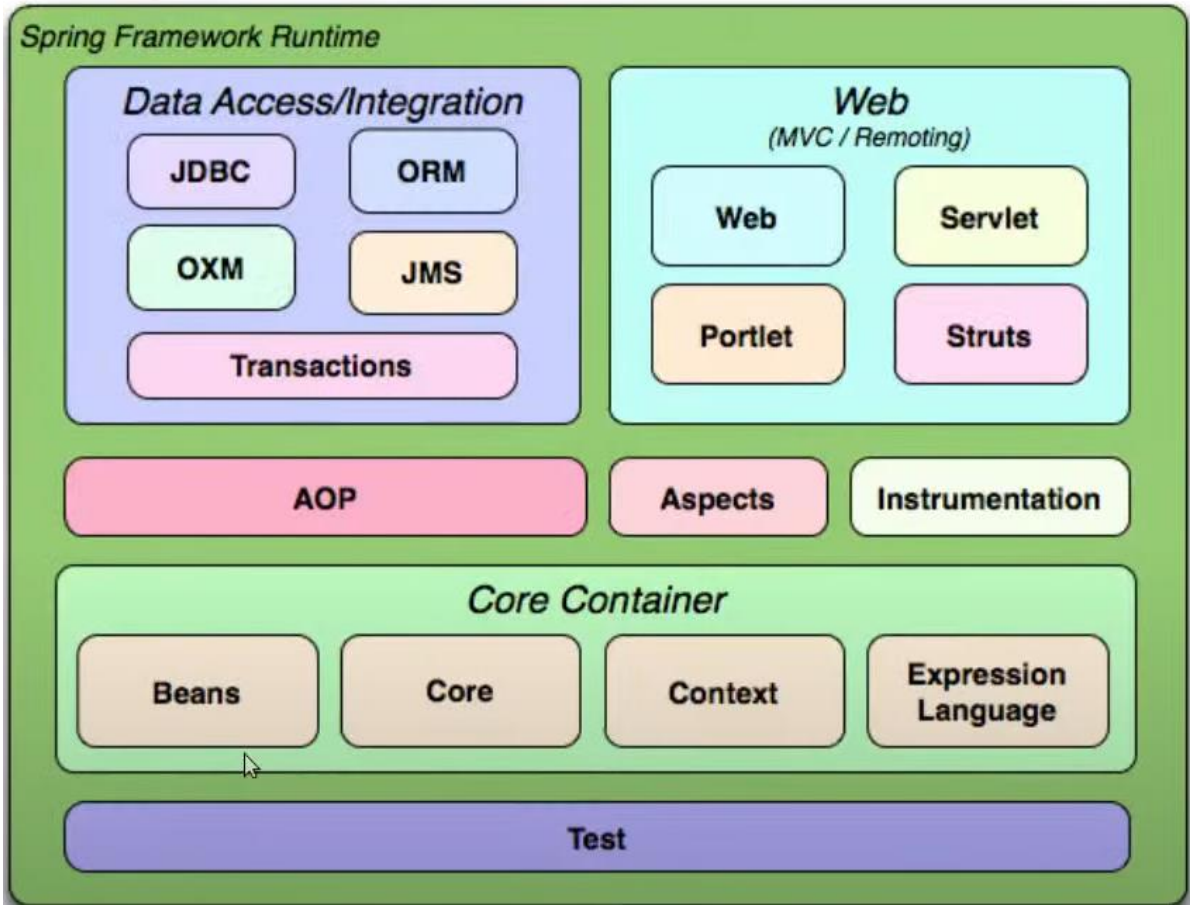
Sockets – bi-directional real time servlets

Asynchronous architecture with React Api (with Flux functional programming for example)

Spring Platform

Spring Platform

- Spring makes **programming Java quicker, easier, and safer(ще ни предпази от грешки) for everybody**
- Spring's **focus is on speed, simplicity, and productivity** built by multiple Spring Projects
  - Spring **Boot**
  - Spring **Framework**
  - Spring **Data**



Spring Module

- Spring **Core Container**
  - The base module of Spring and provides Spring containers
- **Aspect-Oriented Programming**
  - Enables implementing cross-cutting concerns

- **Authentication and Authorization**

- **Data Access**
  - Working with **RDBMS using JDBC and ORM tools**
- **IoC Container(Inversion of control Containers)**
  - Configuration of application **components and lifecycle management of Java objects**, done mainly via **dependency injection**
- **Testing**
  - Support classes for writing **unit tests and integration tests**

Spring Projects

Spring Boot

- Makes it easy to **create stand-alone**, **production-grade Spring based Applications**
- Бял лист с нарисувано първата картинка. Докато още се уча – ползвам книжката за оцветяване

Spring Framework

- Provides a **comprehensive programming and configuration model** for modern Java-based enterprise applications - on any kind of deployment platform
- **Абсолютно бял лист**
- **Open-Source Application framework** and **inversion of control container** for the Java platform
- Core features can be used by any Java application **extensions** for building web applications **on top of the Java EE specification**
- **Spring е имплементация на Java EE specification**
- **Абсолютно бял лист**

Spring Data

- Spring Data's mission is to **provide a familiar and consistent**, Spring-based **programming model for data access** while still retaining the special traits of the underlying data store

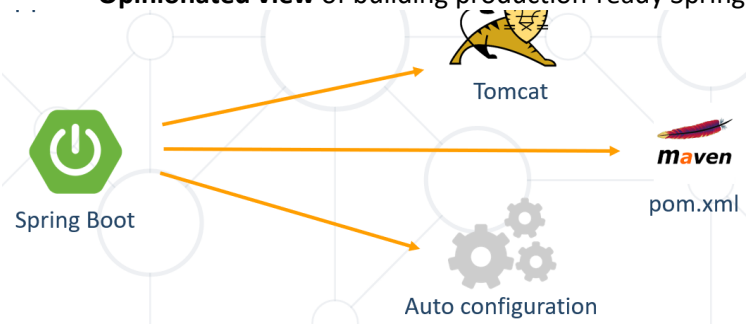
Spring Cloud

- Spring Cloud **provides tools for developers to quickly build** some of the **common patterns in distributed systems**

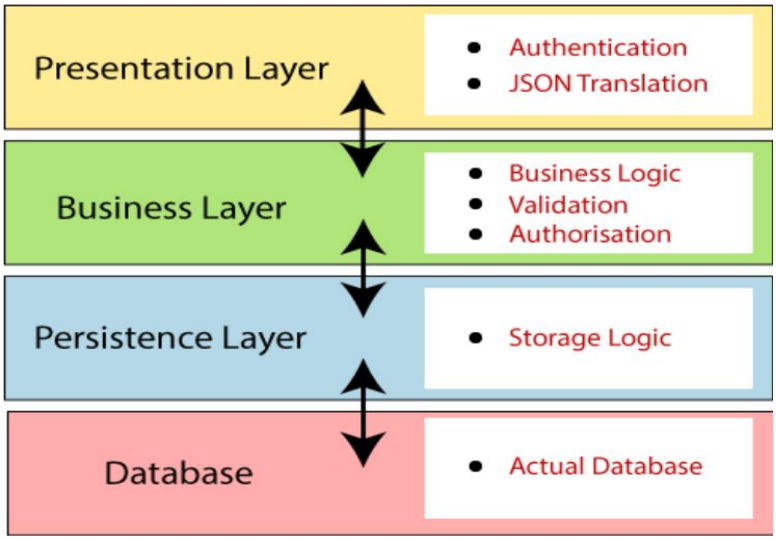
4. Spring Boot

Architecture

- **Opinionated view** of building production-ready Spring applications



https://www.javatpoint.com/spring-boot-architecture

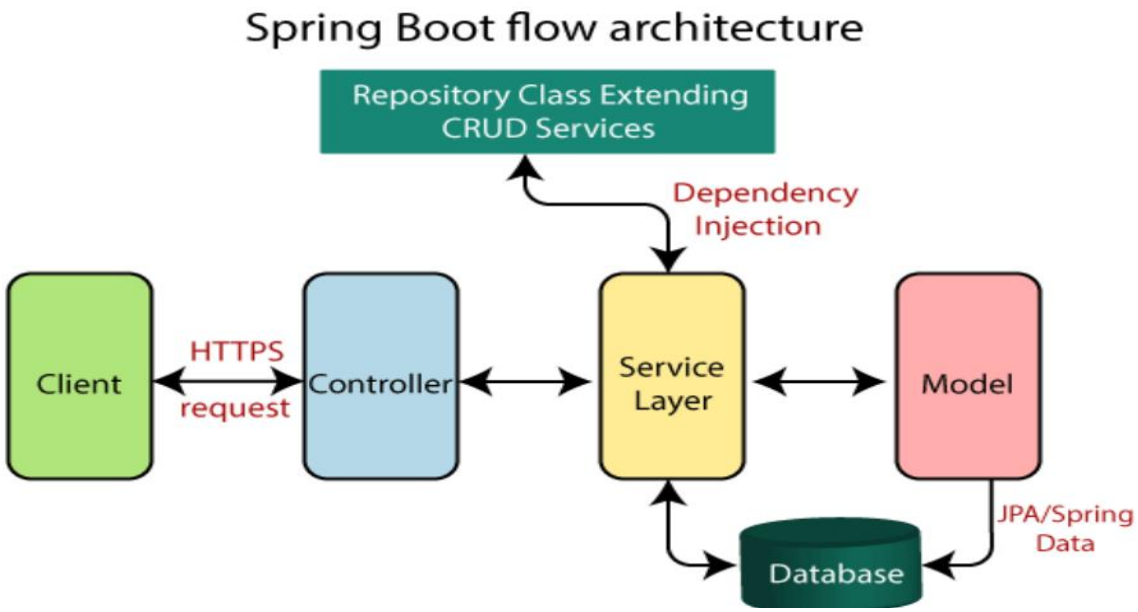


Presentation Layer: The presentation layer handles the HTTP requests, translates the JSON parameter to object, and authenticates the request and transfer it to the business layer. In short, it consists of **views** i.e., frontend part.

Business Layer: The business layer handles all the **business logic**. It consists of service classes and uses services provided by data access layers. It also performs **authorization** and **validation**.

Persistence Layer: The persistence layer contains all the **storage logic** and translates business objects from and to database rows.

Database Layer: In the database layer, **CRUD** (create, retrieve, update, delete) operations are performed.



Advantage of Convention Over Configuration – предимство на конвенция пред конфигурация

- Creates stand-alone Spring applications
  - Provide opinionated 'starter' POMs to simplify your Maven configuration
- Automatically configure Spring whenever possible

- Absolutely no code generation and no requirement for XML configuration

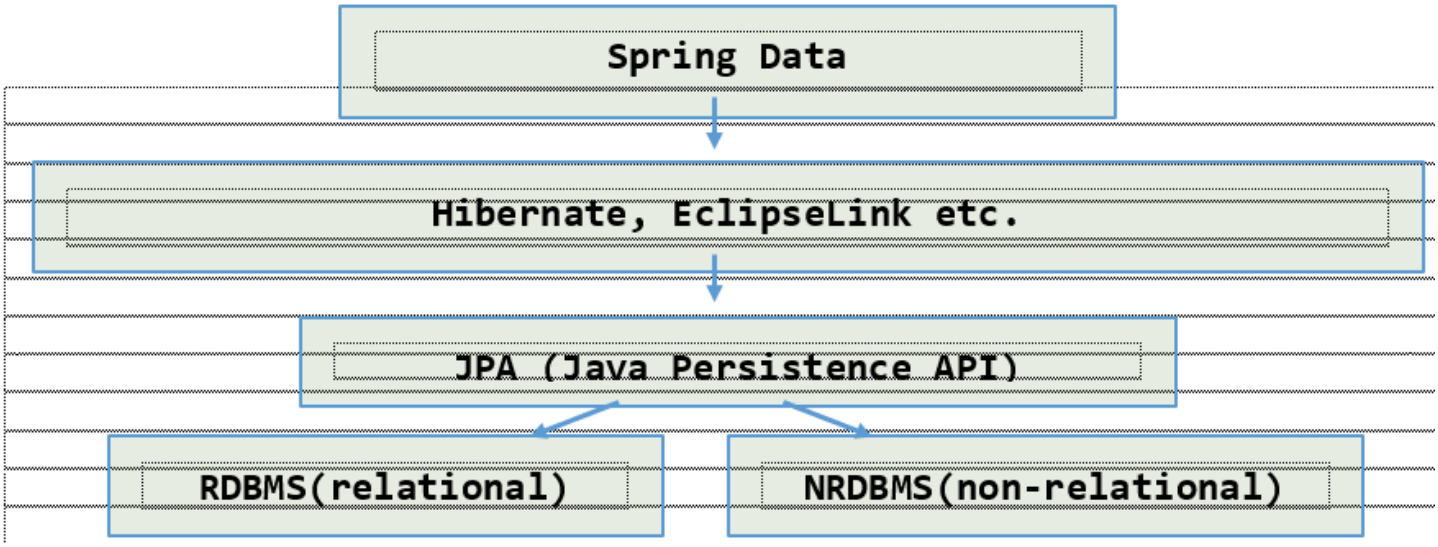
5. Spring Data

What is Spring Data

- Library that adds an **extra layer of abstraction** on the top of our **JPA** (Java Persistence API (Application Programming Interface)) provider
- Provides:
  - Dynamic query derivation from repository method names
  - Possibility to integrate custom repositories and many more
- What Spring Data is not:
  - **Spring Data JPA is not a JPA provider**

Spring Data Role

Extra layer of abstraction over the used ORM



Dependencies in Maven

pom.xml

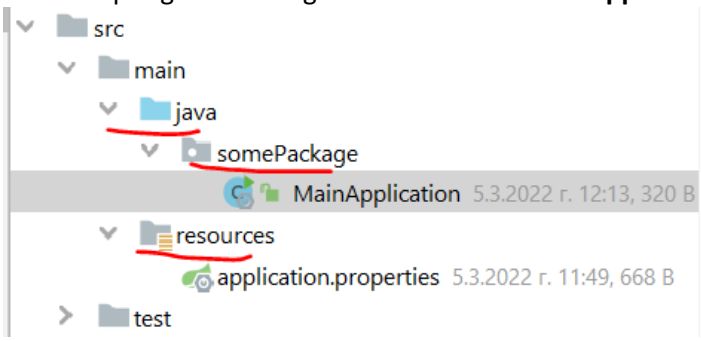
```
<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-compiler-plugin</artifactId>
            <version>3.6.1</version>
            <configuration>
                <source>14</source>
                <target>14</target>
            </configuration>
        </plugin>
    </plugins>
</build>
```

```
<parent>  //служи за вземане на базови настройки на базов pom.xml файл. На базовия има по-базов,
```

```
и там има persistence
    <groupId>org.springframework.boot</groupId>
    <version> 2.3.8.RELEASE</version>
    <artifactId>spring-boot-starter-parent</artifactId>
</parent>


<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-jpa</artifactId> //тук са ни оставили опция какво да
ползваме, може да не е ORM data-jpa, а JDBC примерно, или пък non-relational db
        <version>2.6.3</version>
    </dependency>

    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <version>8.0.28</version>
        <scope>runtime</scope>
    </dependency>
</dependencies>
```

Configuration

- Spring boot configurations are held in an **application.properties** file

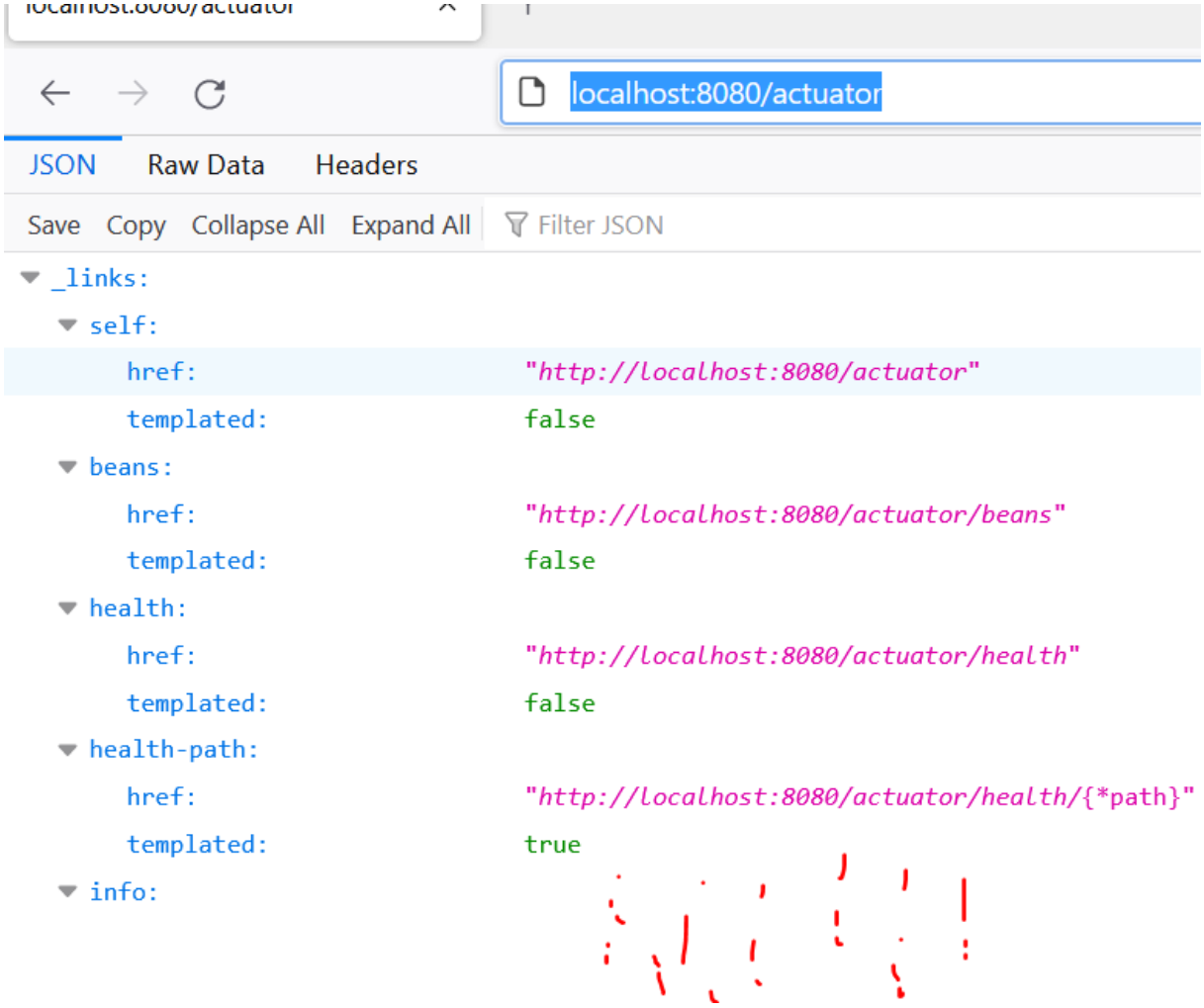


application.properties

```
#Data Source Properties
spring.datasource.driverClassName = com.mysql.cj.jdbc.Driver

spring.datasource.url = jdbc:mysql://localhost:3306/school?useSSL=false
Или
spring.datasource.url =
jdbc:mysql://localhost:3306/bookshop?createDatabaseIfNotExist=true&useSSL=false

spring.datasource.username = root
spring.datasource.password =
spring.datasource.password =  12345


Изпълняване на SQL заявки при стартиране (execute SQL / run SQL automatically)
#spring.datasource.initialization-mode = always
#spring.sql.init.mode = always

#spring.datasource.data=classpath:insert-data.sql
#spring.sql.init.data-locations=classpath:insert-data.sql
```

```
#JPA Properties
spring.jpa.properties.hibernate.dialect = org.hibernate.dialect.MySQL8Dialect
spring.jpa.properties.hibernate.show_sql = TRUE
spring.jpa.properties.hibernate.format_sql = TRUE
spring.jpa.properties.hibernate.use_sql_comments = TRUE


spring.jpa.hibernate.ddl-auto = create-drop

###Logging Levels
# Disable the default loggers – двете долу ако са активирани, скриват доста от default логовете
logging.level.org = WARN //контролираме логовете
logging.level.blog = WARN //контролираме логовете

#Show SQL executed with parameter bindings
logging.level.org.hibernate.SQL = DEBUG
logging.level.org.hibernate.type.descriptor = TRACE

# Spring actuator settings - service monitoring metrics
#management.endpoints.web.base-path=/
management.endpoints.web.exposure.include=info, health, httptrace, metrics, threaddump, heapdump,
shutdown, beans
management.trace.http.enabled=true
management.endpoint.health.show-details=always
info.app.name=Article repository application
info.app.description=This sample application allows to manage articles and users
info.app.version=1.0.0


#Change server port
#server.port=8000
```

http://localhost:8080/actuator



main – задължително си правим някаква папка/пакет и в него хакаме MainApplication

```
package somePackage;

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

@SpringBootApplication
public class MainApplication {
    public static void main(String[] args) {
        SpringApplication.run(MainApplication.class,args);
    }
}
```

6. JdbcTemplate from Spring Jdbc

pom.xml

```
<dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-jpa</artifactId> //тук са ни оставили опция какво да ползваме, може да не е ORM data-jpa, а JDBC примерно, или пък non-relational db
</dependency>

<dependency>
        <groupId>com.mysql</groupId>
        <artifactId>mysql-connector-j</artifactId>
```

```
        <scope>runtime</scope>
</dependency>
--------------

package bg.jug.academy.DataWithSpringJdbc;

import bg.jug.academy.DataWithSpringJdbc.module.EmployeeEntity;
import bg.jug.academy.DataWithSpringJdbc.module.EmployeeMapper;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.CommandLineRunner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.simple.SimpleJdbcInsert;

import java.util.List;
import java.util.Map;

@SpringBootApplication
public class DataWithSpringJdbcApplication implements CommandLineRunner {

    @Autowired
    private JdbcTemplate jdbcTemplate; //from spring-jdbc library = org.springframework.jdbc.core

    @Autowired
    private EmployeeMapper employeeMapper;

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

    @Override
    public void run(String... args) throws Exception {
        Integer employeesCount = countEmployees();
        EmployeeEntity employeeById = getEmployeeById("1");
        List<EmployeeEntity> employees = getEmployees();

        insertEmployee();
    }

    private void insertEmployee() {
        SimpleJdbcInsert simpleJdbcInsert = new SimpleJdbcInsert(jdbcTemplate);
        simpleJdbcInsert.setTableName("Employees");
        simpleJdbcInsert.execute(Map.of("id", "120", "name", "Ivanov", "salary", 2000L));

        jdbcTemplate.update("UPDATE Employees SET name = 'Petrov' WHERE id=120");
    }

    private Integer countEmployees() {
        return jdbcTemplate.queryForObject("SELECT count(*) FROM employees", Integer.class);
    }

    private EmployeeEntity getEmployeeById(String id) {
//        return jdbcTemplate.queryForObject("SELECT id, name, salary from hrm.employees where id =
%s".formatted(id), new EmployeeMapper()); //queryForObject(String sql, RowMapper<T> rowMapper)
        return jdbcTemplate.queryForObject("SELECT id, name, salary from hrm.employees where id =
%s".formatted(id), employeeMapper); //queryForObject(String sql, RowMapper<T> rowMapper)
    }

    private List<EmployeeEntity> getEmployees() {
//    jdbcTemplate.queryForList("SELECT id, name, salary from hrm.employees");
//        return jdbcTemplate.query("SELECT id, name, salary from hrm.employees", new
EmployeeMapper());  //query(String sql, RowMapper<T> rowMapper)
```

```
        return jdbcTemplate.query("SELECT id, name, salary from hrm.employees",
employeeMapper);  //query(String sql, RowMapper<T> rowMapper)
    }
}
```

—----------

```
import org.springframework.jdbc.core.RowMapper;
import org.springframework.stereotype.Component;

import java.sql.ResultSet;
import java.sql.SQLException;

@Component
public class EmployeeMapper implements RowMapper<EmployeeEntity> {
    @Override
    public EmployeeEntity mapRow(ResultSet rs, int rowNum) throws SQLException {
        return new EmployeeEntity(rs.getString("id"), rs.getString("name"), rs.getLong("salary"));
```

—----------

```
import lombok.AllArgsConstructor;
import lombok.Getter;
import lombok.NoArgsConstructor;
import lombok.Setter;

@Getter
@Setter
@AllArgsConstructor
@NoArgsConstructor
public class EmployeeEntity {
    private String id;
    private String name;
    private Long salary;
}
```

7. Spring Data Repositories

Spring Repository – вдигане на абстракцията още

Repository-то ни дава всичко, което работи с базата

- Abstraction to significantly reduce the amount of boilerplate code required to implement data access layers
    - **Perform CRUD Operations**
    - **Automatically generates JPQL/SQL code**
    - Highly customizable

Built-in CRUD Operations

Едно такова repository ще отговаря за един клас

```
JPA REPOSITORY
- <S extends T> S save(S var1);
- <S extends T> Iterable<S>
save(Iterable<S> var1);
- T findOne(ID var1);
- boolean exists(ID var1);
- Iterable<T> findAll();
- long count();
- void delete(ID var1);
void deleteAll();
…
```

8. Spring Data Query Creation

Query Creation

- Queries are created via a query builder mechanism built into Spring Data
  - Strips the prefixes like **find…By**, **read…By**, **query…By** and starts parsing the rest of it
- Spring Data JPA will do a property check and traverse nested properties
- **Спираме да пише SQL/JPQL заявки**

Custom CRUD Operations

//Custom method

```
@Repository
public interface StudentRepository extends CrudRepository<Student, Long> {
    List<Student> findByMajor(Major major);
}
```

```
@Repository
public interface StudentRepository extends JpaRepository<Student, Long> {
    List<Student> findByMajor(Major major);
}
```

```
SQL
SELECT s.*
  FROM students AS s
 INNER JOIN majors AS m
    ON s.major_id = m.id
 WHERE m.id = ?
```

Query Lookup Strategies

https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#jpa.query-methods.query-creation

Keyword	Sample	JPQL
And	findByLastnameAndFirstName	… where x.last_name = ?1 and x.firstname = ?2
Or	findByLastnameOrFirstname	… where x.lastname = ?1 or x.firstname = ?2
Between	findByStartDateBetween	… where x.startDate between 1? and ?2
LessThan	findByAgeLessThan	… where x.age < ?1
Containing	findByFirstnameContaining	… where x.firstname like ?1 (par ameter bound wrapped in %)
In	findByAgeIn(Collection<Age> ages)	… where x.age in ?1

9. Spring Data Services - Encapsulating Business Logic

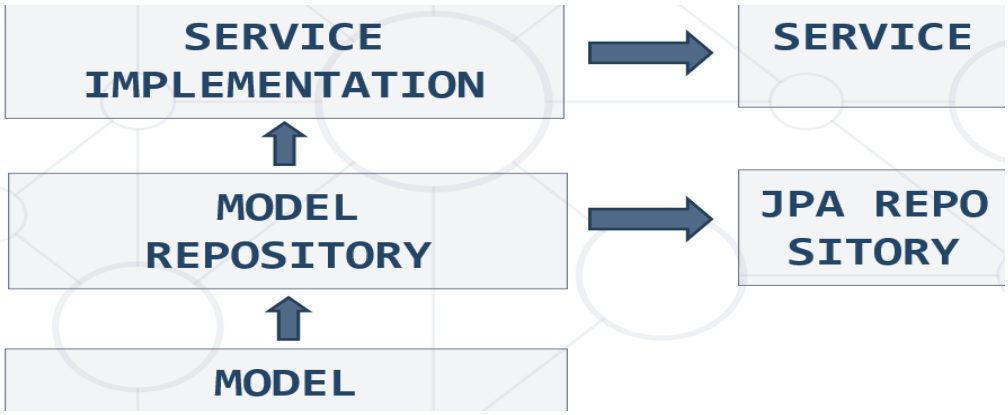
Service-а ни дава всичко, което е бизнес логика след като сме взели данните от базата данни.
Service-а използва Repository-та, които от своя страна си говорят с базата данни.

Service Pattern

- Service Layer is a design pattern of organizing business logic into layers
  - Service classes are categorized into a particular layer and share functionality
- Main concept is **not exposing details** of internal processes on entities
  - Services **interact closely** with Repositories

In Spring Data Framework, the usage of **`@Service`**, **`@Repository`** or **`@Component`** annotations is needed to separate different "**layers**" in the application. They are mainly used for programmers to know a class's role and which logical layer it belongs to.
The **`@Autowired`** annotation is required when **injecting a resource**, e.g., **Repository** to **Service** or **Service to Component.**

Spring Data Architecture



Services

```
public interface AccountService {
    void withdrawMoney(BigDecimal amount, Long id);

    void depositMoney(BigDecimal amount, Long id);
}

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import softuni.repositories.AccountRepository;
@Service("AccountServ")  //можем да му зададем име на service-а в Spring-а да бъде bean, но
реално не го използваме
/**
 * The value may indicate a suggestion for a logical component name,
 * to be turned into a Spring bean in case of an autodetected component.
 * @return the suggested component name, if any (or empty String otherwise)
 */
@AliasFor(annotation = Component.class)
String value() default "";

public class AccountServiceImpl implements AccountService {
    private final AccountRepository accountRepository;

    @Autowired – през конструктора
    public AccountServiceImpl(AccountRepository accountRepository) {
        this.accountRepository = accountRepository;
    }

    @Override
    public void withdrawMoney(BigDecimal amount, Long id) {    }

    @Override
    public void depositMoney(BigDecimal amount, Long id) {    }
}
```

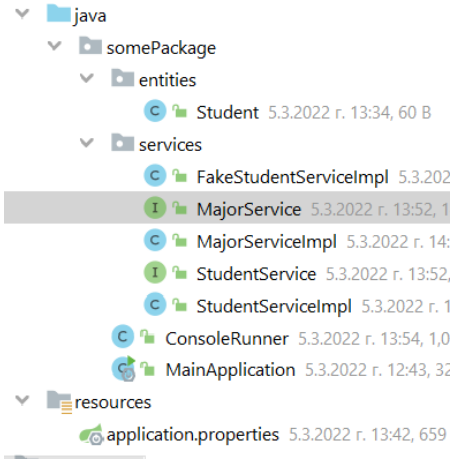
Освен с **@Autowired**, можем да ползваме **@Inject** и **@Resource**, но при @Autowired можем да ползваме Optional.
Optional<String> name = …
String str = name.get(); //взема резултата ако го има

Entry Point

main – задължително си правим някаква папка/пакет и в него хакаме MainApplication

```
package somePackage;

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

@SpringBootApplication
public class MainApplication {
    public static void main(String[] args) {
        SpringApplication.run(MainApplication.class,args);
    }
}
```

10. Command Line Runner




```java
public interface StudentService {
    void register();
}

public interface MajorService {
    void register();
}

import org.springframework.stereotype.Service;

@Service
public class StudentServiceImpl implements StudentService {
    @Override
    public void register() {
//        studentRepository.register(); //благодарение на Спринг, пропускаме този ред да го пишем
    }
}

import org.springframework.stereotype.Service;

@Service
public class MajorServiceImpl implements MajorService {
    @Override
    public void register() {
  //  studentRepository.register();  //благодарение на Спринг, пропускаме този ред да го пишем
    }
}



import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.CommandLineRunner;
import org.springframework.stereotype.Component;
import somePackage.services.MajorService;
import somePackage.services.StudentService;

@Component
public class ConsoleRunner implements CommandLineRunner {
//    @Autowired
    private StudentService studentService;

//    @Autowired
    private MajorService majorService;

    @Autowired  //може и през конструктора да го auto-wire-нем
```

```
    public ConsoleRunner(StudentService studentService, MajorService majorService) {
        this.studentService = studentService;
        this.majorService = majorService;
    }

    @Override
    public void run(String... strings) throws Exception {
        System.out.println("ha ha ha");
        this.studentService.register();
        this.majorService.register();
//        Major major = new Major("Java DB Fundamentals");
//        Student student = new Student("John",new Date(), major);
//        majorService.create(major);
//        studentService.register(student);
    }
}
```

11. ApplicationRunner вместо CommandLineRunner interface

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.ApplicationArguments;
import org.springframework.boot.ApplicationRunner;
import org.springframework.stereotype.Component;


@Component
public class AppRunner implements ApplicationRunner {
    private final SeedService seedService;
    private final BookRepository bookRepository;
    private final AuthorRepository authorRepository;

    @Autowired
    public AppRunner(SeedService seedService, BookRepository bookRepository, AuthorRepository
authorRepository) {
        this.seedService = seedService;
        this.bookRepository = bookRepository;
        this.authorRepository = authorRepository;
    }

    @Override
    public void run(ApplicationArguments args) throws Exception {
        System.out.println("Starting");
//        this.seedService.seedAuthors();
//        this.seedService.seedCategories();
//        this.seedService.seedAll();

//        this._01_booksAfter2000();
//        this._02_allAuthorsWithBookBefore1990();
//        this._03_allAuthorsOrderedByBookCount();
        this._04_GetAllBooksByAuthorname();
    }
```

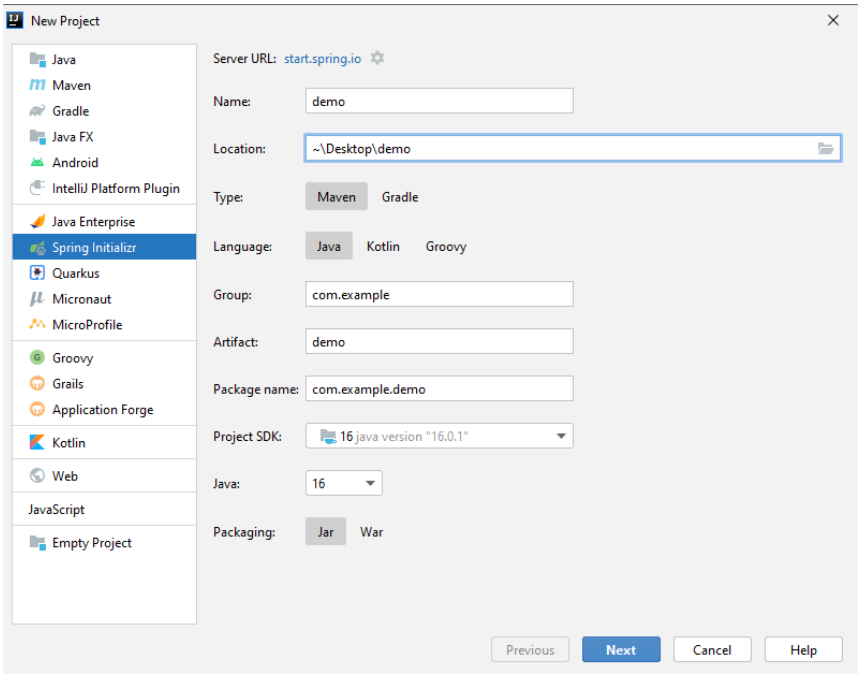
12. Creating project via start.spring.io

start.spring.io

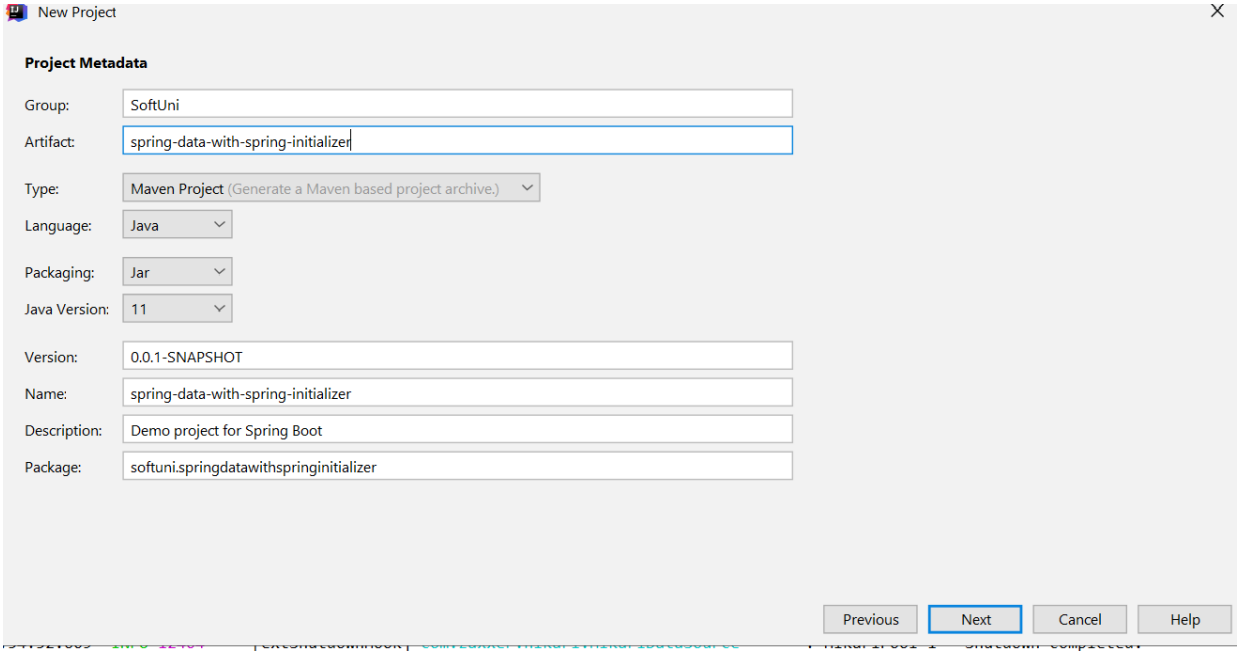
Избираме си от сайта какво искаме – горе долу същите неща като през IntelliJ

13. Creating project via Spring initializer (in IntelliJ Iltimate or a plug-in should be installed)

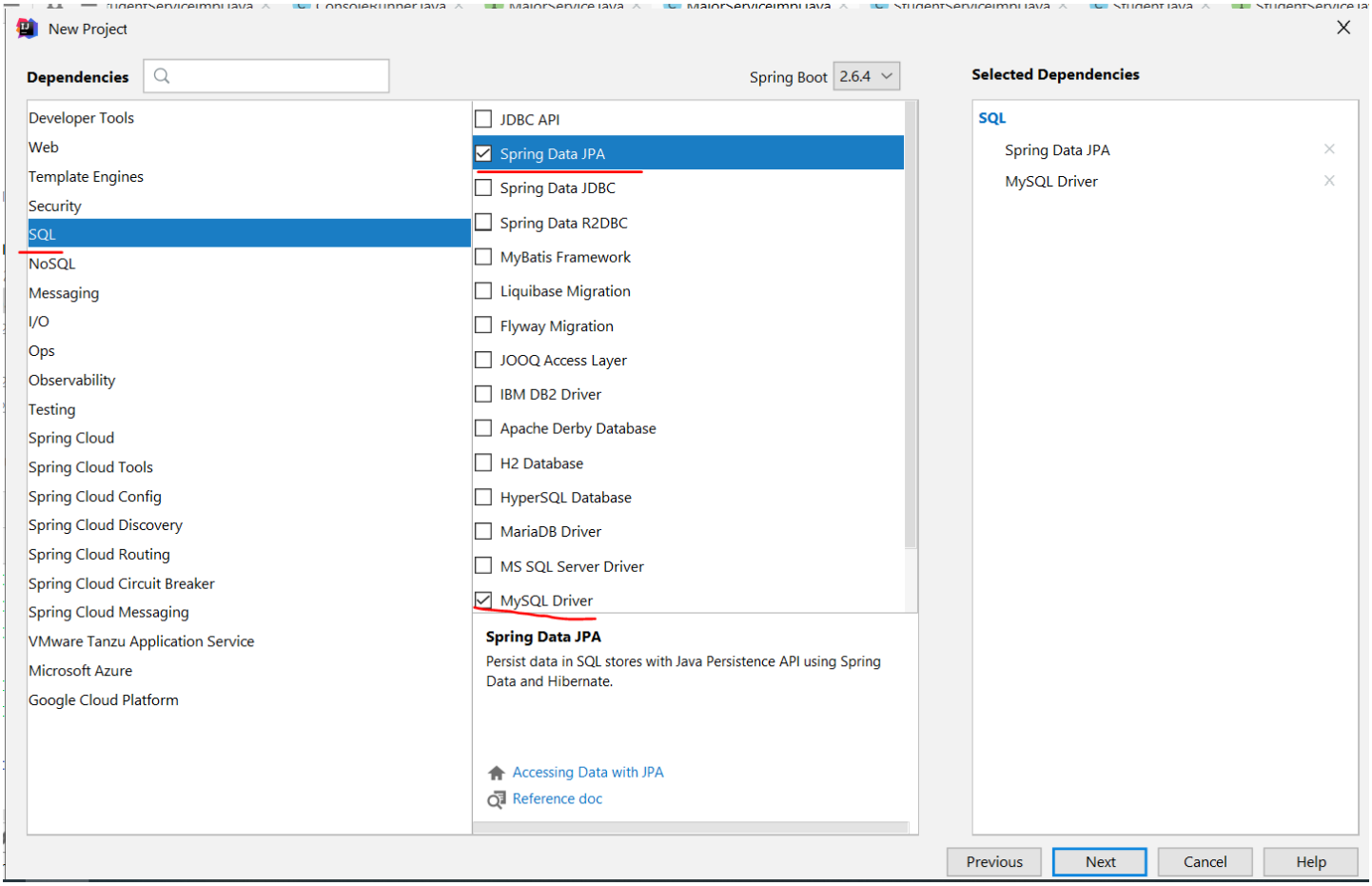
(not Maven, but it is like Maven again)



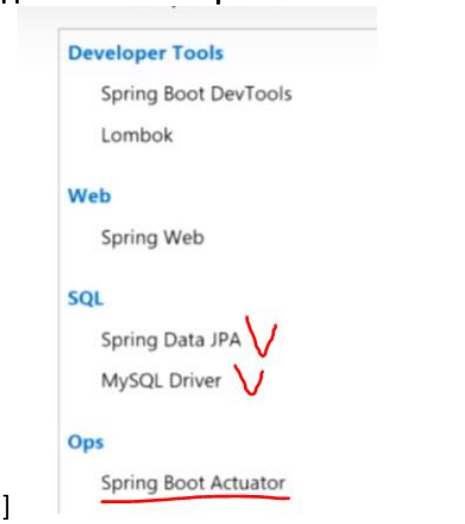
Add name and version:



Add Spring Data JPA and MySQL Driver (in the pom.xml file in reality):



Actuator ни служи за опресняване на данните и следи здравето на нашето приложение, да не се налага да даваме refresh прекалено често



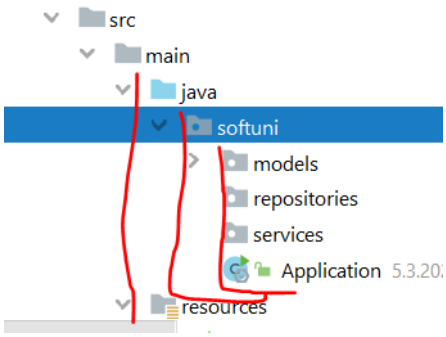
]

applications.properties

In the resources folder, add new **applications.properties** file, which will hold the Spring configuration of the project:

```
#Data Source Properties
spring.datasource.driverClassName = com.mysql.cj.jdbc.Driver
spring.datasource.url =
jdbc:mysql://localhost:3306/bookshop?createDatabaseIfNotExist=true&useSSL=false
spring.datasource.username = root
```

```
spring.datasource.password =

#JPA Properties
spring.jpa.properties.hibernate.dialect = org.hibernate.dialect.MySQL8Dialect
spring.jpa.properties.hibernate.show_sql = TRUE
spring.jpa.properties.hibernate.format_sql = TRUE
spring.jpa.properties.hibernate.use_sql_comments = TRUE


spring.jpa.hibernate.ddl-auto = create-drop
spring.jpa.hibernate.ddl-auto = create       //реално drop и след това create
spring.jpa.hibernate.ddl-auto = update

###Logging Levels
# Disable the default loggers - двете долу ако са активирани, скриват доста от default логовете
logging.level.org = WARN //контролираме логовете
logging.level.blog = WARN //контролираме логовете

#Show SQL executed with parameter bindings
logging.level.org.hibernate.SQL = DEBUG
logging.level.org.hibernate.type.descriptor = TRACE


# Spring actuator settings - service monitoring metrics
#management.endpoints.web.base-path=/
management.endpoints.web.exposure.include=info, health, httptrace, metrics, threaddump, heapdump,
shutdown, beans
management.trace.http.enabled=true
management.endpoint.health.show-details=always
info.app.name=Article repository application
info.app.description=This sample application allows to manage articles and users
info.app.version=1.0.0
```

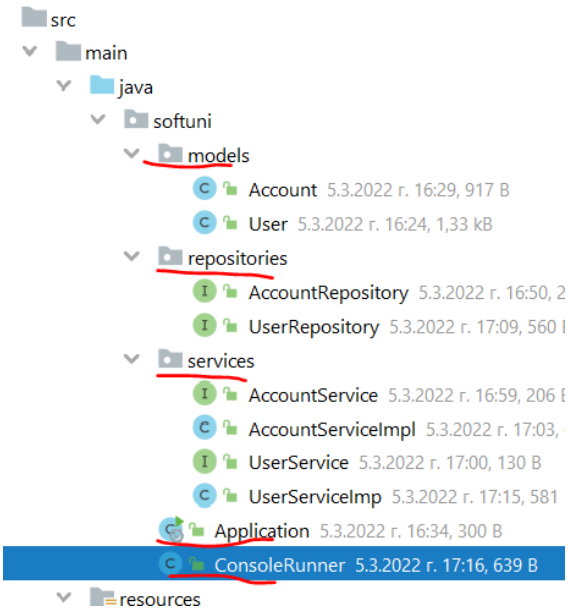


persistence(javax.persistence) за Релациите

```
public class User {
    @Id
    @ Id (javax.persistence)
    @ Id (org.springframework.data.annotation)
```

14. Demo Lab

Една анотация на един Interface ни дава тонове информация наготово



models

```
import javax.persistence.*;
import java.math.BigDecimal;
@Entity(name = "accounts")
public class Account {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;

    @Column(nullable = false)
    private BigDecimal balance;

    @ManyToOne
    private User user;

    public Account() {
    }


@Entity(name = "users")
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;

    @Column(unique = true, nullable = false)
    private String username;

    private int age;

    @OneToMany(targetEntity = Account.class, mappedBy = "user")
    private Set<Account> accounts;

    public User() {
        this.accounts = new HashSet<>();
    }
```

Repositories, customs repositories (Custom CRUD Operations)

```
/* @param <T> the domain type the repository manages
  * @param <ID> the type of the id of the entity the repository manages */

@Repository  //наготово получаваме всички (CRUD) create, read, update, delete операции наготово
public interface UserRepository extends JpaRepository<User, Integer> {
    User findByUsername(String username);
}


import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;
import softuni.models.Account;
@Repository
public interface AccountRepository extends JpaRepository<Account, Integer> {
}
```

The **JpaRepository** interface contains methods like:

- **save(E entity)**
- **findOne(Id primaryKey)**
- **findAll()**
- **count()**
- **delete(E entity)**
- **exists(Id primaryKey)**
- You can define a **custom repository**, which extends the **JpaRepository** and defines several methods for operating with data besides those exposed by the greater interface. The query builder mechanism of Spring Data requires following several rules when you define custom methods. Query creation is done by parsing method names by prefixes like **find…By**, **read…By**, **query…By**, **count…By**, and **get…By**. You can add more criteria by concatenating **And** and **Or** or apply ordering with **OrderBy** with sorting direction **Asc** or **Desc**.

  https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#jpa.query-methods.query-creation

Keyword	Sample	JPQL
And	findByLastnameAndFirstName	… where x.last_name = ?1 and x.firstname = ?2
Or	findByLastnameOrFirstname	… where x.lastname = ?1 or x.firstname = ?2
Between	findByStartDateBetween	… where x.startDate between 1? and ?2
LessThan	findByAgeLessThan	… where x.age < ?1
Containing	findByFirstnameContaining	… where x.firstname like ?1 (par ameter bound wrapped in %)
In	findByAgeIn(Collection<Age> ages)	… where x.age in ?1

services

In bigger applications mixing business logic and crud operations to the database is not wanted. Having a repository objects is implementing the **Domain Driven Design**. Repositories are classes responsible **only for write/transactional operations** towards the data source. Any business logic like validation, calculations and so on is implemented by **a Service Layer**. One of the most important concepts to keep in mind is that a **service** should **never expose details of the internal processes,** or the business entities used within the application.

Implement those services with classes **`AccountServiceImpl`** and **`UserServiceImpl`**. Those classes will do the business logic of the application. To do that, they should have certain type of **`Repository`** available – **`AccountRepository`** or **`UserRepository`** according to the service type.

The implementation of the methods is up to you. Here are some several tips:

- AccountServiceImpl
  - Money withdrawal – should only happen if account **is present** in the database, **belongs to user** and **has enough balance**
  - Money transfer – should only happen if **account belongs to user** and transfer value **is not negative**
- UserServiceImpl
  - User registration – should only happen if user does not exist in the database

```
public interface AccountService {
    void withdrawMoney(BigDecimal amount, Long id);

    void depositMoney(BigDecimal amount, Long id);
}

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import softuni.repositories.AccountRepository;
@Service
public class AccountServiceImpl implements AccountService {
    private final AccountRepository accountRepository;

    @Autowired – през конструктора
    public AccountServiceImpl(AccountRepository accountRepository) {
        this.accountRepository = accountRepository;
    }

    @Override
    public void withdrawMoney(BigDecimal amount, Long id) {   }

    @Override
    public void depositMoney(BigDecimal amount, Long id) {    }
}


public interface UserService {
    void registerUser(User user);
}


import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import softuni.models.User;
```

```java
import softuni.repositories.UserRepository;
@Service
public class UserServiceImp implements UserService {

    @Autowired
    private UserRepository userRepository;

    @Override
    public void registerUser(User user) {
//checking if user exists in the database
        User found = this.userRepository.findByUsername(user.getUsername());

        if (found == null) {
            this.userRepository.save(user);
        }
    }
}
```

15. Откъде четем не absolut path

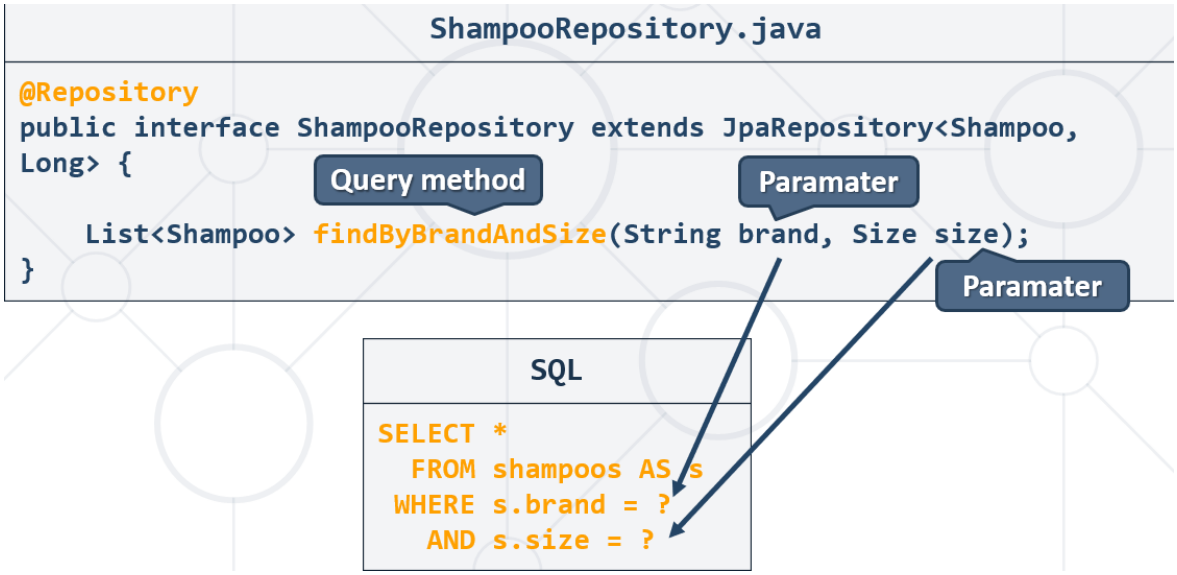
Започвайки от папка src

```java
private static final String RESOURCE_PATH = "src/main/resources/files";
private static final String AUTHORS_FILE_PATH = RESOURCE_PATH + "/authors.txt";
```

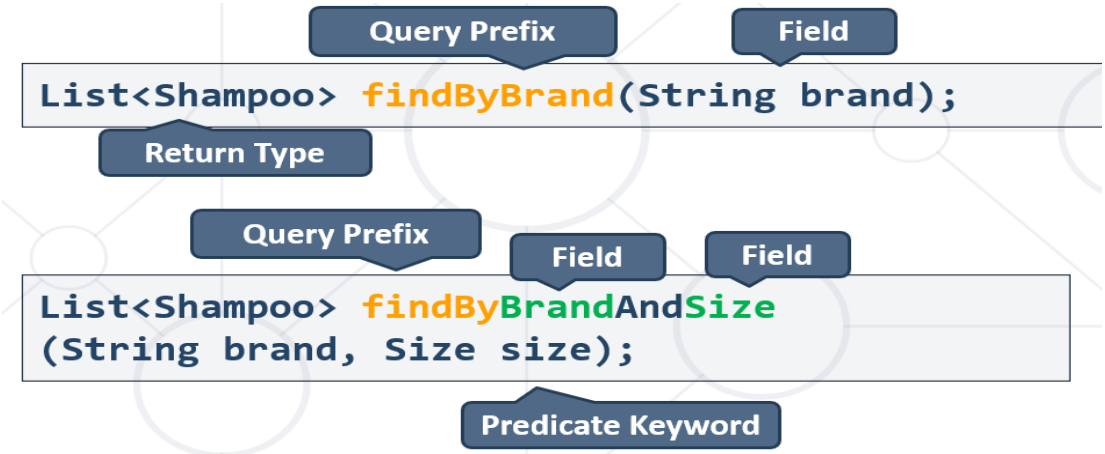
16. Примери за customs repositories (Custom CRUD Operations)

Освен специалните думички като find, get, OrderBy, And, то използваме както имената на полетата, така и типовете на тези полета.

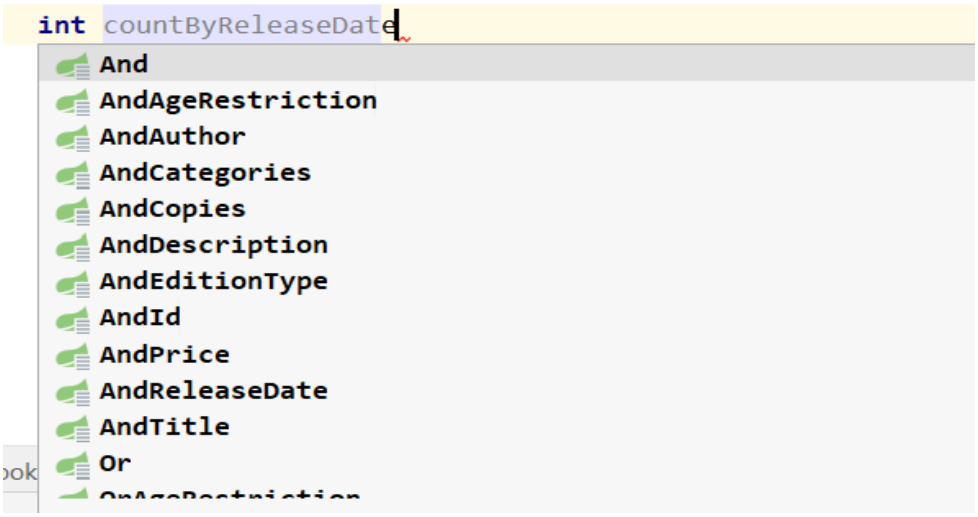
Като аргумент очаква типа на полето, което сме цитирали в името на метода.



0. Подсказки – Automated JPA queries - Query look up



Реално ми излизат подсказки за имената на custom методите!! - яко



Долната черта показва, че влизаме в полето на класа:

```
List<ProductEntity> findAllByCategory_Name(CategoryName categoryName);
```

```
@Entity
@Table(name = "products")
public class ProductEntity {
    @ManyToOne
    private CategoryEntity category;


@Entity
@Table(name = "categories")
public class CategoryEntity {
    @Column(unique = true, nullable = false)
    @Enumerated(EnumType.STRING)
    private CategoryName name;
```

1. countByReleaseDateAfter

Get all books after the year 2000. Print only their titles.

```
@Repository
public interface BookRepository extends JpaRepository<Book, Integer> {
    List<Book> findByReleaseDateAfter(LocalDate releaseDate);

    int countByReleaseDateAfter(LocalDate releaseDate);
}

@Component
public class ConsoleRunner implements CommandLineRunner {
    private final SeedService seedService;
    private final BookRepository bookRepository;

    @Autowired
    public ConsoleRunner(SeedService seedService, BookRepository bookRepository) {
        this.seedService = seedService;
        this.bookRepository = bookRepository;
    }

    private void _01_booksAfter2000() {
        LocalDate yearAfter = LocalDate.of(2000, 1, 1 );
        List<Book> books = bookRepository.findByReleaseDateAfter(yearAfter);

        books.forEach(b -> System.out.println(b.getReleaseDate() + " " + b.getTitle()));

System.out.println("Total count result: " + bookRepository.countByReleaseDateAfter(yearAfter));

    }

    @Override
    public void run(String... args) throws Exception {
        System.out.println("Starting");
//        this.seedService.seedAuthors();
//        this.seedService.seedCategories();
//        this.seedService.seedAll();

        this._01_booksAfter2000();
    }
```

2. findDistinctByBooksReleaseDateBefore

Get all authors with at least one book with release date before 1990. Print their first name and last name.

```
@Component
public class ConsoleRunner implements CommandLineRunner {
    private final SeedService seedService;
    private final BookRepository bookRepository;
    private final AuthorRepository authorRepository;

    @Autowired
    public ConsoleRunner(SeedService seedService, BookRepository bookRepository, AuthorRepository
authorRepository) {
        this.seedService = seedService;
        this.bookRepository = bookRepository;
        this.authorRepository = authorRepository;
    }
```

```
    @Override
    public void run(String... args) throws Exception {
//        this._01_booksAfter2000();
        this._02_allAuthorsWithBookBefore1990();
    }

    private void _02_allAuthorsWithBookBefore1990() {
        LocalDate year1990 = LocalDate.of(1990, 1, 1);
        List<Author> authors =
this.authorRepository.findDistinctByBooksReleaseDateBefore(year1990);

        authors.forEach(a -> System.out.println(a.getFirstName() + " " + a.getLastName()));
    }


@Repository
public interface AuthorRepository extends JpaRepository<Author, Integer> {
    List<Author> findDistinctByBooksReleaseDateBefore(LocalDate releaseDate);
}
```

3. find… – метод на JpaRepository – връща lazy

3.1. Ако искаме да ни върши работа, заявяваме го EAGER

```
@Entity(name = "authors")
public class Author {
@OneToMany(targetEntity = Book.class, mappedBy = "author", fetch = FetchType.EAGER)
private Set<Book> books;


import demo.entities.Author;
import demo.entities.Book;
import demo.repositories.AuthorRepository;
import demo.repositories.BookRepository;
import demo.services.SeedService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.CommandLineRunner;
import org.springframework.stereotype.Component;

import java.time.LocalDate;
import java.util.List;

@Component
public class ConsoleRunner implements CommandLineRunner {
    private final SeedService seedService;
    private final BookRepository bookRepository;
    private final AuthorRepository authorRepository;

    @Autowired
    public ConsoleRunner(SeedService seedService, BookRepository bookRepository, AuthorRepository
authorRepository) {
        this.seedService = seedService;
        this.bookRepository = bookRepository;
        this.authorRepository = authorRepository;
    }

    @Override
    public void run(String... args) throws Exception {
        System.out.println("Starting");
        this._03_allAuthorsOrderedByBookCount();
    }
```

```
    private void _03_allAuthorsOrderedByBookCount() {
        List<Author> authors = this.authorRepository.findAll();  //тук не взема всички автори ако
е lazy
        authors.stream()
                .sorted((f, s) -> Integer.compare(s.getBooks().size(), f.getBooks().size()))
                .forEach(a -> System.out.printf("%s %s -> %d%n", a.getFirstName(),
                        a.getLastName(), a.getBooks().size()));
    }
}
```

3.2. Втори вариант е да използваме Transactional анотация

Тогаваме няма нужда да използваме EAGER

```
@Entity(name = "authors")
public class Author {
@OneToMany(targetEntity = Book.class, mappedBy = "author")
private Set<Book> books;
```

```
import demo.entities.Author;
import demo.entities.Book;
import demo.repositories.AuthorRepository;
import demo.repositories.BookRepository;
import demo.services.SeedService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.CommandLineRunner;
import org.springframework.stereotype.Component;

import java.time.LocalDate;
import java.util.List;

@Component
public class ConsoleRunner implements CommandLineRunner {
    private final SeedService seedService;
    private final BookRepository bookRepository;
    private final AuthorRepository authorRepository;

    @Autowired
    public ConsoleRunner(SeedService seedService, BookRepository bookRepository, AuthorRepository
authorRepository) {
        this.seedService = seedService;
        this.bookRepository = bookRepository;
        this.authorRepository = authorRepository;
    }

//и двата transactional бачкат
    @Override
        @Transactional - import javax.transaction.Transactional;
    @Transactional   - import org.springframework.transaction.annotation.Transactional;
    public void run(String... args) throws Exception {
        System.out.println("Starting");
        this._03_allAuthorsOrderedByBookCount();
    }

    private void _03_allAuthorsOrderedByBookCount() {
        List<Author> authors = this.authorRepository.findAll();  //тук не взема всички автори ако
е lazy, но с иползването на Transactional в run метода минава
        authors.stream()
                .sorted((f, s) -> Integer.compare(s.getBooks().size(), f.getBooks().size()))
```

```
                .forEach(a -> System.out.printf("%s %s -> %d%n", a.getFirstName(),
                        a.getLastName(), a.getBooks().size()));
    }
}
```

3.3. findByFirstNameAndLastName

```
private void _04_GetAllBooksByAuthorname() {
    String[] nameOfAuthor = "George Powell".split("\\s+");
    Author author = this.authorRepository.findByFirstNameAndLastName(nameOfAuthor[0],
nameOfAuthor[1]);
    List<Book> allBooksByAuthor = this.bookRepository.findAllByAuthorId(author.getId());
    allBooksByAuthor.stream()
            .sorted((f, s) -> {
                int i = s.getReleaseDate().compareTo(f.getReleaseDate());
                if (i == 0) {
                    i = f.getTitle().compareTo(s.getTitle());
                }

                return i;
            })
            .forEach(book -> System.out.printf("Book: %s, Release Date: %s%n", book.getTitle(),
book.getReleaseDate()));
}
```

4. OrderBy... - метод на JpaRepository

```
@Override
public List<String> getAllBooksFromAuthorOrderedByReleaseDateDescAndBookTitleAscending
        (String firstName, String lastName) {

    List<String> bookInfo = new ArrayList<>();
    bookRepository
            .findAllByAuthorOrderByReleaseDateDescTitle(authorService.getAuthor(firstName,
lastName))
            .forEach(book -> {
                bookInfo.add(String.format("%s %s %d"
                        , book.getTitle()
                        , book.getReleaseDate()
                        , book.getCopies()
                ));
            });

    return bookInfo;
}
```

5. LessThanOrGreaterThan

```
List<Book> findByPriceLessThanOrPriceGreaterThan(BigDecimal lowerBound, BigDecimal upperBound);
```

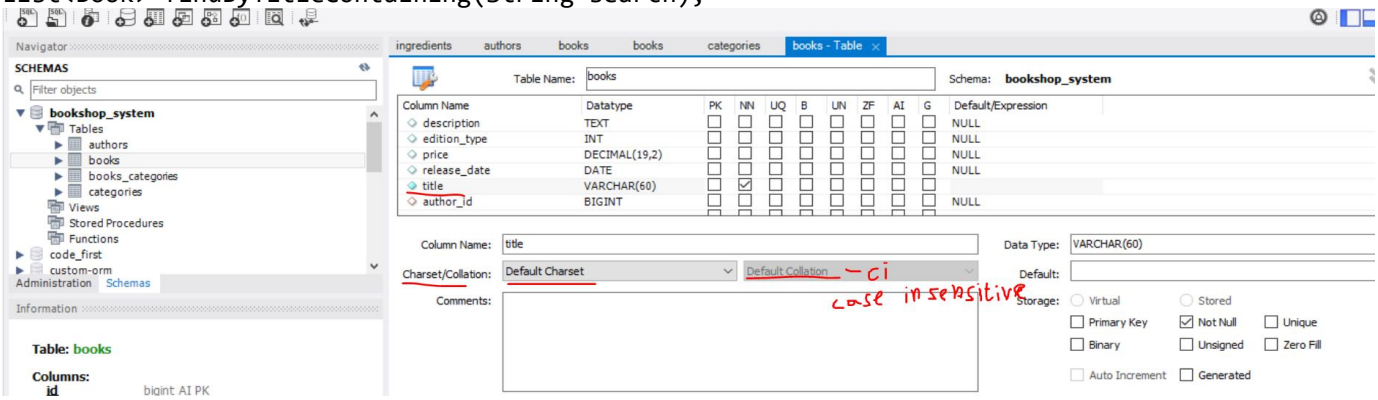
6. Between

7. EndingWith/StartingWith

```
List<Author> findByFirstNameEndingWith(String endsWith);

List<Book> findByAuthorLastNameStartingWith(String search);
```

8. Containing – работи case-insensitive

```
List<Book> findByTitleContaining(String search);
```



17. Transactions

Има изпълнение в две насоки:

- На база едно и също proxy
- На база една и съща инстанция

Ако използваме @Transactional от друг клас(класа CommandLineRunner и метода run), то няма да се бърка.

Много е вероятно ако викаме на един service клас на различните му методи една и съща анотация @Transactional, то транзакциите да не се изпълнят правилно.

AccountServiceImpl.java

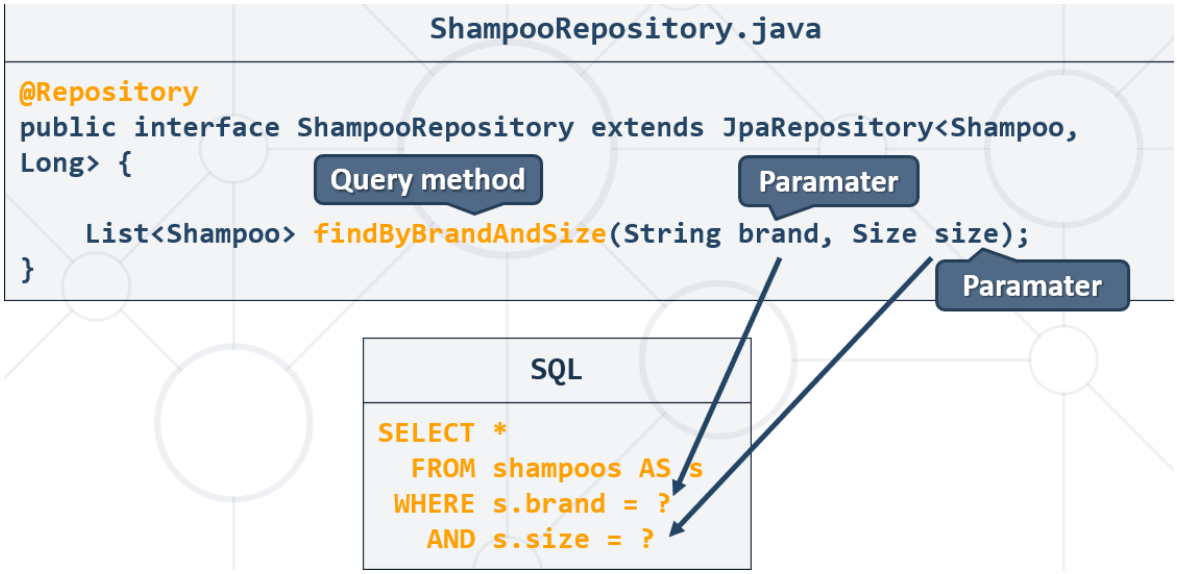
```
@Transactional
@Override
public void transferMoney(long fromId, long toId, BigDecimal amount) {
    depositMoney(toId, amount);
    withdrawMoney(fromId, amount);
}
```

8. Spring Data Advanced Querying

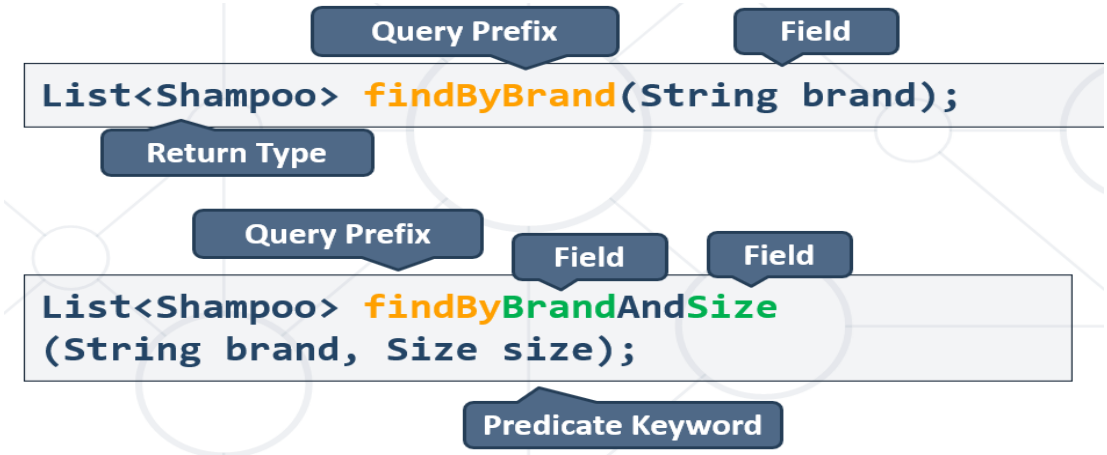
Стратегия – пишем от по-общото (от run() метода) към по частното, за да може IntelliJ да ни създаде лесно в съответните методи на различните layers методите!

1. Custom DSL Automated Queries

Domain Specific language (DSL) – такъв език за специфична система, който като напишем нещо с findBy, OrderBy и т.н. системата го разбира.
Освен специалните думички като find, get, OrderBy, And, то използваме както имената на полетата, така и типовете на тези полета.
Като аргумент очаква типа на полето, което сме цитирали в името на метода.



Query look up



2. Custom Manual JPQL query = JPA query

@Query анотацията се изпълнява без значение дали името съвпада с custom automated query на Spring

Пример 1:

```
@Component
public class Runner implements CommandLineRunner {
    private final ShampooRepository shampooRepository;

    @Autowired
    public Runner(ShampooRepository shampooRepository) {
        this.shampooRepository = shampooRepository;
    }

    @Override
    public void run(String... args) throws Exception {
        _0007_SelectShampoosbyIngredients();
    }

      private void _0007_SelectShampoosbyIngredients() {
          Scanner sc = new Scanner(System.in);
```

```
            String first = sc.nextLine();
            String second = sc.nextLine();

            Set<String> names = Set.of(first, second);
            Set<String> collect = this.shampooService._07_selectByIngredientsNames(names)
                .stream()
                .map(s -> s.getBrand())
                .collect(Collectors.toSet());

        collect.forEach(System.out::println);

        }
}
```

```
@Repository
public interface ShampooRepository extends JpaRepository<Shampoo, Long> {

        @Query("SELECT s FROM Shampoo s" +
                " JOIN s.ingredients AS i"+
                " WHERE i.name IN :ingredientNames")  //ако някоя от съставките за даден шампоан е
        в списъка с търсени съставки
        List<Shampoo> findByIngredientsNames(
                @Param(value = "ingredientNames") Set<String> ingredientNames); //може и без value


        @Query("SELECT s FROM Shampoo s" +
                " WHERE s.ingredients.size < :count") //deprecated syntax for size
           или     " WHERE size(s.ingredients) < :count")
        List<Shampoo> findByIngredientCountLessThan(@Param(value = "count")int count); //може и без
        value
}
```

Пример 2:

```
@Repository
public interface IngredientRepository extends JpaRepository<Ingredient, Long>{

    @Query(value = "select s from Shampoo s " +
            "join s.ingredients i where i in :ingredients")
    List<Shampoo> findByIngredientsIn(@Param(value = "ingredients")
                                              Set<Ingredient> ingredients);
}
```

3. Java Persistence Query Language (JPQL = a JPA query)

- **Object-oriented** query language – не оперираме с таблицата от базата данни, а с класа/обекта
  - Part of the Java Persistence API
  - Used to make queries against entities stored in a relational database
  - SQL syntax **operating with entities**, not tables in the data source

4. JPQL Custom queries - functionalities

Insert

Insert е persist реално

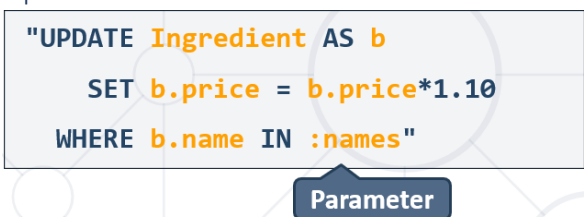
Select

Оказва се, че в Spring Data JPQL лесно работи освен с цял обект на клас, то и с определени полета от класа! В Hibernate JPA беше по-трудно!!!

В следната заявка, JPQL връща java.lang.String, и ние сме го писали List<String> което си съответства.

```
@Repository
public interface BookRepository extends JpaRepository<Book, Long> {


    @Query("SELECT b.title FROM Book b WHERE b.ageRestriction = :ageRestriction")
    List<String> findTitleByAgeRestriction(@Param(value = "ageRestriction") AgeRestriction
restriction); //може и без value
```

Update




```
@Component
public class Runner implements CommandLineRunner {
    private final ShampooService shampooService;
    private final IngredientService ingredientService;

    @Autowired
    public Runner(ShampooService shampooService, IngredientService ingredientService) {
        this.shampooService = shampooService;
        this.ingredientService = ingredientService;
    }

    @Override
    public void run(String... args) throws Exception {
        _10_UpdateIngredientsbyPrice();
    }

    private void _10_UpdateIngredientsbyPrice() {
        this.ingredientService.increasePriceByPercentage(0.1);
    }
}
```

```java
public interface IngredientService {
    void increasePriceByPercentage(double v);
}
```

```java
@Service
public class IngredientServiceImpl implements IngredientService {
    @Autowired
    private IngredientRepository ingredientRepository;

    public IngredientServiceImpl(IngredientRepository ingredientRepository) {
        this.ingredientRepository = ingredientRepository;
    }

    @Override
    @Transactional //Ако има повече операции в сервиса, го слагаме тук
    public void increasePriceByPercentage(double percent) {
        BigDecimal actualPercent = BigDecimal.valueOf(1 + percent); //преобразуване на данни в сервиса
        this.ingredientRepository.increasePriceByPercent(actualPercent);
    }
}
```

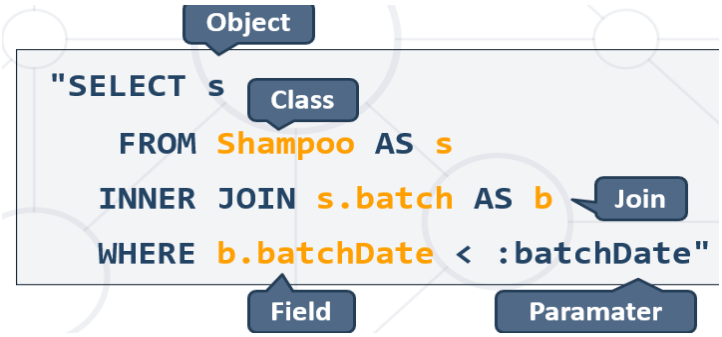
```java
@Repository
public interface IngredientRepository extends JpaRepository<Ingredient, Long> {

    @Modifying
        @Transactional  //Ако е единична операция, го слагаме тук. Т.е. се стараем да слагаме transactional максимално навътре/надолу в долните layers
    @Query("UPDATE Ingredient i SET i.price = i.price * :multiplier") //тук си го умножава BigDecimal без проблем. Типовете се валидират в Java, но @Query се изпълнява на ниво база данни.
    void increasePriceByPercent(@Param(value = "multiplier") BigDecimal percent);


      @Modifying //върви заедно само с @Query
      @Query("UPDATE Ingredient i SET i.price = i.price * :multiplier WHERE i.name IN :listIngr")
      void increasePriceByPercentByAGivenList(@Param(value = "multiplier") BigDecimal percent,
                                              @Param(value = "listIngr") Set<String> ingredients);
      //може и без value
      }
```

Delete

```
"DELETE FROM Ingredient AS b
  WHERE b.name = :name"
```

```java
@Repository
public interface BookRepository extends JpaRepository<Book, Long> {

    @Transactional   //в случая тук няма нужда от @Modifying, защото нямаме @Query
    int deleteByCopiesLessThan(int amountCopies);
}
```

Join




```
@Repository
public interface ShampooRepository extends JpaRepository<Shampoo, Long> {

	@Query("SELECT s FROM Shampoo s" +
			" JOIN s.ingredients AS i"+
			" WHERE i.name IN :ingredientNames")  //ако някоя от съставките за даден шампоан е
	в списъка с търсени съставки
	List<Shampoo> findByIngredientsNames(
			@Param(value = "ingredientNames") Set<String> ingredientNames); //може и без value



	@Query("SELECT s FROM Shampoo s" +
			" WHERE s.ingredients.size < :count") //deprecated syntax for size
	   или     " WHERE size(s.ingredients) < :count")
	List<Shampoo> findByIngredientCountLessThan(@Param(value = "count")int count);  //може и
	без value
}
```

:параметри и ?1 ?2 позиционни параметри

Това не работи

```
@Query("UPDATE Ingredient i SET i.price = i.price * ?1 WHERE i.name IN ?2")
void increasePriceByPercentByAGivenList(@Param(value = "multiplier") BigDecimal percent,
                                        @Param(value = "listIngr") Set<String> ingredients);
```

Това работи

```
TypedQuery<Customer> typedQuery = em.createQuery("SELECT c FROM Customer c WHERE c.firstName =
?1", Customer.class);
typedQuery.setParameter(1,"Mike");

List<Customer> customers=typedQuery.getResultList();
```

5. Repository Inheritance

Repository Inheritance

- In bigger applications, we have **similar entities**, extending an **abstract class**
- Their base attributes and actions, towards them, are the same regardless of their differences
- We can set up a **base repository** to reduce query and code duplication
- It can be **inherited** to clear up specifics - можем последващ клас да имплементира както JpaRepository, CRUDRepository, така и нашия CustomRepository

Example:

Not a repository

IngredientRepository.java

```
@NoRepositoryBean
public interface IngredientRepository<T extends Ingredient>
extends JpaRepository<T, Long>{
        //…
}
```

ChemicalIngredientRepository.java

```
@Repository
public interface ChemicalIngredientRepository extends IngredientRepository
<BasicChemicalIngredient> {
   List<ChemicalIngredient> findByChemicalFormula(String chemicalFormula);
}
```

CustomShampooRepository.java

```
public interface CustomShampooRepository {

    void create(BasicShampoo basicShampoo);
}
```

CustomShampooRepositoryImpl.java

```
@Repository
public class CustomShampooDaoImpl implements CustomShampooRepository {
    @PersistenceContext
    private EntityManager entityManager;

    @Transactional
    public void create(BasicShampoo basicShampoo){
        entityManager.persist(basicShampoo);
    }
}
```

Inject Entity Manager

Single Transaction

6. Spring Custom Configuration

Application Properties configuration

- So far, we've configured our project with a spring properties file:

application.properties

```
#Data Source Properties
spring.datasource.driverClassName=com.mysql.cj.jdbc.Driver
spring.datasource.url=jdbc:mysql://localhost:3306/shampoo_company?useSSL=false&createDatabaseIfNo
tExist=true
spring.datasource.username=root
spring.datasource.password=

#JPA Properties
spring.jpa.properties.hibernate.dialect = org.hibernate.dialect.MySQL8Dialect
spring.jpa.properties.hibernate.format_sql = TRUE
spring.jpa.hibernate.ddl-auto = update
spring.jpa.open-in-view=false
```

```
###Logging Levels
# Disable the default loggers – двете долу ако са активирани, скриват доста от default логовете
logging.level.org = WARN
logging.level.blog = WARN

#Show SQL executed with parameter bindings
logging.level.org.hibernate.SQL = DEBUG
logging.level.org.hibernate.type.descriptor = TRACE


#Change server port
#server.port=8000
```

Java-Based Configuration

При започнал вече проект обикновено се използва

JavaConfig.java

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.PropertySource;
import org.springframework.core.env.Environment;
import org.springframework.data.jpa.repository.config.EnableJpaRepositories;
import org.springframework.jdbc.datasource.DriverManagerDataSource;
import org.springframework.orm.jpa.JpaTransactionManager;
import org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean;
import org.springframework.orm.jpa.vendor.Database;
import org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter;
import org.springframework.transaction.PlatformTransactionManager;
import org.springframework.transaction.annotation.EnableTransactionManagement;

import javax.persistence.EntityManagerFactory;
import javax.sql.DataSource;
import java.util.Properties;

@Configuration  //Configuration class
@EnableJpaRepositories(basePackages = "advquerying.repositories")  //repositories directory
@EnableTransactionManagement
@PropertySource(value = "application.properties")
public class JavaConfig {
    //Add configuration
    @Autowired   //Look up прилагане на Inversion of control principle
    private Environment environment;

    //Data Source Connection
    @Bean
    public DataSource dataSource() {
        DriverManagerDataSource driverManagerDataSource = new DriverManagerDataSource();


driverManagerDataSource.setDriverClassName(environment.getProperty("spring.datasource.driverClass
Name"));
        driverManagerDataSource.setUrl(environment.getProperty("spring.datasource.url"));

driverManagerDataSource.setUsername(environment.getProperty("spring.datasource.username"));

driverManagerDataSource.setPassword(environment.getProperty("spring.datasource.password"));
```

```
        return driverManagerDataSource;
    }

    //JPA Configuration
    @Bean
    public EntityManagerFactory entityManagerFactory() {
        HibernateJpaVendorAdapter vendorAdapter = new HibernateJpaVendorAdapter();
        vendorAdapter.setDatabase(Database.MYSQL);
        vendorAdapter.setGenerateDdl(true);
        vendorAdapter.setShowSql(true);

        LocalContainerEntityManagerFactoryBean factory = new
LocalContainerEntityManagerFactoryBean();
        factory.setJpaVendorAdapter(vendorAdapter);
        factory.setPackagesToScan("advquerying");
        factory.setDataSource(dataSource());

        Properties jpaProperties = new Properties();
        jpaProperties.setProperty("hibernate.hbm2ddl.auto", "validate");
        jpaProperties.setProperty("hibernate.format_sql", "true");

        factory.setJpaProperties(jpaProperties);
        factory.afterPropertiesSet();

        return factory.getObject();
    }

    //Transaction Manager Configuration
    @Bean
    public PlatformTransactionManager transactionManager() {
        JpaTransactionManager txManager = new JpaTransactionManager();
        txManager.setEntityManagerFactory(entityManagerFactory());
        return txManager;
    }
}
```

7. Правилен подход при custom JPQL (= JPA) queries

Важи както за SpringData JPQL, така и за Hybernate JPA JPQL, така и за Hybernate HQL, така и за JDBC API SQL.

Вариант 3 и вариант 4 са правилните!!!

Вариант 1 – CONCAT и java.lang.String class

```
    @Query("SELECT CONCAT(a.firstName, ' ', a.lastName, ' ', SUM(b.copies)), SUM(b.copies) AS
totalCopies" +
        " FROM Author a" +
        " JOIN a.books AS b" +
        " GROUP BY b.author" +
        " ORDER BY totalCopies DESC")
List<String> getWithTotalCopies();
```

Вариант 2 – Object[]

Виж търси по Object[] в този, за да го намериш лесно

Вариант 3 – Closed Projection с използване на interface – не сработва с GSON

```
public interface AuthorNamesWithTotalCopies {
    String getFirstName();
    String getLastName();
    long getTotalCopies();
}
```

```
@Repository
public interface AuthorRepository extends JpaRepository<Author, Long> {

    @Query("SELECT a FROM Author a ORDER BY a.books.size DESC") //Deprecated, новия JPQL синтаксис е
SIZE(a.books), но става и чрез COUNT(a.books)
        @Query("SELECT a FROM Author a ORDER BY size(a.books) DESC")
    List<Author> findAllByBooksSizeDESC();

    List<Author> findByFirstNameEndingWith(String endsWith);

//Използваме Alias тук спрямо интерфейс имената на getters
        @Query("SELECT a.firstName AS firstName, a.lastName AS lastName, SUM(b.copies) AS
totalCopies" +
            " FROM Author a" +
            " JOIN a.books AS b" +
            " GROUP BY b.author" +
            " ORDER BY totalCopies DESC")
    List<AuthorNamesWithTotalCopies> getWithTotalCopies();
}

public interface AuthorService {
    List<AuthorNamesWithTotalCopies> _10_getWithTotalCopies();
}

@Service
public class AuthorServiceImpl implements AuthorService {
    private final AuthorRepository authorRepository;

    @Autowired
    public AuthorServiceImpl(AuthorRepository authorRepository) {
        this.authorRepository = authorRepository;
    }

    @Override
    public List<AuthorNamesWithTotalCopies> _10_getWithTotalCopies() {
        return this.authorRepository.getWithTotalCopies();
    }
}
```

```
@Component
public class CommandLineRunnerImpl implements CommandLineRunner {

    private final CategoryService categoryService;
    private final AuthorService authorService;
    private final BookService bookService;

    @Autowired
    public CommandLineRunnerImpl(CategoryService categoryService, AuthorService authorService,
BookService bookService) {
```

```
        this.categoryService = categoryService;
        this.authorService = authorService;
        this.bookService = bookService;
    }

    @Override
    public void run(String... args) throws Exception {
        this.authorService._10_getWithTotalCopies()
                .forEach(a -> System.out.println(a.getFirstName() + " "
                        + a.getLastName() + " - "
                        + a.getTotalCopies()));
    }
}
```

Вариант 4 – Open Projection – отново с използване на interface - не сработва с GSON

```
public interface ProductWithoutBuyerDTO {
    String getName();

    BigDecimal getPrice();

    //Open projection
    @Value("#{target.seller.firstName + ' ' + target.seller.lastName}")
    String getSeller();
}
```

```
@Override
    public void run(String... args) throws Exception {
//        this.seedService.seedAll();
        List<ProductWithoutBuyerDTO> productsForSell =
this.productService.getProductsInPriceRangeForSell(500.0f, 1000.0f);
        productsForSell
        .forEach(p -> System.out.println(String.format("Product name: %s --- Product price: %.2f
--- Seller full name: %s",
                p.getName(), p.getPrice(), p.getSeller())));
Gson не го харесва при този сценарий с Open Projection !!!

    }
```

```
@Override
public List<ProductWithoutBuyerDTO> getProductsInPriceRangeForSell(float from, float to) {
    BigDecimal rangeStart = BigDecimal.valueOf(from);
    BigDecimal rangeEnd = BigDecimal.valueOf(to);

    return this.productRepository.findAllByPriceBetweenAndBuyerIsNullOrderByPriceAsc(rangeStart,
rangeEnd);
}
```

```
@Repository
public interface ProductRepository extends JpaRepository<Product, Integer> {
    List<ProductWithoutBuyerDTO> findAllByPriceBetweenAndBuyerIsNullOrderByPriceAsc
            (BigDecimal rangeStart, BigDecimal rangeEnd);
}
```

Вариант 5 – с използване на class

Class based projections do not work with native queries. As a workaround you may use named queries with `ResultSetMapping` or the Hibernate specific `ResultTransformer`

Извикваме new конструктор с целия път все едно – работело така, ама не е ок като цяло

```
@Query("SELECT new com.example.springintro.model.entity.BookSummaryImpl(b.title, b.editionType," +
        " b.ageRestriction, b.price)" +
        " FROM Book b" +
        " WHERE b.copies > :copies")
List<BookSummaryImpl> findAllByCopiesGreaterThan(int copies);
```

Пример:

```
public class ProductWithoutBuyerDTO {
    private String name;
    private BigDecimal price;
    private String seller;

    public ProductWithoutBuyerDTO(String name, BigDecimal price, String firstName, String
lastName) {
        this.name = name;
        this.price = price;

        if (firstName == null) {
            this.seller = lastName;
        } else {
            this.seller = firstName + " " + lastName;
        }
    }

    public String getName() {
        return name;
    }

    public BigDecimal getPrice() {
        return price;
    }

    public String getSeller() {
        return seller;
    }
}
```

```
@Repository
public interface ProductRepository extends JpaRepository<Product, Integer> {

        //цитираме целият път на класа, понеже @Query заявката не го намира иначе
    @Query("SELECT new
com.example.json_ex.productsshop.entities.products.ProductWithoutBuyerDTO(" +
            " p.name, p.price, p.seller.firstName, p.seller.lastName)" +
            " FROM Product AS p" +
            " WHERE p.price >= :rangeStart AND p.price <= :rangeEnd AND p.buyer IS NULL" +
            " ORDER BY p.price ASC")
    List<ProductWithoutBuyerDTO> findAllByPriceBetweenAndBuyerIsNullOrderByPriceAsc
```

```
            (BigDecimal rangeStart, BigDecimal rangeEnd);
}


@Entity
@Table(name = "products") //за да върви горната заявка, то иска @Table
public class Product {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;

    @Column(nullable = false)
    private String name;

    @Column(nullable = false)
    private BigDecimal price;

    @ManyToOne
//    @JoinColumn(name = "seller_id")
    private User seller;

    @ManyToOne
//    @JoinColumn(name = "buyer_id")
    private User buyer;

    @ManyToMany
    private Set<Category> categories;

    public Product() {
        this.categories = new HashSet<>();
    }

    public Product(String name, BigDecimal price) {
        this();

        this.name = name;
        this.price = price;
    }
```

Вариант 6 – следващата тема за DTO(Data Transfer Object) в комбинация с ModelMapper и Converter

- We can **map entity objects to DTOs** using ModelMapper – вземаме обекта/листа от обекти от базата от repository-то , и го мапваме към специфично DTO

```
Да използваме method reference пише в документацията за вътрешната команда.map
@Service
public class ProductServiceImpl implements ProductService {
    private final ProductRepository productRepository;
    private final ModelMapper mapper;
    private final TypeMap<Product, ProductWithAttributesDTO> productToDtoMapping;

    @Autowired
    public ProductServiceImpl(ProductRepository productRepository) {
        this.productRepository = productRepository;
        this.mapper = new ModelMapper();

        //seller -> firstName + lastName
        //от User на Стринг
```

```
        Converter<User, String> userToFullNameConverter = context ->
                context.getSource() == null ? null : context.getSource().getFullName();
//getSource() реално извиква инстанцията на User класа

        TypeMap<Product, ProductWithAttributesDTO> typeMap =
this.mapper.createTypeMap(Product.class, ProductWithAttributesDTO.class);

        this.productToDtoMapping = typeMap.addMappings(m -> m
                .using(userToFullNameConverter)
                .map(Product::getSeller, ProductWithAttributesDTO::setSeller));


или

        // <V> void map(SourceGetter<S> sourceGetter, DestinationSetter<D, V> destinationSetter);
        this.productToDtoMapping = typeMap.addMappings(m -> m
//                .when(Objects::nonNull)
                .using(userToFullNameConverter)
//                .map(Product::getSeller, ProductWithAttributesDTO::setSeller));
                .map(Product::getSeller, ((destination, value) -> destination.setSeller((String)
value))));


        this.mapper.addConverter(userToFullNameConverter);
    }

    @Override
    public ExportProductsInRangeDTO getInRange(float from, float to) {
        BigDecimal rangeFrom = BigDecimal.valueOf(from);
        BigDecimal rangeTo = BigDecimal.valueOf(to);

        List<Product> products =
this.productRepository.findAllByPriceBetweenAndBuyerIsNullOrderByPriceDesc(rangeFrom, rangeTo);
        List<ProductWithAttributesDTO> dtos = products
                .stream()
                .map(p -> this.productToDtoMapping.map(p))
                .collect(Collectors.toList());

        return new ExportProductsInRangeDTO(dtos);
    }
}



@Entity
@Table(name = "users")
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;

    @Column(name = "first_name", length = 255)
    private String firstName;

    @Column(name = "last_name", nullable = false, length = 255)
    private String lastName;

    private Integer age;

    @OneToMany(targetEntity = Product.class, mappedBy = "seller")
    private List<Product> sellingItems;
```

```
    @OneToMany(targetEntity = Product.class, mappedBy = "buyer")
    private List<Product> itemsBought;

    @ManyToMany
    private Set<User> friends;


    public User() {
        this.sellingItems = new ArrayList<>();
        this.itemsBought = new ArrayList<>();
        this.friends = new HashSet<>();
    }

    public User(String firstName, String lastName, Integer age) {
        this();

        this.firstName = firstName;
        this.lastName = lastName;
        this.age = age;
    }

    public String getFullName() {
        if (this.firstName == null) {
            return this.lastName;
        } else {
            return this.firstName + " " + this.lastName;
        }
    }
}
```

Вариант 7 – следващата тема за DTO(Data Transfer Object) и чрез извикване на конструктора на DTO-то

```
public class EmployeeSpringDto {
    private String name;
    private String lastName;

    public EmployeeSpringDto(Employee employee) {
        this.name = employee.getFirstName() + “ ” + employee.getLastName();
    }
```

Пълен пример е следния:

```
@Controller
public class ExportController {
    private final EmployeeService employeeService;
    private final Gson gson;

    @Autowired
    public ExportController(ProjectService projectService, EmployeeService employeeService, Gson
gson) {
        this.projectService = projectService;
        this.employeeService = employeeService;
        this.gson = gson;
    }

    @GetMapping("/export/employees-above")
    public ModelAndView showEmployeesAbove25(){
        ModelAndView modelAndView = new ModelAndView("export/export-employees-with-age");
```

```
        List<ExportEmployeeDTO> employeesAbove25 = this.employeeService.getEmployeesAbove25();

        StringBuilder sb = new StringBuilder();
        this.gson.toJson(employeesAbove25, sb);

        modelAndView.addObject("employeesAbove", sb.toString());

        return modelAndView;
    }
}


public class ExportEmployeeDTO {
    private String fullName;
    private int age;
    private String projectName;

    public ExportEmployeeDTO(Employee employee){
        this.fullName = employee.getFirstName() + " " + employee.getLastName();
        this.age = employee.getAge();
        this.projectName = employee.getProject().getName();
    }
}


@Service
public class EmployeeService {
    private final Path xmlPath = Path.of("src/main/resources/files/xmls/employees.xml");
    private final EmployeeRepository employeeRepository;

    @Autowired
    public EmployeeService(EmployeeRepository employeeRepository) {
        this.employeeRepository = employeeRepository;
    }

    public List<ExportEmployeeDTO> getEmployeesAbove25() {
        List<Employee> employees =
this.employeeRepository.findByAgeGreaterThanOrderByProjectNameAsc(25);

        return employees.stream()
                .map(e -> new ExportEmployeeDTO(e))
                .collect(Collectors.toList());
    }
}


@Repository
public interface EmployeeRepository extends JpaRepository<Employee, Long> {
    List<Employee> findByAgeGreaterThanOrderByProjectNameAsc(int i);
}
```

8. Stored procedures in Spring data JPA repositories

Some method to our repository that call stored procedure

1. Map a Stored Procedure Name Directly

- We can define a stored procedure method using the @Procedure annotation, and map the stored procedure name directly.

```
@Procedure
int GET_TOTAL_CARS_BY_MODEL(String model);
```

- If we want to define a different method name, we can put the stored procedure name as the element of the @Procedure annotation:

```
@Procedure("GET_TOTAL_CARS_BY_MODEL")
int getTotalCarsByModel(String model);
```

- We can also use the procedureName attribute to map the stored procedure name:

```
@Procedure(procedureName = "GET_TOTAL_CARS_BY_MODEL")
int getTotalCarsByModelProcedureName(String model);
```

- Finally, we can use the value attribute to map the stored procedure name:

```
@Procedure(value = "GET_TOTAL_CARS_BY_MODEL")
int getTotalCarsByModelValue(String model);
```

2. Reference a Stored Procedure Defined in Entity

We can also use the @NamedStoredProcedureQuery annotation to define a stored procedure in the entity class:

```
@Entity
@NamedStoredProcedureQuery(name = "Car.getTotalCardsbyModelEntity",
     procedureName = "GET_TOTAL_CARS_BY_MODEL", parameters = {
           @StoredProcedureParameter(mode = ParameterMode.IN, name = "model_in", type =
String.class),
           @StoredProcedureParameter(mode = ParameterMode.OUT, name = "count_out", type =
Integer.class)})
public class Car { // class definition }
```

Then we can reference this definition in the repository:

```
@Procedure(name = "Car.getTotalCardsbyModelEntity")
int getTotalCarsByModelEntiy(@Param("model_in") String model);

@Procedure(name="udp_get_books_count_by_author")
int getBooksCountByAuthor(@Param("first_name") String firstName, @Param("last_name") String
lastName);
```

We use the name attribute to reference the stored procedure defined in the entity class. For the repository method, we use @Param to match the input parameter of the stored procedure. We also match the output parameter of the stored procedure to the return value of the repository method.

3. Reference a Stored Procedure With the @Query Annotation

In SQL database, we create the stored procedure

```
DELIMITER $$
CREATE PROCEDURE totalAmountOfBooksByAuthor(authorNames VARCHAR(45))
BEGIN
        SELECT count(*) FROM books AS b
  JOIN authors AS a
```

```
   ON a.id = b.author_id
   WHERE CONCAT(a.first_name, ' ', a.last_name) = authorNames;
END $$

CALL totalAmountOfBooksByAuthor('Amanda Rice');
```

We can also call a stored procedure directly with the @Query annotation:

In this method, we use a native query to call the stored procedure. We store the query in the value attribute of the annotation.
Similarly, we use @Param to match the input parameter of the stored procedure. We also map the stored procedure output to the int.

```
@Query(value = "CALL totalAmountOfBooksByAuthor(:authorNames);", nativeQuery = true)
int findCountBooksAnAuthorWrote(String authorNames);
```

9. Special parameter handling – Pagable, Slice and Sort

https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#repositories.special-parameters

Using `Pageable`, `Slice`, and `Sort` in query methods

```
Page<User> findByLastname(String lastname, Pageable pageable);

Slice<User> findByLastname(String lastname, Pageable pageable);

List<User> findByLastname(String lastname, Sort sort);

List<User> findByLastname(String lastname, Pageable pageable);
```

Разписване за Pageable

```
@Override
public void _12PrintInPages5() {
    Page<Shampoo> page = shampooRepository.findAll(PageRequest.of(0, 5));

    System.out.printf("Page %d of %d:%n-------------%n",
            page.getNumber()+1, page.getTotalPages()   //брои ги от 0, затова добавям + 1
    );

    page.forEach(s -> System.out.printf("%s %s %s %.2f %s%n",
            s.getBrand(), s.getSize(), s.getLabel().getTitle(), s.getPrice(),
            s.getIngredients().stream().map(Ingredient::getName).collect(Collectors.toList())));

    while (page.hasNext()) {
        page = shampooRepository.findAll(page.nextPageable());
        System.out.printf("Page %d of %d:%n-------------%n",
                page.getNumber()+1, page.getTotalPages()    //брои ги от 0, затова добавям + 1
        );

        page.forEach(s -> System.out.printf("%s %s %s %.2f %s%n",
                s.getBrand(), s.getSize(), s.getLabel().getTitle(), s.getPrice(),

s.getIngredients().stream().map(Ingredient::getName).collect(Collectors.toList())));
```

```
    }
}
```

9. Auto Mapping Objects DTO (Data Transfer Object)

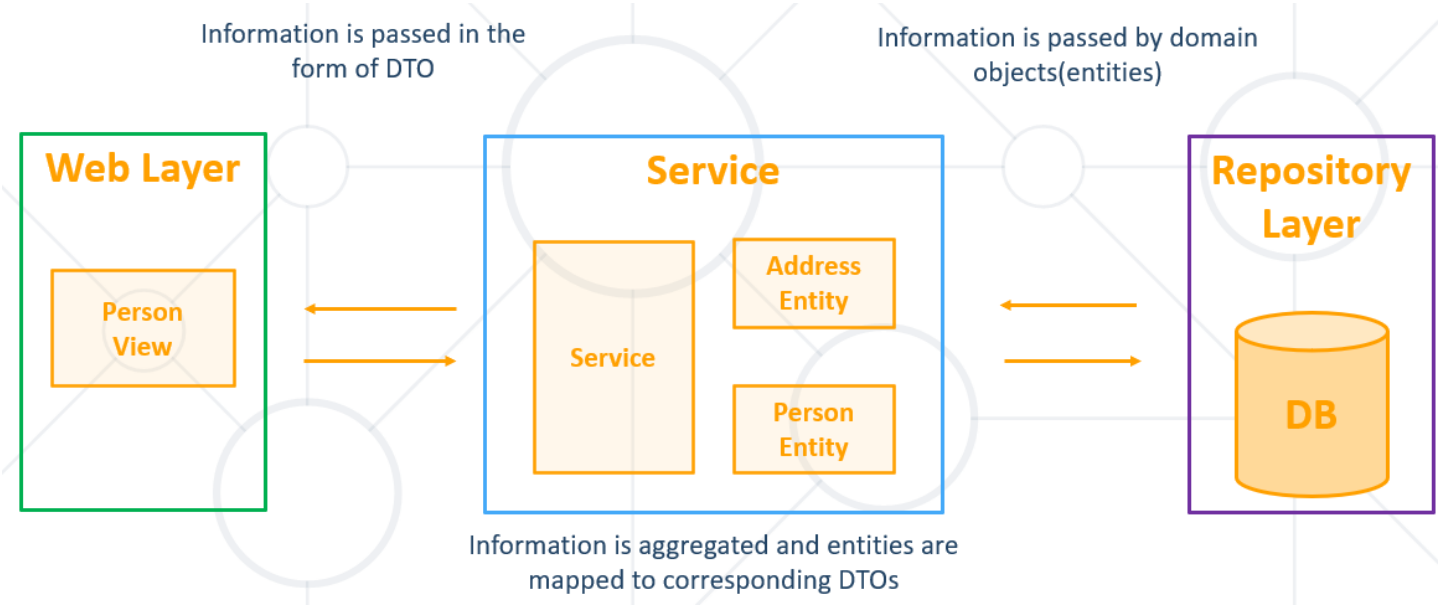
9.1. Data Transfer Objects

Transmitting Aggregated Data from Entities

Data Transfer Object Concept

- Domain objects are mapped to view models – **DTO**s
  - A DTO is a **container class – но само с исканите полета от потребителя**
  - Exposes only properties, **not methods – няма методи, които да вършат нещо**
- In **simple** applications, domain objects can be used in the meaning of DTOs
  - Otherwise, we accomplish nothing but **object replication**

Entity Usage



DTO Usage

Част от полета на различни наши класове, които потребителят иска да вземе като информация от web страницата примерно:

Employee.java

```
@Entity
@Table(name = "employees")
public class Employee {
    //…
    @Column(name = "first_name")
    private String firstName;
    @Column(name = "salary")
    private BigDecimal salary;
    @ManyToOne
    @JoinColumn(name = "address_id")
    private Address address;
    //…}
```

Address.java

```
@Entity
@Table(name = "addresses")
public class Address {
    //…
    @Column
    private String city;
    //…
}
```

EmployeeDTO.java

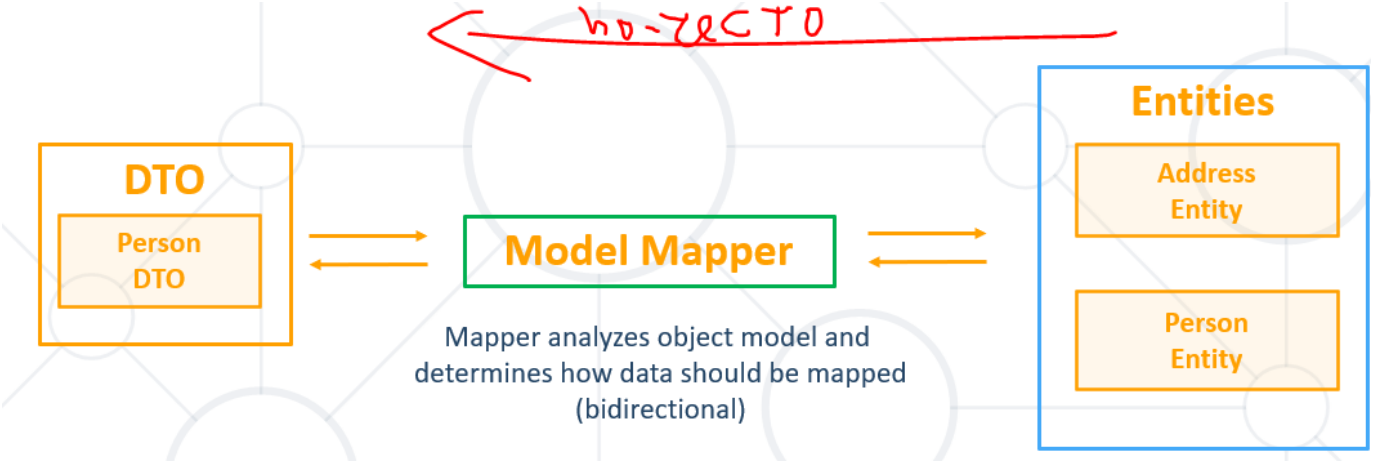
```
public class EmployeeDto {

    private String firstName;
    private BigDecimal salary;
    private String addressCity;
}
```

9.2. Model Mapping - Converting Entity Objects to DTOs

Model Mapping

- We often want to map data between **objects with similar structure**
- Model mapping is an easy way to **convert one model to another**
- Separate **models** must **remain segregated**
- We can **map entity objects to DTOs** using ModelMapper – вземаме обекта/листа от обекти от базата от repository-то , и го мапваме към специфично DTO
- Uses **conventions** to determine how properties and values are mapped to each other

Model Mapper



http://modelmapper.org/getting-started/

И в двете посоки можем да го използваме.

1. Ако вземаме данни от базата, то от Entity към EntityDTO.

2. Ако записваме в базата данни, то от EntityDTO към Entity. (RegisterDTO -> Register) (LoginDTO -> Login).

3. Ако вземаме данни от JSON (файл), то от JSON към EntityDTO.

```java
@Override
public void seedCategories() throws FileNotFoundException {
    FileReader fileReader = new FileReader(CATEGORIES_JSON_PATH);
    CategoryImportDTO[] categoriesImportDTOS = this.gson.fromJson(fileReader,
CategoryImportDTO[].class); //вземи данни от JSON и ги съхрани в масив от DTO-та.

//масива от DTO-та го преобразувай на лист от Обекти, които са с формат тези на базата(в случая
класа Category)
    List<Category> categories = Arrays.stream(categoriesImportDTOS)
            .map(importDTO -> this.modelMapper.map(importDTO, Category.class))
            .collect(Collectors.toList());

    this.categoryRepository.saveAll(categories);
}
```

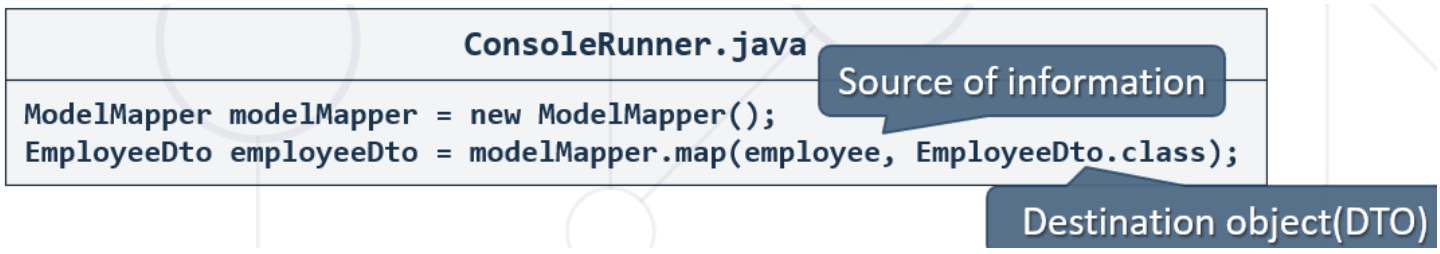
Adding Model Mapper

Add as maven dependency - pom.xml

```xml
<dependency>
    <groupId>org.modelmapper</groupId>
    <artifactId>modelmapper</artifactId>
    <version>3.0.0</version>
</dependency>
```

Create object:

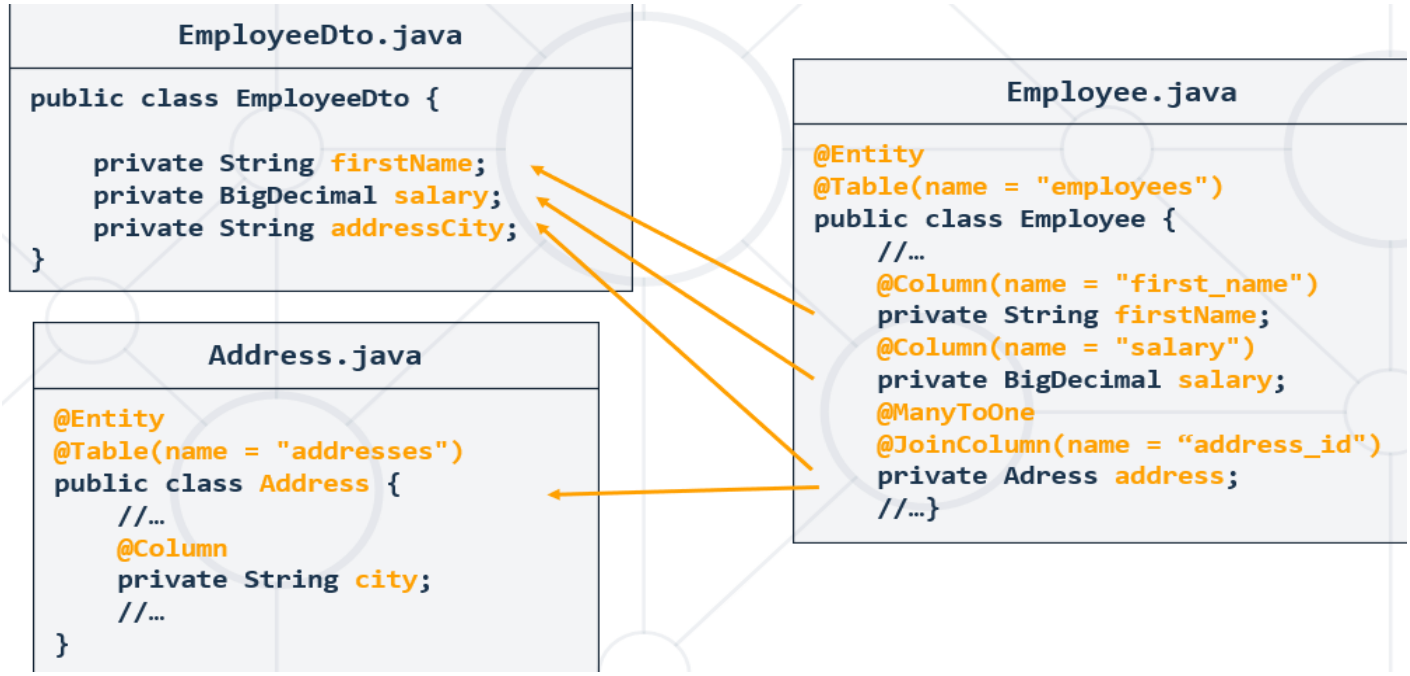
```java
import org.modelmapper.ModelMapper;

ModelMapper mapper = new ModelMapper();

Address address = new Address("boris 3", 3, "Sofia", "Bulgaria");
Employee source = new Employee("first", "last",
BigDecimal.TEN, LocalDate.now(), address);

EmployeeDTO dto = mapper.map(source, EmployeeDTO.class);
}
```



Simple Mapping Entity to DTO

ВАЖНО – да използваме същите имена и в двата класа и то camelCase - за да ги разпознва Spring и modellMapper-а.

От Employee го мапни на DTO EmployDto.java




```
import org.modelmapper.ModelMapper;

public class MapperMain {
    public static void main(String[] args) {
        ModelMapper mapper = new ModelMapper();

        Address address = new Address("boris 3", 3, "Sofia", "Bulgaria");
        Employee source = new Employee("first", "last",
                BigDecimal.TEN, LocalDate.now(), address);

        EmployeeDTO dto = mapper.map(source, EmployeeDTO.class);

        System.out.println(dto);
//EmployeeDTO{firstName='first', lastName='last', salary=10, city='Sofia'}
    }
}
```

```
public class Employee {
    private int id;
    private String firstName;
    private String lastName;
    private BigDecimal salary;
    private LocalDate birthday;
    private Address address;
….
}
```

```
public class Address {
    private int id;
    private String street;
    private int streetNumber;
    private String city;
    private String country;
…..
}
```

```
public class EmployeeDTO {
    private String firstName;
    private String lastName;
    private BigDecimal salary;
    private String addressCity;  //тук го задаваме нарочно address и City слято
…….
```

getters (and setters sometimes) obligatory – сетъри слагаме особено за modellMapper-а
empty constructor forbidden!!! – за .xml и за .json обаче задължително
Normal constructor – maybe no – в единия случай не го хареса, че има конструктор

Class-based Projections (DTOs)7

Another way of defining projections is by using value type DTOs (Data Transfer Objects) that hold properties for the fields that are supposed to be retrieved. These DTO types can be used in exactly the same way projection interfaces are used, except that no proxying happens and no nested projections can be applied.

If the store optimizes the query execution by limiting the fields to be loaded, the fields to be loaded are determined from the parameter names of the constructor that is exposed.

```
…….
}
```

Manual(Explicit) Model Mapping

- ModelMapper uses **conventions** to map objects
  - Sometimes fields differ and mapping **won't be done properly**
  - In this case some manual mapping is needed

Java 6/7

```
ModelMapper modelMapper = new ModelMapper();
PropertyMap<EmployeeDto, Employee> employeeMap = new PropertyMap<EmployeeDto, Employee>() {
    @Override
    protected void configure() {
        map().setFirstName(source.getName());
        // Add mappings for other fields
        map().setAddressCity(source.getAddress().getCity().getName());
    }
};

modelMapper.addMappings(employeeMap).map(employeeDto, employee);
```

Java 8 and later

Пример 1:

```
ModelMapper modelMapper = new ModelMapper();
TypeMap<Employee, EmployeeDto > typeMap = mapper.createTypeMap(Employee.class,
EmployeeDto.class);
typeMap.addMappings(m -> m.map(src -> src.getName(), Employee::setFirstName));
същото като ((destination, value) -> destination.setFirstName(value))));
typeMap.map(employeeDto);
```

Пример 2:

```
        ModelMapper mapper = new ModelMapper();
        TypeMap<Employee, EmployeeSpringDto> typeMap =          //source - target
mapper.createTypeMap(Employee.class, EmployeeSpringDto.class);
```

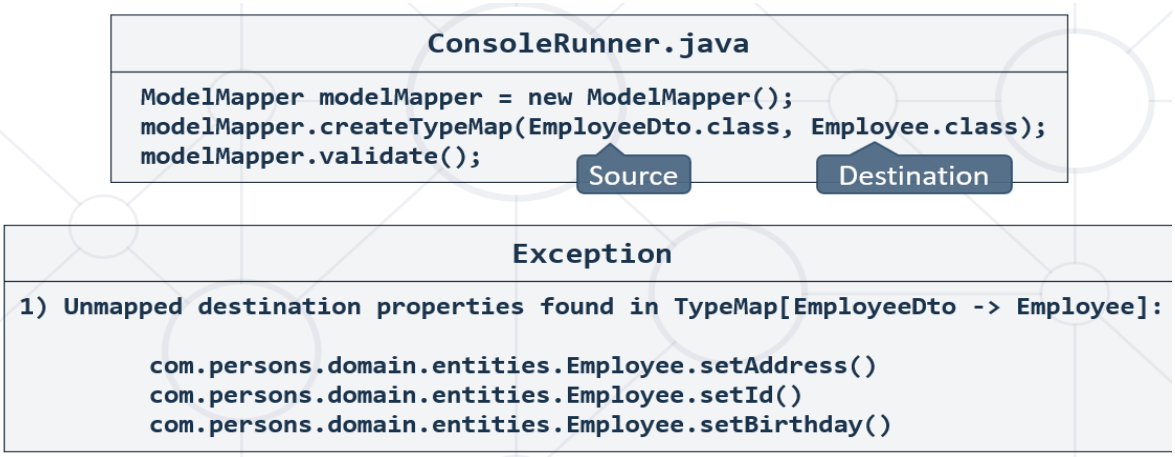
```
typeMap.addMappings(m -> m.map(
        source -> source.getManager().getLastName().length(),
        ((destination, value) -> destination.setManagerLastNameLength(value))));
        същото като      (EmployeeSpringDto::setManagerLastNameLength)));


Минаваме и през Converter, за да не ни кара value да бъде от тип Object
//от Employee към Integer
Converter<Employee, Integer> getLastNameLength =
        ctx -> ctx.getSource() == null ? null : ctx.getSource().getLastName().length();

employeeToCustom.addMappings(mapping ->
        mapping.when(Objects::nonNull)
                .using(getLastNameLength)
                .map(Employee::getManager,
                        EmployeeSpringDto::setManagerLastNameLength)
);


all
        .stream()
        .map(e -> employeeToCustom.map(e))
        .forEach(System.out::println);
```

Пример 3:

```
import org.modelmapper.Conditions;
import org.modelmapper.Converter;
import org.modelmapper.ModelMapper;
import org.modelmapper.Provider;

final ModelMapper modelMapper = new ModelMapper();
final Provider<UserEntity> loggedUserProvider = request -> userService.loggedUser();

modelMapper
        .createTypeMap(UserEntity.class, UserDTO.class)
        .addMappings(mapping -> mapping
                .map(UserEntity::getUsername, UserDTO::setUsername))
        .addMappings(mapping -> mapping
                .map(UserEntity::getFullName, UserDTO::setFullName))
        .addMappings(mapping -> mapping
                .map(UserEntity::getEmail, UserDTO::setEmail))
        .addMappings(mapping -> mapping
                .map(UserEntity::getImageUrl, UserDTO::setImageUrl));
```

Validation

```
ModelMapper modelMapper = new ModelMapper();
modelMapper.createTypeMap(EmployeeDto.class, Employee.class);  //source - target
modelMapper.validate();
```



Валидацията на входните данни е най-ефективна тогава, когато е приложена възможно най-рано в потока от данни на приложението!

Skipping Properties

Java 6/7

```
ModelMapper modelMapper = new ModelMapper();
PropertyMap<EmployeeDto, Employee> employeeMap = new PropertyMap<EmployeeDto, Employee>() {
    @Override
    protected void configure() {
        skip().setSalary(null);  //skipping salary
    }
};

modelMapper.addMappings(employeeMap).map(employeeDto, employee);
```

Java 8 and later

```
typeMap.addMappings(mapper -> mapper.skip(Employee::setSalary));
typeMap.map(employeeDto);
```

Converting Properties

Java 6/7

Пример 1:

```
ModelMapper modelMapper = new ModelMapper();
Converter<String, String> stringConverter = new AbstractConverter<String, String>() {
    @Override
    protected String convert(String s) {
        return s == null ? null : s.toUpperCase();  //Convert Strings to Upper Case
    }
};

PropertyMap<EmployeeDto, Employee> employeeMap = new PropertyMap<EmployeeDto, Employee>() {
    @Override
    protected void configure() {
        using(stringConverter).map().setFirstName(source.getName());  //Use Convertion
    }
};
```

```
modelMapper.addMappings(employeeMap).map(employeeDto, employee);
```

Пример 2:

```
//от User на Стринг
Converter<User, String> userToFullNameConverter = new Converter<User, String>() {
    @Override
    public String convert(MappingContext<User, String> context) {
        return context.getSource() == null ? null : context.getSource().getFullName();
//getSource() реално извиква инстанцията на User класа
    }
};
```

Java 8 and later

Пример 1:

```
ModelMapper modelMapper = new ModelMapper();

//от String в String
Converter<String, String> toUppercase = ctx -> ctx.getSource() == null ? null :
ctx.getSource().toUpperCase();

TypeMap<EmployeeDto, Employee> typeMap = mapper.createTypeMap(EmployeeDto.class, Employee.class)
        .addMappings(mapper -> mapper.using(toUppercase).map(EmployeeDto::getName, Employee::setFirstName));

typeMap.map(employeeDto);
```

Пример 2:

```
//от User на Стринг
Converter<User, String> userToFullNameConverter = context ->
        context.getSource() == null ? null :
context.getSource()       //getSource() реално извиква инстанцията на User класа
.getFullName();
```

Дървен ръчен вариант за mapping – чрез конструктор на DTO класа

```
public class EmployeeSpringDto {
    private String firstName;
    private String lastName;

    public EmployeeSpringDto(Employee employee) {
        this.firstName = employee.getFirstName();
        this.lastName = employee.getLastName();
    }
```

CustomModelMapper, TypeMap, Converter, addMappings using the converter

Да използваме method reference пише в документацията за вътрешната команда.map

```
@Service
public class ProductServiceImpl implements ProductService {
    private final ProductRepository productRepository;
    private final ModelMapper mapper;
    private final TypeMap<Product, ProductWithAttributesDTO> productToDtoMapping;
```

```
    @Autowired
    public ProductServiceImpl(ProductRepository productRepository) {
        this.productRepository = productRepository;
        this.mapper = new ModelMapper();

        //seller -> firstName + lastName
        //от User на Стринг
        Converter<User, String> userToFullNameConverter = context ->
                context.getSource() == null ? null : context.getSource().getFullName();
//getSource() реално извиква инстанцията на User класа

        TypeMap<Product, ProductWithAttributesDTO> typeMap =
this.mapper.createTypeMap(Product.class, ProductWithAttributesDTO.class);

        this.productToDtoMapping = typeMap.addMappings(m -> m
                .using(userToFullNameConverter)
                .map(Product::getSeller, ProductWithAttributesDTO::setSeller));

или

        // <V> void map(SourceGetter<S> sourceGetter, DestinationSetter<D, V> destinationSetter);
        this.productToDtoMapping = typeMap.addMappings(m -> m
//                .when(Objects::nonNull)
                .using(userToFullNameConverter)
//                .map(Product::getSeller, ProductWithAttributesDTO::setSeller));
                .map(Product::getSeller, ((destination, value) -> destination.setSeller((String)
value))));


        this.mapper.addConverter(userToFullNameConverter);
    }


    @Override
    public ExportProductsInRangeDTO getInRange(float from, float to) {
        BigDecimal rangeFrom = BigDecimal.valueOf(from);
        BigDecimal rangeTo = BigDecimal.valueOf(to);

        List<Product> products =
this.productRepository.findAllByPriceBetweenAndBuyerIsNullOrderByPriceDesc(rangeFrom, rangeTo);
        List<ProductWithAttributesDTO> dtos = products
                .stream()
                .map(p -> this.productToDtoMapping.map(p))
                .collect(Collectors.toList());

        return new ExportProductsInRangeDTO(dtos);
    }
}



@Entity
@Table(name = "users")
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;

    @Column(name = "first_name", length = 255)
```

```
    private String firstName;

    @Column(name = "last_name", nullable = false, length = 255)
    private String lastName;

    private Integer age;

    @OneToMany(targetEntity = Product.class, mappedBy = "seller")
    private List<Product> sellingItems;

    @OneToMany(targetEntity = Product.class, mappedBy = "buyer")
    private List<Product> itemsBought;

    @ManyToMany
    private Set<User> friends;


    public User() {
        this.sellingItems = new ArrayList<>();
        this.itemsBought = new ArrayList<>();
        this.friends = new HashSet<>();
    }

    public User(String firstName, String lastName, Integer age) {
        this();

        this.firstName = firstName;
        this.lastName = lastName;
        this.age = age;
    }

    public String getFullName() {
        if (this.firstName == null) {
            return this.lastName;
        } else {
            return this.firstName + " " + this.lastName;
        }
    }
}
```

ModelMapper and addConverter on it – from String to Date

Добавяне на един или повече converter-и на нашия modelMapper – като срещне поле от единия клас от String в другия клас Date, и да си се оправя/конвъртва.

Java 8 and later

```
private final ModelMapper modelMapper;


//this works :)
this.modelMapper.addConverter((Converter<String, LocalDate>)
        ctx -> LocalDate.parse(ctx.getSource(), DateTimeFormatter.ofPattern("dd/MM/yyyy")),
        String.class, LocalDate.class);

//this also works :)
this.modelMapper.addConverter(ctx -> LocalDate.parse(ctx.getSource(),
DateTimeFormatter.ofPattern("dd/MM/yyyy")),
        String.class, LocalDate.class);
```

Java 6-7

//this one also works

```
@Bean  //@Component  също върши работа ако е главен клас
public ModelMapper modelMapper() {
    ModelMapper modelMapper = new ModelMapper();

    modelMapper.addConverter(new Converter<String, LocalDate>() {
        @Override
        public LocalDate convert(MappingContext<String, LocalDate> mappingContext) {
            return LocalDate.parse(mappingContext.getSource(),
DateTimeFormatter.ofPattern("dd/MM/yyyy"));
        }
    });

    modelMapper.addConverter(new Converter<String, LocalDateTime>() {
        @Override
        public LocalDateTime convert(MappingContext<String, LocalDateTime> mappingContext) {
            return LocalDateTime.parse(mappingContext.getSource(),
DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss"));
        }
    });

    return modelMapper;
}
```

10. JSON

Exporting and Importing Data from JSON Format

https://www.json.org/json-en.html

1. JSON

Transmitting Data Objects Via Attribute-value Pairs

JSON служи най-вече предаване на данни. Например в браузъра.
Но е възможно да служи и като настройка файл за дадена програма, или за извличане от SQL бази данни или за съхранение на данни в браузъра (в случаите когато данни не се предават по мрежата, а се генерират в бразуъра от интеракцията с потребителя).
Можем да пазим и log файлове с JSON формат.

JSON не поддържа хипервръзки. Използва се по-скоро за дървовидна структура

JSON

- **J**ava**S**cript **O**bject **N**otation
  - Human-readable format to transmit **data objects** consisting of **attribute–value pairs** and **arrays**
  - Subset of **JavaScript** syntax
- Supports several data types:
  - Number, String, Boolean, Array, Object, null

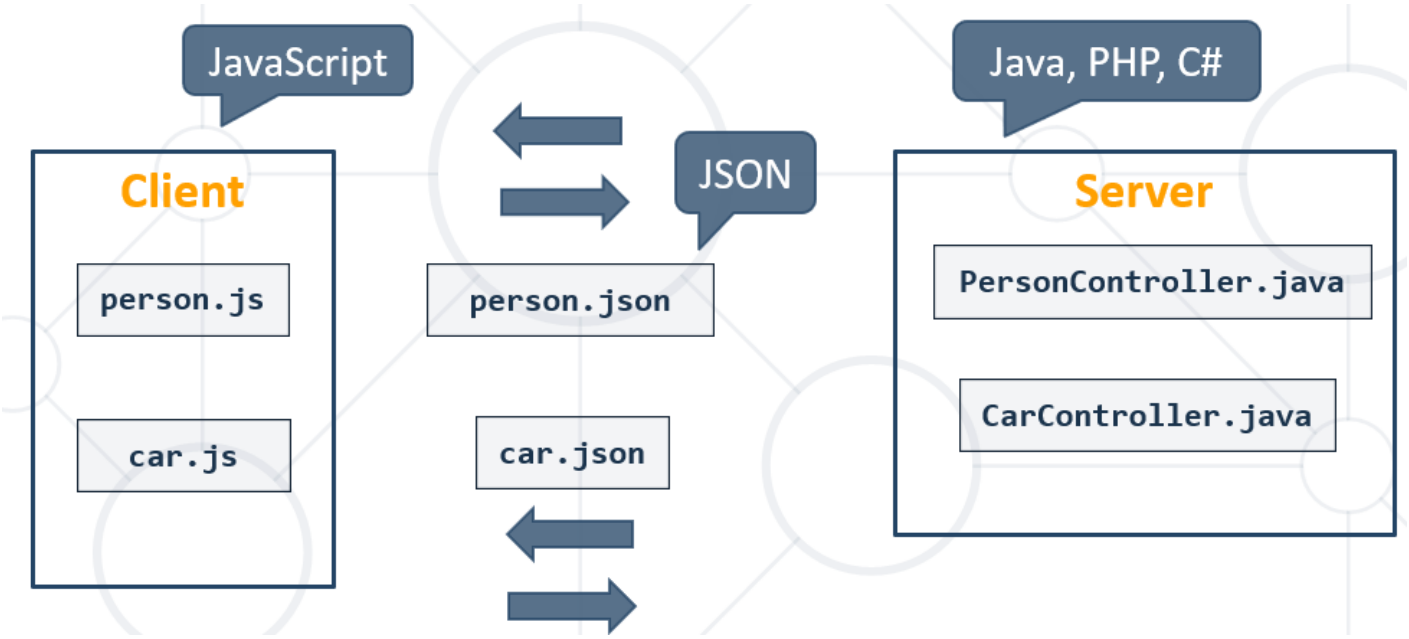
Key-Value

```
{
  "firstName": "Daniel",
  "lastName": "Sempre",
  "age": 24,
  "isMarried": true
}
```

With Array type value:

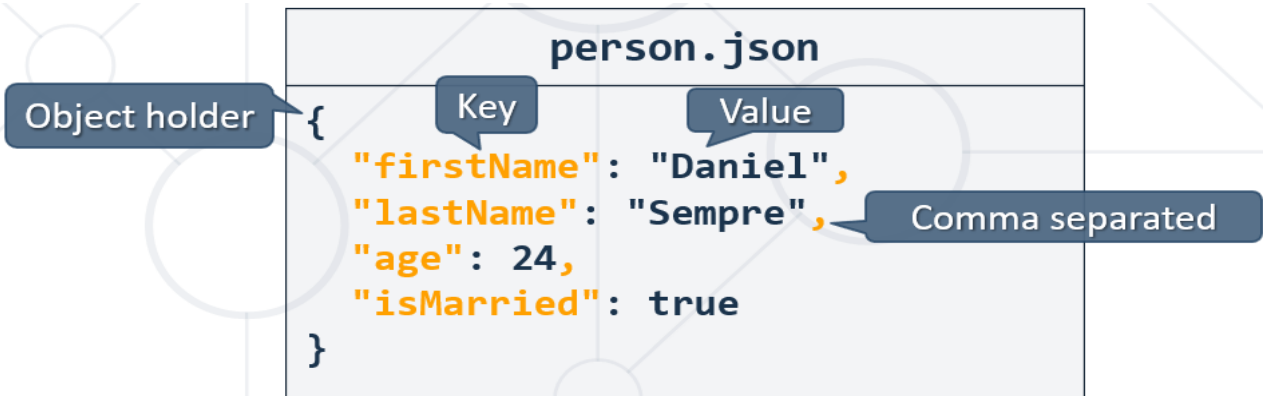
```
{
  "firstName": "Daniel",
  "lastName": "Sempre",
  "age": 24,
  "courses": [
    {
      "name": "Java DB"         no comma here
    },
    {
      "name": "HTML"              no comma here

    }
  ]
}
```

JSON Function

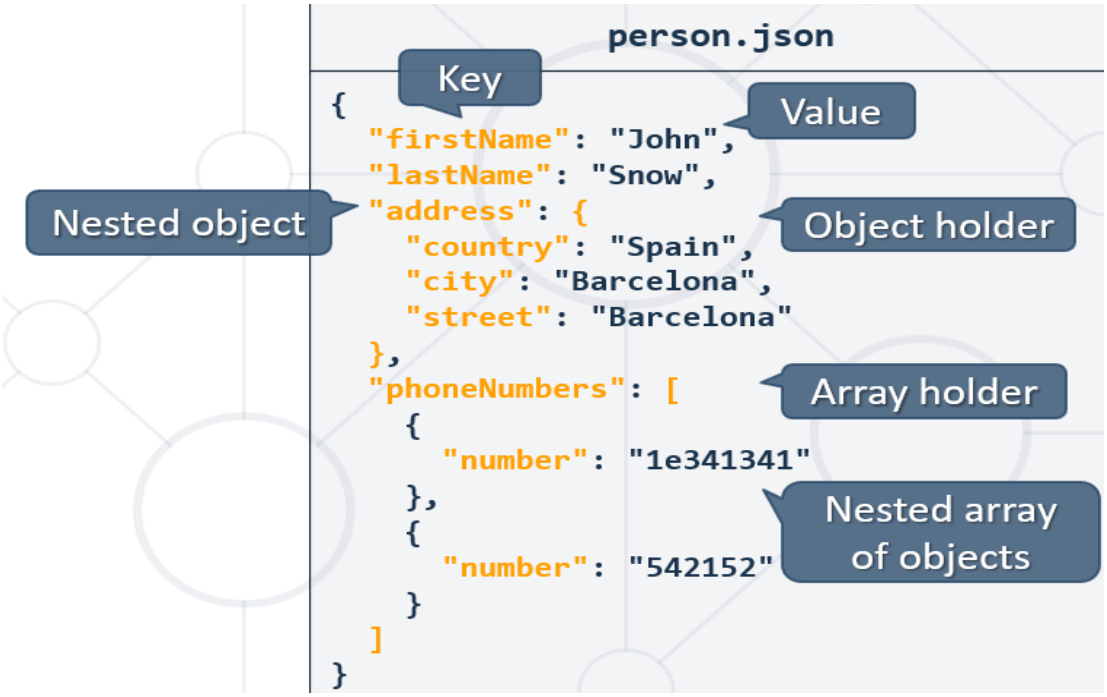


JSON Structure

- Data is represented in **name/value** pairs
- Curly braces hold objects
- Square brackets hold **arrays**




```
{
  "firstName": "John",
  "lastName": "Snow",
  "address": {
    "country": "Spain",
    "city": "Barcelona",
    "street": "Barcelona"
  },
  "phoneNumbers": [
    {
      "number": "1e341341"
    },
    {
      "number": "542152"
    }
  ]
}
```



2. GSON parser

Serialize and De-serialize objects with Java

GSON parser intro

- Java library to operate with JSON files
- Provides easy to use mechanisms to convert **Java to JSON** and vice-versa
  - Originally developed by Google
- Generates compact and readability JSON output

pom.xml

```
<dependencies>
    <!-- https://mvnrepository.com/artifact/com.google.code.gson/gson -->
    <dependency>
        <groupId>com.google.code.gson</groupId>
        <artifactId>gson</artifactId>
        <version>2.9.0</version>
    </dependency>

</dependencies>
```

GSON Initialization

- Gson objects are responsible for the JSON manipulations
  - GsonBuilder creates an instance of GSON
  - **excludeFieldsWithoutExposeAnnotation()** – excludes fields without **@Expose** annotation
  - **setPrettyPrinting()** – aligns and justifies the created JSON format
  - **create()** – creates an instance of Gson

```
import com.google.gson.Gson;
import com.google.gson.GsonBuilder;

public class Main {
    public static void main(String[] args) {
        Gson gson = new GsonBuilder()
                .excludeFieldsWithoutExposeAnnotation()
                .setPrettyPrinting()
                .create();
    }
}
```

Serializartion and Deserialization

Сериализираме от обекта към JSON string пренос формата на данни
Десериализираме от JSON string към съответния искан формат на обекта

Expose варианти:

```
@Expose(serialize = false, deserialize = true) //от обекта към JSON стринг не го вземай това поле
@NonNull  //от lombok
@NotNull          //jakarta validation   javax.validation.constraints на Java класа полето да не
e null, Accepts any type, including Enum
@NotBlank - //jakarta validation   javax.validation.constraints The annotated element must not be
null and must contain at least one non-whitespace character. Accepts any type, including Enum

@Length(min = 4, max = 30)   //полето в JAVA средата да е с тези ограничения; @Length не включва
notNull, org.hibernate.validator.constraints.Length; - in pom.xml file added
```

```
@Column(unique = true, nullable = false) //за базата SQL ограничение при създаването
private String password;
```

Export Single Object to JSON (Serializartion)

```
import com.google.gson.annotations.Expose;

import java.io.Serializable;

public class AddressJsonDto implements Serializable {
    @Expose  //The field will be imported/exported
    private String country;
    @Expose //The field will be imported/exported
    private String city;
    @Expose  //The field will be imported/exported
    private String street;
}
```

```
import com.google.gson.Gson;
import com.google.gson.GsonBuilder;

public class Main {
    public static void main(String[] args) {
        Gson gson = new GsonBuilder()
                .excludeFieldsWithoutExposeAnnotation()
                .setPrettyPrinting()
                .create();

        AddressJsonDto addressJsonDto = new AddressJsonDto();
        addressJsonDto.setCountry("Bulgaria");
        addressJsonDto.setCity("Sofia");
        addressJsonDto.setStreet("Mladost 4");
        String content = gson.toJson(addressJsonDto);

        System.out.println(content);
    }
}
```

Отпечатва се:

```
{
  "country": "Bulgaria",
  "city": "Sofia",
  "street": "Mladost 4"
}
```

Export Multiple Object to JSON (Serializartion)

```
import com.google.gson.annotations.Expose;

import java.io.Serializable;

public class AddressJsonDto implements Serializable {
    @Expose  //The field will be imported/exported
    private String country;
```

```
    @Expose //The field will be imported/exported
    private String city;
    @Expose  //The field will be imported/exported
    private String street;
}



import com.google.gson.Gson;
import com.google.gson.GsonBuilder;

import java.util.ArrayList;
import java.util.List;

public class Main {
    public static void main(String[] args) {
        Gson gson = new GsonBuilder()
                .excludeFieldsWithoutExposeAnnotation() //ако използваме expose на полетата на
класа
                .setPrettyPrinting()
                .create();

        AddressJsonDto addressJsonDtoBulgaria = new AddressJsonDto();
        addressJsonDtoBulgaria.setCountry("Bulgaria");
        addressJsonDtoBulgaria.setCity("Sofia");
        addressJsonDtoBulgaria.setStreet("Mladost 4");

        AddressJsonDto addressJsonDtoSpain = new AddressJsonDto();
        addressJsonDtoSpain.setCountry("Spain");
        addressJsonDtoSpain.setCity("Barcelona");
        addressJsonDtoSpain.setStreet("Las Ramblas");

        List<AddressJsonDto> addressJsonDtos = new ArrayList<>();
        addressJsonDtos.add(addressJsonDtoBulgaria);
        addressJsonDtos.add(addressJsonDtoSpain);
        String content = gson.toJson(addressJsonDtos);

        System.out.println(content);
    }
}
```

Отпечатва се:

```
[
 {
  "country": "Bulgaria",
  "city": "Sofia",
  "street": "Mladost 4"
 },
 {
  "country": "Spain",
  "city": "Barcelona",
  "street": "Las Ramblas"
 }
]
```

Export Nested Multiple Object to JSON (Serializartion)

```
public class CustomerOutputToJSONDto {
    private String firstName;
    private String lastName;
    private String email;
    private TownJsonDto town;

        public CustomerOutputToJSONDto() {
      }
}

public class TownJsonDto {
    private String name;

    public TownJsonDto() {
    }
}

public class Main {
    public static void main(String[] args) throws IOException {
        Gson gson = new GsonBuilder()
//                .excludeFieldsWithoutExposeAnnotation() //ако използваме expose на полетата на
класа
                .setPrettyPrinting()
                .create();

        CustomerOutputToJSONDto entry1 = new CustomerOutputToJSONDto();
        entry1.setFirstName("Ivan");
        entry1.setLastName("Petkov");
        entry1.setEmail("ivan@abv.bg");
        TownJsonDto tn1 = new TownJsonDto();
        tn1.setName("Pazardjik");
        entry1.setTown(tn1);

        CustomerOutputToJSONDto entry2 = new CustomerOutputToJSONDto();
        entry2.setFirstName("Svilen");
        entry2.setLastName("Velikov");
        entry2.setEmail("svilen@abv.bg");
        TownJsonDto tn2 = new TownJsonDto();
        tn2.setName("Shumen");
        entry2.setTown(tn2);

        List<CustomerOutputToJSONDto> toJSONDtos = new ArrayList<>();
        toJSONDtos.add(entry1);
        toJSONDtos.add(entry2);

        String content = gson.toJson(toJSONDtos);

        Path path = Path.of("src/main/resources/output/outputToJSON.json");

        FileWriter fileWriter = new FileWriter(String.valueOf(path));
        fileWriter.write(content);
        fileWriter.close();
    }
}
```

Изхода е следния:

```
[
  {
```

```
    "firstName": "Ivan",
    "lastName": "Petkov",
    "email": "ivan@abv.bg",
    "town": {
      "name": "Pazardjik"
    }
  },
  {
    "firstName": "Svilen",
    "lastName": "Velikov",
    "email": "svilen@abv.bg",
    "town": {
      "name": "Shumen"
    }
  }
]
```

Export via appendable one or more Objects or nested objects

```
@GetMapping("/export/employees-above")
public ModelAndView showEmployeesAbove25(){
    ModelAndView modelAndView = new ModelAndView("export/export-employees-with-age");

    List<ExportEmployeeDTO> employeesAbove25 = this.employeeService.getEmployeesAbove25();

    StringBuilder sb = new StringBuilder();
    this.gson.toJson(employeesAbove25, sb);   //това е void!!!!

    modelAndView.addObject("employeesAbove", sb.toString());

    return modelAndView;
}
```

Import Single Object from JSON (Deserializartion)

```
import com.google.gson.annotations.Expose;

import java.io.Serializable;

public class AddressJsonDto implements Serializable {
    @Expose  //The field will be imported/exported
    private String country;
    @Expose //The field will be imported/exported
    private String city;
    @Expose  //The field will be imported/exported
    private String street;
}
```

```
import com.google.gson.Gson;
import com.google.gson.GsonBuilder;

public class Main {
    private static final String jsonText = """     //text block from Java 15
    {
        "country": "Bulgaria",
        "city": "Sofia",
        "street": "Mladost 4"
```

```
    }""";

    public static void main(String[] args) {
        Gson gson = new GsonBuilder()
                .excludeFieldsWithoutExposeAnnotation()
                .setPrettyPrinting()
                .create();


        AddressJsonDto addressJsonDto =
                gson.fromJson(jsonText, AddressJsonDto.class);

    }
}
```

Import Multiple Object from JSON (Deserializartion)

```
import com.google.gson.annotations.Expose;

import java.io.Serializable;

public class AddressJsonDto implements Serializable {
    @Expose  //The field will be imported/exported
    private String country;
    @Expose //The field will be imported/exported
    private String city;
    @Expose  //The field will be imported/exported
    private String street;
}
```

```
import com.google.gson.Gson;
import com.google.gson.GsonBuilder;

public class Main {
    private static final String jsonText = """     //text block from Java 15
           [
              {
                "country": "Bulgaria",
                "city": "Sofia",
                "street": "Mladost 4"
              },
              {
                "country": "Spain",
                "city": "Barcelona",
                "street": "Las Ramblas"
              }
           ]""";

    public static void main(String[] args) {
        Gson gson = new GsonBuilder()
                .excludeFieldsWithoutExposeAnnotation()
                .setPrettyPrinting()
                .create();
```

```
        // Object Array
        AddressJsonDto[] addressJsonDtos =
                gson.fromJson(jsonText, AddressJsonDto[].class);

    }
}
```

Import Nested Multiple Object from JSON (Deserializartion)

```
[
  {
    "firstName": "A",
    "lastName": "Duell",
    "email": "aduell0@nifty.com",
    "registeredOn": "26/07/2020",
    "town": {
      "name": "Rome"
    }
  },
  {
    "firstName": "Lorne",
    "lastName": "C",
    "email": "lcurtayne1@nbcnews.com",
    "registeredOn": "27/06/2020",
    "town": {
      "name": "Kyiv"
    }
  }
]
```

```
public class CustomerSeedFromJSONDto {
    @Size(min = 2)
    private String firstName;

    @Size(min = 2)
    private String lastName;

    @Email
    private String email;

    private String registeredOn;

    private TownJsonDto town;

    public CustomerSeedFromJSONDto() {
    }
}


public class TownJsonDto {
    private String name;

    public TownJsonDto() {
    }
}

@Service
public class CustomerServiceImpl implements CustomerService {
    private static final String CUSTOMERS_FILE_PATH =
"src/main/resources/files/json/customers.json";
```

```
    private final CustomerRepository customerRepository;
    private final TownService townService;

    private final Gson gson;
    private final ModelMapper modelMapper;
    private final ValidationUtil validationUtil;

    @Autowired
    public CustomerServiceImpl(CustomerRepository customerRepository, TownService townService,
Gson gson, ModelMapper modelMapper, ValidationUtil validationUtil) {
        this.customerRepository = customerRepository;
        this.townService = townService;
        this.gson = gson;
        this.modelMapper = modelMapper;
        this.validationUtil = validationUtil;
    }

    @Override
    public boolean areImported() {
        return this.customerRepository.count() > 0;
    }

    @Override
    public String readCustomersFileContent() throws IOException {
        return Files.readString(Path.of(CUSTOMERS_FILE_PATH));
    }

    @Override
    public String importCustomers() throws IOException {
        CustomerSeedFromJSONDto[] customersSeedDtos =
gson.fromJson(this.readCustomersFileContent(), CustomerSeedFromJSONDto[].class);

        return Arrays.stream(customersSeedDtos)
                .map(this::importOneCustomer)
                .collect(Collectors.joining("\n"));
    }

    private String importOneCustomer(CustomerSeedFromJSONDto dto) {
        boolean isValid = this.validationUtil.isValid(dto);
        if (!isValid) {
            return "Invalid customer";
        }

        Optional<Customer> optCustomer = this.customerRepository.findByEmail(dto.getEmail());
        if (optCustomer.isPresent()) {
            return "Invalid customer";  //customer already exists
        }

        Optional<Town> optTown = this.townService.findTownByName(dto.getTown().getName());

        Customer customer = this.modelMapper.map(dto, Customer.class);
        customer.setTown(optTown.get());
        this.customerRepository.save(customer);

        return "Successfully imported Customer " +
                dto.getFirstName() + " " + dto.getLastName() + " - " + dto.getEmail();
    }
}
```

```java
@Configuration
//Indicates that a class declares one or more @Bean methods and may be processed by the Spring
container to generate bean definitions and service requests for those beans at runtime, for
example:
public class ApplicationBeanConfiguration {
    @Bean   //Indicates that a method produces a bean to be managed by the Spring container
    public Gson gson() {
        return new GsonBuilder()
//                .excludeFieldsWithoutExposeAnnotation()
                .setPrettyPrinting()
                .create();
    }

    @Bean //Indicates that a method produces a bean to be managed by the Spring container
    public ModelMapper modelMapper() {
        ModelMapper modelMapper = new ModelMapper();

        modelMapper.addConverter(new Converter<String, LocalDate>() {
            @Override
            public LocalDate convert(MappingContext<String, LocalDate> mappingContext) {
                return LocalDate.parse(mappingContext.getSource(),
DateTimeFormatter.ofPattern("dd/MM/yyyy"));
            }
        });

        modelMapper.addConverter(new Converter<String, LocalDateTime>() {
            @Override
            public LocalDateTime convert(MappingContext<String, LocalDateTime> mappingContext) {
                return LocalDateTime.parse(mappingContext.getSource(),
DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss"));
            }
        });

modelMapper.addConverter(new Converter<String, LocalTime>() {
    @Override
    public LocalTime convert(MappingContext<String, LocalTime> mappingContext) {
        return LocalTime.parse(mappingContext.getSource(),
DateTimeFormatter.ofPattern("HH:mm:ss"));
    }
});


        return modelMapper;
    }
}
```

GSON and Projection

GSON does not work with interface type projections!!!!

TypeToken class helping GSON when serialize/deserialize collection of generic type object

3. Примерен проект – две свързани DTO-та

Задача: Get all users who have **at least 1 item sold** with a **buyer**. Order them by **last name**, then by **first name**. Select the person's **first** and **last name**. For each of the **products sold** (products with buyers), select the product's **name**, **price** and the buyer's **first** and **last name**.

```
public class SoldProductDTO {
    private String name;
    private BigDecimal price;
    private String buyerFirstName;
    private String buyerLastName;
}
```

```
public class UserWithSoldProductsDTO {
    private String firstName;
    private String lastName;
    private List<SoldProductDTO> itemsBought;
}
```

```
@Repository
public interface UserRepository extends JpaRepository<User, Integer> {
    @Query("SELECT u FROM User u" +
            " JOIN u.sellingItems AS pr" +
            " WHERE pr.buyer IS NOT NULL")
    List<User> findAllWithSoldProducts();
}
```

```
public interface UserService {
    List<UserWithSoldProductsDTO> getUsersWithSoldProducts();
}
```

```
@Service
public class UserServiceImpl implements UserService {
    private final UserRepository userRepository;
    private final ModelMapper modelMapper;

    @Autowired
    public UserServiceImpl(UserRepository userRepository) {
        this.userRepository = userRepository;
        modelMapper = new ModelMapper();
    }

    @Override
    @Transactional
    public List< UserWithSoldProductsDTO> getUsersWithSoldProducts() {
        List<User> allWithSoldProducts = this.userRepository.findAllWithSoldProducts();

        List<UserWithSoldProductsDTO> allMappedToDTO = allWithSoldProducts.stream()
                .map(u -> this.modelMapper.map(u, UserWithSoldProductsDTO.class))
                .collect(Collectors.toList());

        return allMappedToDTO;
    }
}
```

```
@Component
public class ProductShopRunner implements CommandLineRunner {
    private final SeedService seedService;
    private final ProductService productService;
    private UserService userService;

    private final Gson gson;


    @Autowired
    public ProductShopRunner(SeedService seedService, ProductService productService, UserService
userService) {
        this.seedService = seedService;
        this.productService = productService;
        this.userService = userService;
        gson = new GsonBuilder()
                .setPrettyPrinting()
                .create();
    }

    @Override
    public void run(String... args) throws Exception {
//        this.seedService.seedAll();
//        _01_Query_ProductsInRange();
        _02_Query_SuccessfullySoldProducts();
//        _03_Query_CategoriesByProductsCount();
//        _04_Query_UsersAndProducts();
    }

    private void _02_Query_SuccessfullySoldProducts() {
        List<UserWithSoldProductsDTO> usersWithSoldProducts =
this.userService.getUsersWithSoldProducts();
        String jsonContent = this.gson.toJson(usersWithSoldProducts);
        System.out.println(jsonContent);
    }
}
```

Вариант с изход във файл:

```
@Component
public class CommandLineRunnerImpl implements CommandLineRunner {

    private final CategoryService categoryService;
    private final UserService userService;
    private final ProductService productService;
    private final BufferedReader bufferedReader;
    private final Gson gson;
    private static final String JSON_OUT_PATH = "src/main/resources/files/out/";
    private static final String SOLD_PRODUCTS_FILE_NAME = "sold-products.json";

    public CommandLineRunnerImpl(CategoryService categoryService, UserService userService,
ProductService productService, Gson gson) {
        this.categoryService = categoryService;
        this.userService = userService;
        this.productService = productService;
        this.gson = gson;
        this.bufferedReader = new BufferedReader(new InputStreamReader(System.in));
    }
```

```
    @Override
    public void run(String... args) throws Exception {
        seedData();
        System.out.println("Enter query number (1-4)");
        String input = bufferedReader.readLine();
        switch (input) {
            case "1" -> productsInRange();
            case "2" -> soldProducts();
            case "3" -> categoriesCount();
            case "4" -> userWithSoldProducts();
        }

    }


    private void soldProducts() throws IOException {
        List<UserSellerDto> userSellerDtos = userService.getAllUsersWithAtLeastOneProductSold();
        String infoOut = gson.toJson(userSellerDtos);
        Files.writeString(Path.of(JSON_OUT_PATH + SOLD_PRODUCTS_FILE_NAME), infoOut);
    }
}
```

4. Annotation Type SerializedName

https://www.javadoc.io/doc/com.google.code.gson/gson/2.6.2/com/google/gson/annotations/SerializedName.html

An annotation that indicates this member should be serialized to JSON with the provided name value as its field name.

Here is an example of how this annotation is meant to be used:

```
public class MyClass {
  @SerializedName("name") String a;
  @SerializedName(value="name1", alternate={"name2", "name3"}) String b;
  String c;

  public MyClass(String a, String b, String c) {
    this.a = a;
    this.b = b;
    this.c = c;
  }
}
```

The following shows the output that is generated when serializing an instance of the above example class:

```
MyClass target = new MyClass("v1", "v2", "v3");
Gson gson = new Gson();
String json = gson.toJson(target);
System.out.println(json);

===== OUTPUT =====
{"name":"v1","name1":"v2","c":"v3"}
```

NOTE: The value you specify in this annotation must be a valid JSON field name.

While deserializing, all values specified in the annotation will be deserialized into the field. For example:

```
MyClass target = gson.fromJson("{'name1':'v1'}", MyClass.class);
assertEquals("v1", target.b);
target = gson.fromJson("{'name2':'v2'}", MyClass.class);
assertEquals("v2", target.b);
target = gson.fromJson("{'name3':'v3'}", MyClass.class);
assertEquals("v3", target.b);
```

Note that MyClass.b is now deserialized from either name1, name2 or name3.

5. JACKSON parser intro – part of MVC Spring

Dependencies

```
<dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-databind</artifactId>
    <version>2.13.0</version>
</dependency>
```

https://www.baeldung.com/jackson-object-mapper-tutorial

```
import com.fasterxml.jackson.annotation.JsonIgnore;
import com.fasterxml.jackson.annotation.JsonProperty;
import com.fasterxml.jackson.annotation.JsonView;
import com.google.gson.annotations.Expose;
import lombok.*;
import org.hibernate.validator.constraints.Length;
import org.hibernate.validator.constraints.URL;

import javax.persistence.*;
import javax.validation.constraints.NotNull;

import java.util.ArrayList;
import java.util.Collection;
import java.util.Date;

import static javax.persistence.GenerationType.IDENTITY;


@Entity
@Data
@NoArgsConstructor
@RequiredArgsConstructor
@AllArgsConstructor
@EqualsAndHashCode(onlyExplicitlyIncluded = true)
public class User {
    public static final String ROLE_USER = "ROLE_USER";
    public static final String ROLE_ADMIN = "ROLE_ADMIN";

//    @JsonView(Views.Post.class)
    @Expose
    @Id
    @GeneratedValue(strategy = IDENTITY)
    @EqualsAndHashCode.Include
    private Long id;

//    @JsonView(Views.Post.class)
    @Expose
    @NonNull
    @Length(min = 2, max = 60)
```

```
    private String firstName;

//    @JsonView(Views.Post.class)
    @Expose
    @NonNull
    @Length(min = 2, max = 60)
    private String lastName;

    @Expose
    @NonNull
    @Length(min = 3, max = 60)
    @Column(unique = true, nullable = false)
    @NotNull
    @EqualsAndHashCode.Include
    private String username;

    @Expose(serialize = false)
    @NonNull
    @Length(min = 4, max = 80)
    @Column(nullable = false)
    @NotNull
    @JsonProperty(access = JsonProperty.Access.WRITE_ONLY)   //Jackson
    private String password;

    @Expose
    @NonNull
    @NotNull
    private String roles;

//    @JsonView(Views.Post.class)
    @Expose
    @NonNull
    @Length(min=8, max=512)
    @URL
    private String imageUrl;

    private boolean active = true;

    @OneToMany(mappedBy = "author")
    @ToString.Exclude
    @JsonIgnore   //Jackson way   com.fasterxml.jackson.annotation;
    @Expose(serialize = false, deserialize = false)
    private Collection<Post> posts = new ArrayList<Post>();

    @Expose
    private Date created = new Date();
    @Expose
    private Date modified = new Date();
}


import com.example.restdemo.model.Post;
import com.example.restdemo.service.PostService;
import com.example.restdemo.service.UserService;
import lombok.extern.slf4j.Slf4j;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;
import org.springframework.web.servlet.mvc.method.annotation.MvcUriComponentsBuilder;

import java.net.URI;
```

```java
import java.util.Collection;

//The Jackson way
@RestController    //се сериализира като Response body
@RequestMapping("api/posts")   //http://localhost:8080/api/posts
@CrossOrigin("http://localhost:3000")  //match other port
@Slf4j
public class PostController {
    @Autowired
    private PostService postService;

    @Autowired
    private UserService userService;

    @GetMapping
    public Collection<Post> getPosts() {
        return postService.getPosts();
    }

    @PostMapping
    public ResponseEntity<Post> addPost(@RequestBody Post post){
        Post created = postService.createPost(post);
        URI location = MvcUriComponentsBuilder.fromMethodName(PostController.class, "addPost",
post)
                .pathSegment("{id}").buildAndExpand(created.getId()).toUri();

        log.info("New Post created: {}", created);
        return ResponseEntity.created(location).body(created);
    }
}


import com.example.restdemo.gson.PostGsonDeserializer;
import com.example.restdemo.gson.PostGsonSerializer;
import com.example.restdemo.model.Post;
import com.example.restdemo.service.PostService;
import com.google.gson.Gson;
import com.google.gson.GsonBuilder;
import lombok.extern.slf4j.Slf4j;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.MediaType;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;
import org.springframework.web.servlet.mvc.method.annotation.MvcUriComponentsBuilder;
import org.springframework.web.servlet.support.ServletUriComponentsBuilder;

import javax.annotation.PostConstruct;
import java.net.URI;

//The Gson way
@CrossOrigin(origins = "http://localhost:4000", maxAge = 3600)
@RestController    //org.springframework.web.bind.annotation
@RequestMapping("/api/simple")    //org.springframework.web.bind.annotation
//http://localhost:8080/api/simple
@Slf4j
public class SimplePostController {
    @Autowired
    private PostService postService;

    private Gson gson;
```

```
    @PostConstruct   //javax.annotation - one such method only in a class can be annotated with
this annotation
    private void init() {
        gson = new GsonBuilder()
                .excludeFieldsWithoutExposeAnnotation()
                .setPrettyPrinting()
                .registerTypeAdapter(Post.class, new PostGsonSerializer())
                .registerTypeAdapter(Post.class, new PostGsonDeserializer())
                .create();
    }

    //org.springframework.web.bind.annotation
    //org.springframework.http
    @GetMapping(produces = MediaType.APPLICATION_JSON_VALUE)
    public String getPosts() {
        return gson.toJson(postService.getPosts());
    }

    //org.springframework.web.bind.annotation
    //org.springframework.http
    @PostMapping(produces = MediaType.APPLICATION_JSON_VALUE)
    public ResponseEntity<String> addPost(@RequestBody String body) {
        log.info("Body received: {}", body);
        Post post = gson.fromJson(body, Post.class);
        log.info("Post deserialized: {}", post);
        Post created = postService.createPost(post);
        URI uri = ServletUriComponentsBuilder
                .fromCurrentRequest()
                .pathSegment("{id}").buildAndExpand(created.getId()).toUri();
        return ResponseEntity.created(uri).body(gson.toJson(created)); //response 201
    }
}
```

11. The DAO Pattern in Java

The Data Access Object (DAO) pattern is a structural pattern that allows us **to isolate the application/business layer from the persistence layer (usually a relational database but could be any other persistence mechanism) using an abstract API.**

Let's define a basic DAO layer so we can see how it can keep the domain model completely decoupled from the persistence layer.

```
public interface Dao<T> {

    Optional<T> get(long id);

    List<T> getAll();

    void save(T t);

    void update(T t, String[] params);

    void delete(T t);
}
```

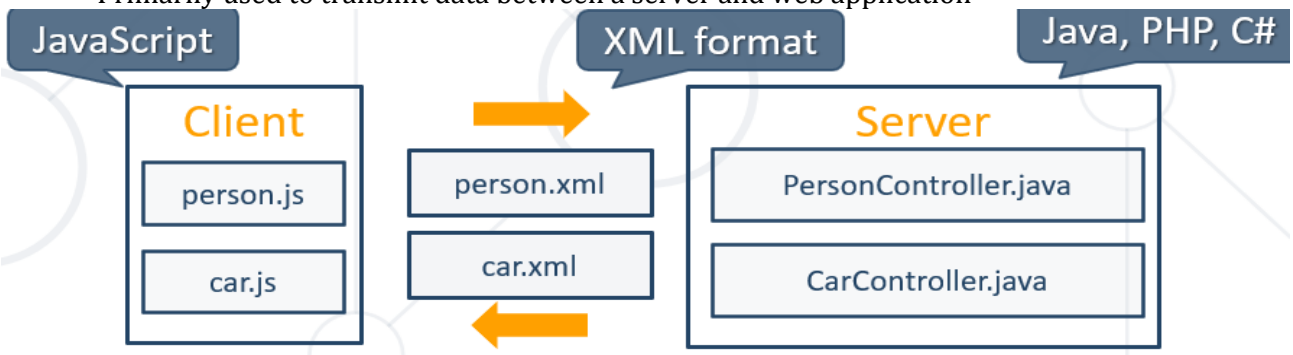
12. XML Processing

Exporting and Importing Data from XML Format

1. XML Processing

XML Specifics

- E**X**tensible **M**ark-up **L**anguage
  - Lightweight format that is used for **data interchanging**
  - XML is language independent
- Primarily used to transmit data between a server and web application



XML е базовия език, който слага едни базови правила, а HTML-а е подвид на XML-а с допълнени правила.
XML е по-рано залегнал(по-стар е) в сравнение с JSON.
Но JSON се е наложил в web страниците заради лекотата си.

XML Markup and Content

- An XML document consists of strings that:
  - Constitute **markup** – usually begin with **<** and end with **>**
  - Are **content** – placed between markup(**tags**)

XML Structure

- XML documents are formed as **element trees**
- An XML tree starts at a **root element** and branches from the root to **sub elements**
  - All elements can have child elements:

An XML element

is everything from (including) the element's start tag to (including) the element's end tag.

An element can contain:

- text
- attributes
- other elements
- or a mix of the above

```
<bookstore>
  <book category="children">
    <title>Harry Potter</title>
    <author>J K. Rowling</author>
    <year>2005</year>
```

```
    <price>29.99</price>
  </book>
  <book category="web">
    <title>Learning XML</title>
    <author>Erik T. Ray</author>
    <year>2003</year>
    <price>39.95</price>
  </book>
</bookstore>
```

XML elements must follow these naming rules:

- Element names are case-sensitive
- Element names must start with a letter or underscore
- Element names cannot start with the letters xml (or XML, or Xml, etc)
- Element names can contain letters, digits, hyphens, underscores, and periods
- Element names cannot contain spaces

Any name can be used, no words are reserved (except xml).

XML elements can be extended to carry more information.
Look at the following XML example:

```
<note>
  <to>Tove</to>
  <from>Jani</from>
  <body>Don't forget me this weekend!</body>
</note>
```

Imagine that the author of the XML document added some extra information to it:

```
<note>
  <date>2008-01-10</date>
  <to>Tove</to>
  <from>Jani</from>
  <heading>Reminder</heading>
  <body>Don't forget me this weekend!</body>
</note>
```

Should the application break or crash?

No. The application should still be able to find the <to>, <from>, and <body> elements in the XML document and produce the same output.

This is one of the beauties of XML. It can be extended without breaking applications.

https://www.w3schools.com/xml/xml_elements.asp

Security Assertion Markup Language is an open standard for exchanging authentication and authorization data between parties, in particular, between an identity provider and a service provider. SAML is an XML-based markup language for security assertions.

2. JAXB parser intro

JAXB

- Processes the schema of the XML **document into a set of Java classes** that represent it
- Generates compact and readable XML output

pom.xml

```
<dependency>
    <groupId>javax.xml.bind</groupId>
    <artifactId>jaxb-api</artifactId>
    <version>2.3.1</version>
</dependency>

<dependency>
    <groupId>javax.activation</groupId>
    <artifactId>activation</artifactId>
    <version>1.1</version>
</dependency>

<dependency>
    <groupId>org.glassfish.jaxb</groupId>
    <artifactId>jaxb-runtime</artifactId>
    <version>2.3.6</version>
</dependency>
```

JAXB Basics

- **Marshalling** - converting a Java Object to XML
- **Unmarshalling** - converting XML to Java Object
- We need to annotate the Java Object to provide instructions for XML reading or creation:

JAXB Annotations

- **@XmlRootElement**
  - Defines XML root object
- **@XmlAccessorType**
  - XmlAccessType.**FIELD**
  - XmlAccessType.**PROPERTY**
  - XmlAccessType.**PUBLIC_MEMBER**
- **@XmlAttribute**
  - Marks the field as an attribute to the object
- **@XmlElement**
  - Marks the field as an element
- **@XmlElementWrapper(name = "...")**
  - Wraps the array of objects – използва се за масив от обекти
- **@XmlTransient**
  - The field won't be exported/imported

XmlAccessType

```
public enum XmlAccessType {
    /**
     * Every getter/setter pair in a JAXB-bound class will be automatically
     * bound to XML, unless annotated by {@link XmlTransient}.
```

```
     *
     * Fields are bound to XML only when they are explicitly annotated
     * by some of the JAXB annotations.
     */
    PROPERTY,
    /**
     * Every non static, non transient field in a JAXB-bound class will be automatically
     * bound to XML, unless annotated by {@link XmlTransient}.
     *
     * Getter/setter pairs are bound to XML only when they are explicitly annotated
     * by some of the JAXB annotations.
     */
    FIELD,
    /**
     * Every public getter/setter pair and every public field will be
     * automatically bound to XML, unless annotated by {@link XmlTransient}.
     *
     * Fields or getter/setter pairs that are private, protected, or
     * defaulted to package-only access are bound to XML only when they are
     * explicitly annotated by the appropriate JAXB annotations.
     */
    PUBLIC_MEMBER,
    /**
     * None of the fields or properties is bound to XML unless they
     * are specifically  annotated with some of the JAXB annotations.
     */
    NONE
}
```

JAXB Initialization

- **JAXBContext** objects are responsible for the XML manipulations
- JAXBContext.newInstance(object.getClass()) - creates an **instance** of JAXBContext
- **object.getClass** is the class that we will export/import
    - E.g. User, Address, Employee…

```
this.jaxbContext = JAXBContext.newInstance(object.getClass());
```

Export Single Object to XML (Marshalling = Serialization)

```
import javax.xml.bind.annotation.XmlAccessType;
import javax.xml.bind.annotation.XmlAccessorType;
import javax.xml.bind.annotation.XmlElement;
import javax.xml.bind.annotation.XmlRootElement;
import java.math.BigDecimal;

@XmlRootElement(name = "category")
@XmlAccessorType(value = XmlAccessType.FIELD)
public class XMLCategoryStatsDTO {
    @XmlElement(name = "name")      //ако е зададено XmlAccessType.FIELD, то няма нужда от
анотация XmlElement. В случая искаме да променим името
    private String category;

    private long productsCount;

    private double averagePrice;
```

```
    private BigDecimal totalRevenue;

    public XMLCategoryStatsDTO() {   //задължително празен конструктор иска!
    }

    public XMLCategoryStatsDTO(CategoryStatsDTO other) {
        this.category = other.getCategory();
        this.productsCount = other.getProductsCount();
        this.averagePrice = other.getAveragePrice();
        this.totalRevenue = other.getTotalRevenue();
    }
}

public class CategoryStatsDTO { //DTO за вземане от базата данни
    private String category;
    private long productsCount;
    private double averagePrice;
    private BigDecimal totalRevenue;
}


        List<CategoryStatsDTO> result = productService.getCategoryStatistics()
//        String json = this.gson.toJson(result);
//        System.out.println(json);

        XMLCategoryStatsDTO first = new XMLCategoryStatsDTO(result.get(0));

        JAXBContext context = JAXBContext.newInstance(XMLCategoryStatsDTO.class);
        Marshaller marshaller = context.createMarshaller();
        marshaller.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, true); //format output
        marshaller.marshal(first, System.out);


резултатът
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<category>
    <name>Drugs</name>
    <productsCount>79</productsCount>
    <averagePrice>862.052911</averagePrice>
    <totalRevenue>68102.18</totalRevenue>
</category>


*   {
*     "category": "Drugs",
*     "productsCount": 79,
*     "averagePrice": 862.052911,
*     "totalRevenue": 68102.18
*   }
*
*   <categories-stats>
*       <category>
*           <name>Drugs</name>
*           <productsCount>79</productsCount>
*           <averagePrice>836</averagePrice>
*           <totalRevenue>56912.80</totalRevenue>
*       </category>
*   </categories-stats>
*/
```

Export Multiple Objects to XML (Marshalling = Serializartion)

Нужда от създаване на допълнителен клас DTO за колекцията

```
import javax.xml.bind.annotation.XmlAccessType;
import javax.xml.bind.annotation.XmlAccessorType;
import javax.xml.bind.annotation.XmlElementWrapper;
import javax.xml.bind.annotation.XmlRootElement;
import java.util.List;

@XmlRootElement(name = "categories-stats")
@XmlAccessorType(value = XmlAccessType.FIELD)
public class XMLCategoryStatsList {
    /**
     * Usage
     * The @XmlElementWrapper annotation can be used with the following program elements:
     * JavaBean property
     * non static, non transient field
     * The usage is subject to the following constraints:
     * The property must be a collection property
     * This annotation can be used with the following annotations: XmlElement, XmlElements,
XmlElementRef, XmlElementRefs, XmlJavaTypeAdapter.
     */
    @XmlElementWrapper(name = "categories")

    private List<XMLCategoryStatsDTO> stats;

    public XMLCategoryStatsList() {   //задължително празен конструктор иска!
    }

    public XMLCategoryStatsList(List<XMLCategoryStatsDTO> stats) {
        this.stats = stats;
    }
}


import javax.xml.bind.annotation.XmlAccessType;
import javax.xml.bind.annotation.XmlAccessorType;
import javax.xml.bind.annotation.XmlElement;
import javax.xml.bind.annotation.XmlRootElement;
import java.math.BigDecimal;

@XmlRootElement(name = "category")
@XmlAccessorType(value = XmlAccessType.FIELD)
public class XMLCategoryStatsDTO {
    @XmlElement(name = "name")      //ако е зададено XmlAccessType.FIELD, то няма нужда от
анотация XmlElement. В случая искаме да променим името
    private String category;

    private long productsCount;

    private double averagePrice;

    private BigDecimal totalRevenue;

    public XMLCategoryStatsDTO() {   //задължително празен конструктор иска!
    }

    public XMLCategoryStatsDTO(CategoryStatsDTO other) {
        this.category = other.getCategory();
        this.productsCount = other.getProductsCount();
```

```
        this.averagePrice = other.getAveragePrice();
        this.totalRevenue = other.getTotalRevenue();
    }
}

public class CategoryStatsDTO { //DTO за вземане от базата данни
    private String category;
    private long productsCount;
    private double averagePrice;
    private BigDecimal totalRevenue;
}

List<CategoryStatsDTO> result = this.productService.getCategoryStatistics();
//        String json = this.gson.toJson(result);
//        System.out.println(json);

        List<XMLCategoryStatsDTO> xmlResult = result
                .stream()
                .map(XMLCategoryStatsDTO::new)
                .collect(Collectors.toList());

        XMLCategoryStatsList xmlCategoryStatsList = new XMLCategoryStatsList(xmlResult);

        JAXBContext context = JAXBContext.newInstance(XMLCategoryStatsList.class);
        Marshaller marshaller = context.createMarshaller();
        marshaller.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, true);
        marshaller.marshal(xmlCategoryStatsList, System.out);
```

Export Nested Multiple Objects to XML (Marshalling = Serializartion)

```
@XmlRootElement(name = "shops")
@XmlAccessorType(XmlAccessType.FIELD)
public class ShopsRootExportToXMLDto {
    @XmlElement(name = "shop")
    private List<ShopOneExportToXMLDto> shops;

    public ShopsRootExportToXMLDto() {
    }

    public List<ShopOneExportToXMLDto> getShops() {
        return shops;
    }

    public ShopsRootExportToXMLDto(List<ShopOneExportToXMLDto> shops) {
        this.shops = shops;
    }
}

@XmlRootElement(name = "shop") //за извеждането на xml файл задължително ни трябва тази анотация
в случая
@XmlAccessorType(XmlAccessType.FIELD)
public class ShopOneExportToXMLDto {
    @XmlElement
    private String name;

    @XmlElement
    private String address;

    @XmlElement(name = "employee-count")
    private int employeeCount;
```

```java
    @XmlElement(name = "shop-area")
    private int shopArea;

    @XmlElement(name = "town")
    private TownToXmlDto town;

    public ShopOneExportToXMLDto() {
    }
}

@XmlRootElement(name = "town")
@XmlAccessorType(XmlAccessType.FIELD)
public class TownToXmlDto {
    @XmlElement
    private String name;

    public TownToXmlDto() {
    }
}

public class Main {
    public static void main(String[] args) throws IOException, JAXBException {
        ShopOneExportToXMLDto entry1 = new ShopOneExportToXMLDto();
        entry1.setName("Bai Kirovata krychma");
        entry1.setAddress("u nas si");
        entry1.setEmployeeCount(8);
        entry1.setShopArea(200);
        TownToXmlDto tn1 = new TownToXmlDto();
        tn1.setName("Sofia");
        entry1.setTown(tn1);

        ShopOneExportToXMLDto entry2 = new ShopOneExportToXMLDto();
        entry2.setName("Emi\'s shop");
        entry2.setAddress("Prava polqna 18");
        entry2.setEmployeeCount(14);
        entry2.setShopArea(150);
        TownToXmlDto tn2 = new TownToXmlDto();
        tn2.setName("Plovdiv");
        entry2.setTown(tn2);

        List<ShopOneExportToXMLDto> outputList = new ArrayList<>();
        outputList.add(entry1);
        outputList.add(entry2);

        ShopsRootExportToXMLDto allShops = new ShopsRootExportToXMLDto(outputList);

        JAXBContext context = JAXBContext.newInstance(ShopsRootExportToXMLDto.class);
        Marshaller marshaller = context.createMarshaller();
        marshaller.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, true);

        Path path = Path.of("src/main/resources/output/outputToXML.xml");
        FileWriter fileWriter = new FileWriter(path.toString());

        marshaller.marshal(allShops, fileWriter);

        fileWriter.close();
    }
}
```

Изхода е:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<shops>
    <shop>
        <name>Bai Kirovata krychma</name>
        <address>u nas si</address>
        <employee-count>8</employee-count>
        <shop-area>200</shop-area>
        <town>
            <name>Sofia</name>
        </town>
    </shop>
    <shop>
        <name>Emi's shop</name>
        <address>Prava polqna 18</address>
        <employee-count>14</employee-count>
        <shop-area>150</shop-area>
        <town>
            <name>Plovdiv</name>
        </town>
    </shop>
</shops>
```

Import Single Object from XML (Unmarshalling = Deserializartion)

```
    private void xmlDemo() throws JAXBException {
Проверяваме дали правилно сме изписали файла xml
        String xml = "<?xml version=\"1.0\" encoding=\"UTF-8\"?>\n" +
                "      <category>\n" +
                "         <name>Drugs</name>\n" +
                "         <productsCount>79</productsCount>\n" +
                "         <averagePrice>836</averagePrice>\n" +
                "         <totalRevenue>56912.80</totalRevenue>\n" +
                "       </category>";

        JAXBContext context = JAXBContext.newInstance(XMLCategoryStatsDTO.class);
        Unmarshaller unmarshaller = context.createUnmarshaller();

//        InputStream inputStream = getClass().getResourceAsStream("src/main…./files/input/xml/
address.xml");
//        BufferedReader bfr = new BufferedReader(new InputStreamReader(inputStream));

        ByteArrayInputStream inputStream = new ByteArrayInputStream(xml.getBytes());
        XMLCategoryStatsDTO result = (XMLCategoryStatsDTO) unmarshaller.unmarshal(inputStream);

        System.out.println(result);
    }
```

```
@XmlRootElement(name = "category")
@XmlAccessorType(value = XmlAccessType.FIELD)
public class XMLCategoryStatsDTO {
    @XmlElement(name = "name")
    private String category;

    private long productsCount;
```

```
    private double averagePrice;

    private BigDecimal totalRevenue;

    public XMLCategoryStatsDTO() {
    }

    public XMLCategoryStatsDTO(CategoryStatsDTO other) {
        this.category = other.getCategory();
        this.productsCount = other.getProductsCount();
        this.averagePrice = other.getAveragePrice();
        this.totalRevenue = other.getTotalRevenue();
    }

    @Override
    public String toString() {
        return "XMLCategoryStatsDTO{" +
                "category='" + category + '\'' +
                ", productsCount=" + productsCount +
                ", averagePrice=" + averagePrice +
                ", totalRevenue=" + totalRevenue +
                '}';
    }
}
```

Изхода:
XMLCategoryStatsDTO{category='Drugs', productsCount=79, averagePrice=836.0, totalRevenue=56912.80}

Import Multiple Objects from XML (Unmarshalling = Deserializartion)

Пример 0:

```
JAXBContext jaxbContext = JAXBContext.newInstance(AddressesDto.class);
InputStream inputStream =
getClass().getResourceAsStream("src/main/files/input/xml/addresses.xml");


BufferedReader bfr = new BufferedReader(new InputStreamReader(inputStream));
или
private static final Path USERS_XML_PATH =
        Path.of("src", "main", "resources", "productsshopInit", "users.xml");
BufferedReader xmlReader = Files.newBufferedReader(USERS_XML_PATH);
или



Unmarshaller unmarshaller = jaxbContext.createUnmarshaller();
AddressesDto addressesDto = (AddressesDto) unmarshaller.unmarshal(bfr);



@Component
public class XmlParserImpl implements XmlParser {

    @Override
    @SuppressWarnings("unchecked")
    public <T> T fromFile(String filePath, Class<T> tClass) throws JAXBException,
```

```
FileNotFoundException {
        JAXBContext jaxbContext = JAXBContext.newInstance(tClass);
        Unmarshaller unmarshaller = jaxbContext.createUnmarshaller();

        return (T) unmarshaller.unmarshal(new FileReader(filePath));

 (ImportEmployeesRootDTO) unmarshaller.unmarshal(new File(xmlPath.toString()));

    }
}
```

Пример 1 - //вариант без ModelMapper, а чрез извикване на конструктор:

```
<?xml version='1.0' encoding='UTF-8'?>
<categories>
   <category>
      <name>Drugs</name>
   </category>
   <category>
      <name>Adult</name>
   </category>
   <category>
      <name>Other</name>
   </category>
</categories>
```

```
import static javax.xml.bind.annotation.XmlAccessType.*;
@XmlRootElement(name = "categories")  //един главен root елемент categories
@XmlAccessorType(FIELD)
public class CategoriesImportDTO {

    @XmlElement(name = "category")     //поделемент category
    private List<CategoryNameDTO> categories;

    public CategoriesImportDTO() {
    }

    public List<CategoryNameDTO> getCategories() {
        return categories;
    }
}
```

```
import javax.xml.bind.annotation.*;

import static javax.xml.bind.annotation.XmlAccessType.FIELD;
@XmlRootElement(name = "category")  //един главен root елемент category
@XmlAccessorType(FIELD)
public class CategoryNameDTO {
    @XmlElement                    //поделемент name
    private String name;

    public CategoryNameDTO() {
    }

    public String getName() {
        return name;
    }
}
```

```
@Override
public void seedCategories() throws FileNotFoundException, JAXBException {
    JAXBContext context = JAXBContext.newInstance(CategoriesImportDTO.class);
    Unmarshaller unmarshaller = context.createUnmarshaller();

    FileReader xmlReader = new FileReader(CATEGORIES_XML_PATH.toAbsolutePath().toString());
    CategoriesImportDTO importDTO = (CategoriesImportDTO) unmarshaller.unmarshal(xmlReader);

//вариант без ModelMapper, а чрез извикване на конструктор
    List<Category> entities = importDTO
            .getCategroies()
            .stream()
            .map(catName -> new Category(catName.getName()))
            .collect(Collectors.toList());

    this.categoryRepository.saveAll(entities);
}
```

Пример 2 - //вариант с modelMapper:

```
<?xml version='1.0' encoding='UTF-8'?>
<users>
   <user first-name="Eugene" last-name="Stewart" age="65"/>
   <user first-name="Fred" last-name="Allen" age="57"/>
   <user first-name="Clarence" last-name="Fowler" age="50"/>
</users>
```

```
import javax.xml.bind.annotation.*;


@XmlRootElement(name = "users")           //един главен root елемент всички users
@XmlAccessorType(XmlAccessType.FIELD)
public class UsersImportDTO {

    @XmlElement(name = "user")         //поделемент колекция от един user
    private List<OneUserImportDTO> users;

    public UsersImportDTO() {
    }

    public List<OneUserImportDTO> getUsers() {
        return users;
    }
}
```

```
import javax.xml.bind.annotation.*;

@XmlRootElement(name = "user")        //всеки единичен user
@XmlAccessorType(XmlAccessType.FIELD)
public class OneUserImportDTO {
    @XmlAttribute(name = "first-name")    //името на атрибута на тага
    private String firstName;

    @XmlAttribute(name = "last-name")        //името на атрибута на тага
    private String lastName;
```

```
    @XmlAttribute            //името на атрибута на тага е същия като името на полето на класа
    private int age;

    public OneUserImportDTO() {
    }
}
```

```
@Override
public void seedUsers() throws IOException, JAXBException {
    JAXBContext context = JAXBContext.newInstance(UsersImportDTO.class);
    Unmarshaller unmarshaller = context.createUnmarshaller();

    BufferedReader xmlReader = Files.newBufferedReader(USERS_XML_PATH);
    UsersImportDTO usersDTO = (UsersImportDTO) unmarshaller.unmarshal(xmlReader);

//вариант с modelMapper
    List<User> entities = usersDTO.getUsers()
            .stream()
            .map(dto -> this.modelMapper.map(dto, User.class))
            .collect(Collectors.toList());

    this.userRepository.saveAll(entities);
}
```

Import Nested Multiple Objects from XML (Unmarshalling = Deserializartion)

```
<?xml version='1.0' encoding='UTF-8'?>
<shops>
    <shop>
        <address>04 Pond Junction</address>
        <employee-count>47</employee-count>
        <income>625273</income>
        <name>Er</name>
        <shop-area>278</shop-area>
        <town>
            <name>Hamburg</name>
        </town>
    </shop>
    <shop>
        <address>779 Parkside Park</address>
        <employee-count>48</employee-count>
        <income>10</income>
        <name>Raynor</name>
        <shop-area>108</shop-area>
        <town>
            <name>Birmingham</name>
        </town>
    </shop>
</shops>
```

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<projects>
    <project>
        <name>HealthCare</name>
        <description>Pharmacy web app</description>
        <start-date>2017-07-12</start-date>
        <is-finished>true</is-finished>
        <payment>10000.00</payment>
```

```
        <company name="Phoenix"/>
    </project>
    <project>
        <name>MMA</name>
        <description>Mixed martial arts club web site</description>
        <start-date>2018-09-21</start-date>
        <is-finished>true</is-finished>
        <payment>7000.00</payment>
        <company name="Vallor"/>    //тук пак използваме вложено DTO, атрибута не е проста
единична стойност
    </project>
</projects>
```

```
@XmlRootElement(name = "shops")
@XmlAccessorType(XmlAccessType.FIELD)
public class ShopsRootImportFromXMLDto {
    @XmlElement(name = "shop")
    private List<ShopOneImportFromXMLDto> shops;

    public ShopsRootImportFromXMLDto() {
    }

    public List<ShopOneImportFromXMLDto> getShops() {
        return shops;
    }
}
```

```
@XmlRootElement(name = "shop")
@XmlAccessorType(XmlAccessType.FIELD)
public class ShopOneImportFromXMLDto {
    @XmlElement
    @Size(min = 4)
    private String address;

    @XmlElement(name = "employee-count")
    @Min(value = 1)
    @Max(50)
    private int employeeCount;

    @XmlElement
    @Min(value = 20000)
    private BigDecimal income;

    @XmlElement
    @Size(min = 4)
    private String name;

    @XmlElement(name = "shop-area")
    @Min(150)
    private int shopArea;

    @XmlElement(name = "town")
    private TownDto town;

    public ShopOneImportFromXMLDto() {
    }
}
```

```java
@XmlRootElement(name = "town")
@XmlAccessorType(XmlAccessType.FIELD)
public class TownDto {

    @XmlElement
    private String name;

    public TownDto() {
    }
}

@Service
public class ShopServiceImpl implements ShopService {
    private final String SHOPS_PATHSTRING_XML = "src/main/resources/files/xml/shops.xml";

    private final ShopRepository shopRepository;
    private final TownService townService;
    private final ModelMapper modelMapper;
    private final ValidationUtil validationUtil;
    private final XmlParser xmlParser;

    @Autowired
    public ShopServiceImpl(ShopRepository shopRepository, TownService townService, ModelMapper
modelMapper,
                           ValidationUtil validationUtil, XmlParser xmlParser) {
        this.shopRepository = shopRepository;
        this.townService = townService;
        this.modelMapper = modelMapper;
        this.validationUtil = validationUtil;
        this.xmlParser = xmlParser;
    }

    @Override
    public boolean areImported() {
        return this.shopRepository.count() > 0;
    }

    @Override
    public String readShopsFileContent() throws IOException {
        return Files.readString(Path.of(SHOPS_PATHSTRING_XML));
    }

    @Override
    public String importShops() throws JAXBException, FileNotFoundException {
        ShopsRootImportFromXMLDto shops = xmlParser.fromFile(SHOPS_PATHSTRING_XML,
                ShopsRootImportFromXMLDto.class);

        return shops.getShops()
                .stream()
                .map(this::importShop)
                .collect(Collectors.joining("\n"));
    }

    private String importShop(ShopOneImportFromXMLDto dto) {
        boolean isValid = this.validationUtil.isValid(dto);
        if (!isValid) {
            return "Invalid shop";
        }

        Optional<Shop> optShop = this.shopRepository.findByName(dto.getName());
```

```
        if (optShop.isPresent()) {
            return "Invalid shop";
        }

        Optional<Town> optTown = this.townService.findTownByName(dto.getTown().getName());

        Shop shop = this.modelMapper.map(dto, Shop.class);
        shop.setTown(optTown.get());
        this.shopRepository.save(shop);

        return "Successfully imported Shop " + shop.getName() + " - " + shop.getIncome();
    }
}


@Configuration
//Indicates that a class declares one or more @Bean methods and may be processed by the Spring
container to generate bean definitions and service requests for those beans at runtime, for
example:
public class ApplicationBeanConfiguration {
    @Bean   //Indicates that a method produces a bean to be managed by the Spring container
    public Gson gson() {
        return new GsonBuilder()
//                .excludeFieldsWithoutExposeAnnotation()
                .setPrettyPrinting()
                .create();
    }

    @Bean //Indicates that a method produces a bean to be managed by the Spring container
    public ModelMapper modelMapper() {
        ModelMapper modelMapper = new ModelMapper();

        modelMapper.addConverter(new Converter<String, LocalDate>() {
            @Override
            public LocalDate convert(MappingContext<String, LocalDate> mappingContext) {
                return LocalDate.parse(mappingContext.getSource(),
DateTimeFormatter.ofPattern("dd/MM/yyyy"));
            }
        });

        modelMapper.addConverter(new Converter<String, LocalDateTime>() {
            @Override
            public LocalDateTime convert(MappingContext<String, LocalDateTime> mappingContext) {
                return LocalDateTime.parse(mappingContext.getSource(),
DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss"));
            }
        });

modelMapper.addConverter(new Converter<String, LocalTime>() {
    @Override
    public LocalTime convert(MappingContext<String, LocalTime> mappingContext) {
        return LocalTime.parse(mappingContext.getSource(),
DateTimeFormatter.ofPattern("HH:mm:ss"));
    }
});


        return modelMapper;
    }
}
```

```java
@Component
public class XmlParserImpl implements XmlParser {
    @Override
    @SuppressWarnings("unchecked")
    public <T> T fromFile(String filePath, Class<T> tClass) throws JAXBException,
FileNotFoundException {
        JAXBContext jaxbContext = JAXBContext.newInstance(tClass);
        Unmarshaller unmarshaller = jaxbContext.createUnmarshaller();

        return (T) unmarshaller.unmarshal(new FileReader(filePath));
    }
}
```

XmlJavaTypeAdapter example

```java
public class CarSeedFromJSONDto {
    @Expose
    private String make;

    @Expose
    private String model;

    @Expose
    private Integer kilometers;

//    @Expose
//    private String registeredOn;

    @Expose
    @XmlJavaTypeAdapter(LocalDateAdapter.class)
    private LocalDate registeredOn;
}
```

```java
package softuni.exam.adapters;

import javax.xml.bind.annotation.adapters.XmlAdapter;
import java.time.LocalDate;
import java.time.format.DateTimeFormatter;

public class LocalDateAdapter extends XmlAdapter<String, LocalDate> {

    @Override
    public LocalDate unmarshal(String src) throws Exception {
        return LocalDate.parse(src, DateTimeFormatter.ofPattern("dd/MM/yyyy"));
    }

    @Override
    public String marshal(LocalDate date) throws Exception {
        return date.toString();
    }
}
```

JPABuddy in minimalistic mode:

Alt + Enter – Show JPA inspector – слага анотации на полета и на класове, edit relations, edit columns
Alt + Ins – generate все едно – JPA Pallete – getters, setters, new Class Entity, New Relation for a new field и т.н.

https://www.jpa-buddy.com/documentation/minimalistic-mode/

HashCode and equals

ВАЖНО: В света на Java и без използване на други framework-ци, имаме ли Set, то трябва задължително да си имплементираме всеки път equals() и hashCode() методите – от Alt + Insert за по-лесно! Иначе, примерно ако създаваме random обекти, два различни обекта може да се окаже че има еднакъв hashCode.
Обаче ако използваме Hibernate, то ние четем вече записан обект от базата, който пристига като managed entity, също така не даваме new в java-та. И т.е. нямаме нужда от hashCode и equals. Единственото дублиране, което изобщо е възможно е ако имаме detached entity, което се опитваме да импортираме/запишем в базата наново.

```
import javax.persistence.*;
import java.util.Objects;

@Entity(name = "categories")
public class Category {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;

    @Column(nullable = false, length = 15)
    private String name;

    public Category() {
    }

    public Category(String name) {
        this.name = name;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Category category = (Category) o;
        return id == category.id;  //сравняваме по уникалност само ключово поле id, защото само
то е уникално в SQL базата
        //дори да променим името на категорията, то тя все още е една и съща категория за базата
данни
    }

    @Override
    public int hashCode() {
        return Objects.hash(id);
    }
}


private Product sendRandomcategories(Product product) {
    Random random = new Random();
    long categoriesDbCount = this.categoryRepository.count();

    int count = random.nextInt((int) categoriesDbCount) + 1;

    Set<Category> categories = new HashSet<>();
    for (int i = 0; i < count; i++) {
        int randomId = random.nextInt((int) categoriesDbCount) + 1;

        Optional<Category> randomCategory = this.categoryRepository.findById(randomId);
```

```
        categories.add(randomCategory.get()); //при добавянето се бърка защото за Java си има различен hashCode, но за базата hashCode трябва да го построим само по Id, а не по id и name.
.get() връща от Optional<Category>  само Category
    }

    product.setCategories(categories);
    return product;
}


List<Product> products = Arrays.stream(productsImportDTOS)
        .map(importDTO -> this.modelMapper.map(importDTO, Product.class))
        .map(this::setRandomSeller)
        .map(this::setRandomBuyer)
        .map(this::sendRandomcategories)
        .collect(Collectors.toList());
this.productRepository.saveAll(products);
```

Annotation Type Embeddable and Embedded

Пример 1

```
@Documented
 @Target(value=TYPE)
 @Retention(value=RUNTIME)
public @interface Embeddable
```

Specifies a class whose instances are stored as an intrinsic part of an owning entity and share the identity of the entity. Each of the persistent properties or fields of the embedded object is mapped to the database table for the entity.

Note that the `Transient` annotation may be used to designate the non-persistent state of an embeddable class.

```
Example 1:

@Embeddable public class EmploymentPeriod {
   @Temporal(DATE) java.util.Date startDate;
   @Temporal(DATE) java.util.Date endDate;
  ...
}

Example 2:

@Embeddable public class PhoneNumber {
    protected String areaCode;
    protected String localNumber;
    @ManyToOne PhoneServiceProvider provider;
    ...
 }

@Entity public class PhoneServiceProvider {
    @Id protected String name;
     ...
 }

Example 3:
```

```
@Embeddable public class Address {
   protected String street;
   protected String city;
   protected String state;
   @Embedded protected Zipcode zipcode;
}

@Embeddable public class Zipcode {
   protected String zip;
   protected String plusFour;
 }
```

Since:

Java Persistence 1.0

Пример 2

Take your time and inspect the new entity com.vidasoft.magman.model.CreditCard. It is declared to be @Embeddable. And the Subscriber entity has a field of this type, which in turns is declared to be @Embedded. What this all means is that the JPA provider will not create a separate table for the CreditCard entity. It will rather add its fields as columns to the entity, where it is embedded

```
@Embeddable
public class CreditCard implements Serializable {
   private String number;

   @Enumerated(EnumType.STRING)
   private CreditCardType creditCardType;
```

```
@Entity
public class Subscriber extends User {
   private String streetAddress;

   @Future
   private LocalDate subscribedUntil;

   @Embedded
   private CreditCard creditCard;
```

@Size, @Length, @Column(length = 50)

https://www.baeldung.com/jpa-size-length-column-differences

Simply put, all of these annotations are meant to **communicate the size of a field.**

@Size and *@Length* are similar. We can use either **to validate the size of a JAVA class field**. The first is a Java-standard annotation (javax.validation)

and the second is specific to Hibernate.

@Column, though, is a JPA annotation that we use to control DDL statements.

We'll use *@Column* to **indicate specific characteristics of the physical database column.** Note that we'll use *@Column* only to specify table column properties **as it doesn't provide validations.**

```
@Entity
public class User {

    @Column(length = 3)
    private String firstName;

    // ...

}
```

Consequently, the resulting column would be generated as a *VARCHAR(3)* and trying to insert a longer string would result in an SQL error.

Note that we'll use *@Column* only to specify table column properties **as it doesn't provide validations.**

Of course, **we can use *@Column* together with *@Size*** to specify database column property with bean validation.

```
@Entity
public class User {

    // ...

    @Column(length = 5)   //custom-извира SQL базата
    @Size(min = 3, max = 5)    import javax.validation.constraints.Size;   via Maven library
@Length(min = 3, max = 15)       import org.hibernate.validator.constraints.Length; - in pom.xml
file added
    private String city;

    // ...

}
```

```
<!-- https://mvnrepository.com/artifact/org.hibernate/hibernate-validator -->
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-validator</artifactId>
    <version>6.2.3.Final</version>
</dependency>
```

```java
@NonNull   -  това е от Lombok, който не било хубаво да го ползвам
@Length(min = 3, max = 60)   //@Length на Java класа полето не включва notNull
@Column(unique = true, nullable = false) //за базата ограничение при създаването
@NotNull          //jakarta validation   javax.validation.constraints на Java класа полето да не
е null
private String username;
```

Javax.validation - http://hibernate.org/validator/

```java
import javax.validation.Validation;
import javax.validation.Validator;
```

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

<dependency>
  <groupId>org.hibernate.validator</groupId>
  <artifactId>hibernate-validator</artifactId>
  <version>6.2.3.Final</version>
  <scope>compile</scope>
</dependency>
```

```java
@Service
public class TownServiceImpl implements TownService {
    private final TownRepository townRepository;
    private final Gson gson;
     private final Validator validator;
    private final ModelMapper modelMapper;

    @Autowired
    public TownServiceImpl(TownRepository townRepository) {
        this.townRepository = townRepository;
        this.gson = new GsonBuilder().create();

          this.validator = Validation
                  .buildDefaultValidatorFactory()
                  .getValidator();

        this.modelMapper = new ModelMapper();
    }

    @Override
    public boolean areImported() {
        return this.townRepository.count() > 0;
    }

    @Override
    public String readTownsFileContent() throws IOException {
        Path path = Path.of("src", "main", "resources", "files", "json", "towns.json");
```

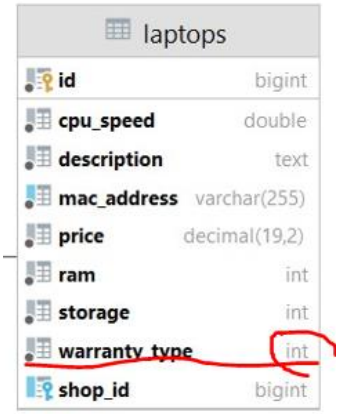
```
        return String.join("\n", Files.readAllLines(path));
    }

@Override
public String readCarsFileContent() throws IOException {
    return Files.readString(Path.of(CARS_FILE_PATH));
}


    @Override
    public String importTowns() throws IOException {
        String json = this.readTownsFileContent();
        ImportTownDTO[] importTownDTOS = this.gson.fromJson(json, ImportTownDTO[].class);

        List<String> result = new ArrayList<>();
        for (ImportTownDTO iTDTO : importTownDTOS) {
            //Ако използваме javax.validation анотации в ImportTownDTO, то използваме следното за
да видим дали има грешка
             Set<ConstraintViolation<ImportTownDTO>> validationErrors
= this.validator.validate(iTDTO);

Set<ConstraintViolation<T>> errors = validator.validate(configuration);

   if (!errors.isEmpty()) {

     logger.error("YAML configuration failed validation");

     for (ConstraintViolation<?> error : errors) {

       logger.error(error.getPropertyPath() + ": " + error.getMessage());
     }

          if (validationErrors.isEmpty()) {
              //valid
              Optional<Town> optTown = this.townRepository.findByName(iTDTO.getName());

              if (optTown.isEmpty()) {
                  Town town = this.modelMapper.map(iTDTO, Town.class);
                  this.townRepository.save(town);
                  result.add(String.format("Successfully imported Town %s - %d",
                          town.getName(), town.getPopulation()));
              } else {
                  result.add("Invalid town");
              }
          } else {
              result.add("Invalid town");
          }
        }

        return String.join("\n", result);
    }
}



import javax.validation.constraints.Min;
import javax.validation.constraints.Positive;
```

```java
import javax.validation.constraints.Size;

public class ImportTownDTO {
    @Size(min = 2)  //дължина на стринга
    private String name;

    @Positive
    @Min(value = 10000)
    private int population;

    @Size(min = 10, max = 19)
    private String travelGuide;

    public ImportTownDTO() {
    }

    public String getName() {
        return name;
    }

    public int getPopulation() {
        return population;
    }

    public String getTravelGuide() {
        return travelGuide;
    }
}
```

```
@NotNull          //jakarta validation   javax.validation.constraints на Java класа полето да не
е null, Accepts any type, including Enum
@NotBlank - //jakarta validation   javax.validation.constraints The annotated element must not be
null and must contain at least one non-whitespace character. Accepts any type, including Enum
```


```
[
  {
    "macAddress": "B5-42-0A-AC-F0-19",
    "cpuSpeed": 1.46,
    "ram": 128,
    "storage": 128,
    "description": "Aenean lectus. Pellentesque eget nunc.",
    "price": 4081.54,
    "warrantyType": "PREMIUM",
    "shop": {
      "name": "Jacobi and Bayer"
    }
  },
  {
    "macAddress": "B5-42-0A-AC-F0-19",
    "cpuSpeed": 1.46,
    "ram": 128,
    "storage": 128,
    "description": "Aenean lectus. Pellentesque eget nunc.",
    "price": 4081.54,
    "warrantyType": "PREMIUM",
    "shop": {
      "name": "Jacobi and Bayer"
    }
  }
]
```

```java
public enum WarrantyType {    BASIC, PREMIUM, LIFETIME   }

@Entity
@Table(name = "laptops")
public class Laptop {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private long id;

    @Column(name = "mac_address", nullable = false, unique = true)
    private String macAddress;

    @Column(name = "cpu_speed", nullable = false)
    private double cpuSpeed;

    @Column
    private int ram;

    @Column
    private int storage;

    @Column(columnDefinition = "TEXT")
    private String description;

    @Column(nullable = false)
    private BigDecimal price;

    @Column(name = "warranty_type")
    @Enumerated(EnumType.ORDINAL)  // в базата записваме или ORDINAL или STRING според задачата
    private WarrantyType warrantyType;
```




```java
}
```

Проверките на javax.validation го правим в Dto-класа

```java
public class LaptopSeedFromJSONDto {
    @Size(min = 9)
    private String macAddress;

    @Positive
    private double cpuSpeed;

    @Min(8)
    @Max(128)
    private int ram;
```

```
    @Min(128)
    @Max(1024)
        @Range(min = 1, max = 120)
    private int storage;

    @Size(min = 10)
    private String description;

    @Positive(message = "Price must be positive")
@DecimalMin(value = "0", message = "Price must be positive")
    private BigDecimal price;

    @NotNull  //ако .ORDINAL или .STRING не е това което четем, то хвърля null, и ние тук
проверяваме ако е null, да я хванем грешката и да не записваме в базата
    private WarrantyType warrantyType;  //String value, not ordinal!!!



@XmlElement
@Email //проверява за е-мейл
@Pattern() //проверява по Regex
@Pattern(regexp = "", message = "INVALID_EMAIL_ADDRESS_MESSAGE")
private String email;
```

Custom Validation - Spring MVC

Вариант 1 – с полета на анотацията

```
@Retention(value = RetentionPolicy.RUNTIME)
@Target(ElementType.FIELD)
@Constraint(validatedBy = {PasswordValidator.class})
public @interface Password {
    String message() default "Default message for @Password annotation";

    Class<?>[] groups() default {};

    Class<? extends Payload>[] payload() default {};

    int minLength() default 8;

    int maxLength() default 150;

    boolean shouldContainDigit() default false;

    boolean shouldContainLowercase() default false;

    boolean shouldContainUppercase() default false;

    boolean shouldContainSpecialSymbol() default false;
}


public class PasswordValidator implements ConstraintValidator<Password, String> {
    private Password password;
    private ConstraintValidatorContext context;
    private String value;
    private int violationsCount = 0;
```

```
    private static final List<Character> specialSymbols = Arrays.asList('!', '@', '#', '$', '%',
'^', '&', '*', '(', ')', '_', '+', '<', '>', '?');

    private static final String PASSWORD_TOO_SHORT = "Password should be at least %d characters";
    private static final String PASSWORD_MIN_MAX_LENGTH_MISMATCH = "Password min length cannot be
greater than max length";
    private static final String PASSWORD_TOO_LONG = "Password should be no longer than %d
characters";
    private static final String PASSWORD_NO_DIGIT = "Password should contain at least 1 digit";
    private static final String PASSWORD_NO_LOWERCASE = "Password should contain at least 1
lowercase letter";
    private static final String PASSWORD_NO_UPPERCASE = "Password should contain at least 1
uppercase letter";
    private static final String PASSWORD_NO_SPECIAL_SYMBOL = "Password should contain at least 1
special symbol (" +
            specialSymbols.stream().map(e -> e + "").collect(Collectors.joining(", ")) + ")";


    @Override
    public void initialize(Password constraintAnnotation) {
        ConstraintValidator.super.initialize(constraintAnnotation);
        this.password = constraintAnnotation;
    }

    @Override
    public boolean isValid(String value, ConstraintValidatorContext context) {
        this.value = value;
        this.context = context;
        context.disableDefaultConstraintViolation();

        if (this.password.minLength() > this.password.maxLength()) {
            setMessage(PASSWORD_MIN_MAX_LENGTH_MISMATCH);
            violationsCount++;
        }

        if (this.value.length() < this.password.minLength()) {
            setMessage(String.format(PASSWORD_TOO_SHORT, this.password.minLength()));
            violationsCount++;
        }

        if (this.value.length() > this.password.maxLength()) {
            setMessage(String.format(PASSWORD_TOO_LONG, this.password.maxLength()));
            violationsCount++;
        }

        if (!doesContainDigit()) {
            violationsCount++;
            setMessage(PASSWORD_NO_DIGIT);
        }

        if (!doesContainLowercase()) {
            violationsCount++;
            setMessage(PASSWORD_NO_LOWERCASE);
        }

        if (!doesContainUppercase()) {
            violationsCount++;
            setMessage(PASSWORD_NO_UPPERCASE);
        }

        if (!doesContainSpecialSymbol()) {
```

```
            violationsCount++;
            setMessage(PASSWORD_NO_SPECIAL_SYMBOL);
        }

        if (violationsCount == 0) {
    return true;
} else {
    violationsCount = 0;
    return false;
}

    }

    private void setMessage(String message) {
        this.context.buildConstraintViolationWithTemplate(message).addConstraintViolation();
    }
}


@Entity
@Table(name = "users")
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;

    @Column(nullable = false, unique = true)
    @Size(min = 4, max = 30)
    private String username;

    @Password(minLength = 16,
            shouldContainSpecialSymbol = true,
            shouldContainLowercase = true,
            shouldContainUppercase = true,
            shouldContainDigit = true)
    @Column(nullable = false)
    private String password;

    @Email
    @Column(nullable = false)
    private String email;

    @Column(name = "registered_on", nullable = false)
    private LocalDateTime registeredOn;

    @Column(name = "last_login_time", nullable = false)
    private LocalDateTime lastLoginTime;

    @Range(min = 1, max = 120)
    @Column(nullable = false)
    private int age;

    private boolean deleted = false;

    @Column(name = "first_name")
    private String firstName;

    @Column(name = "last_name")
    private String lastName;

    public User() {
```

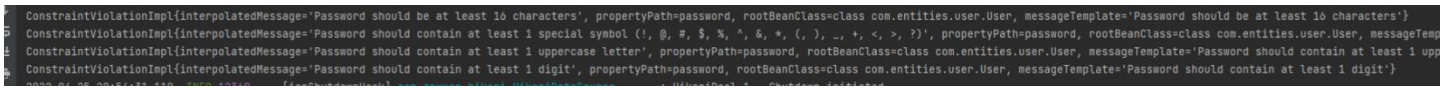
```
    }

    public User(String username, String password, String email, int age) {
        this();
        this.username = username;
        this.password = password;
        this.email = email;
        this.age = age;
        this.registeredOn = LocalDateTime.now();
        this.lastLoginTime = this.registeredOn;
    }
}

@Component
public class Application implements CommandLineRunner {
    private final UserRepositoty userRepositoty;


@Override
public void run(String... args) throws Exception {
List<User> users = new ArrayList<>() {{
    add(new User("IvanBegachkata", "password", "ivan@abv.bg", 15));
    add(new User("dragan", "password", "dragan@abv.bg", 20));
    add(new User("perkan", "password", "perkan@abv.bg", 6));
}};

try {
userRepositoty.saveAll(users);
}
catch (ConstraintViolationException e) {
    e.getConstraintViolations().forEach(System.out::println);
}
}


@SpringBootApplication
public class UserSystem {
    public static void main(String[] args) {
        SpringApplication.run(UserSystem.class, args);
    }
}
```

Вариант 2 – само с анотация @Password без полета на анотацията

```
@Password
@Column(nullable = false)
private String password;


@Retention(value = RetentionPolicy.RUNTIME)
@Target(ElementType.FIELD)
@Constraint(validatedBy = {PasswordValidator.class})
public @interface Password {
    String message() default "Default message for @Password annotation";
    Class<?>[] groups() default {};
    Class<? extends Payload>[] payload() default {};
}
```

```
public class PasswordValidator implements ConstraintValidator<Password, String> {
    private Password password;
    private ConstraintValidatorContext context;
    private String value;
    private int violationsCount = 0;

    private static final List<Character> specialSymbols = Arrays.asList('!', '@', '#', '$', '%',
'^', '&', '*', '(', ')', '_', '+', '<', '>', '?');

    private static final String PASSWORD_TOO_SHORT = "Password should be at least %d characters";
    private static final String PASSWORD_MIN_MAX_LENGTH_MISMATCH = "Password min length cannot be
greater than max length";
    private static final String PASSWORD_TOO_LONG = "Password should be no longer than %d
characters";
    private static final String PASSWORD_NO_DIGIT = "Password should contain at least 1 digit";
    private static final String PASSWORD_NO_LOWERCASE = "Password should contain at least 1
lowercase letter";
    private static final String PASSWORD_NO_UPPERCASE = "Password should contain at least 1
uppercase letter";
    private static final String PASSWORD_NO_SPECIAL_SYMBOL = "Password should contain at least 1
special symbol (" +
            specialSymbols.stream().map(e -> e + "").collect(Collectors.joining(", ")) + ")";

    private static final int MINLENGTH = 16;
    private static final int MAXLENGTH = 30;
    private static final boolean CONTAINSDIGIT = true;
    private static final boolean CONTAINSLOWERCASE = true;
    private static final boolean CONTAINSUPPERCASE = true;
    private static final boolean CONTAINSSPECIALSYMBOL = true;


    @Override
    public void initialize(Password constraintAnnotation) {
        ConstraintValidator.super.initialize(constraintAnnotation);
        this.password = constraintAnnotation;
    }

    @Override
    public boolean isValid(String value, ConstraintValidatorContext context) {
        this.value = value;
        this.context = context;
        context.disableDefaultConstraintViolation();

    if (this.value.length() < MINLENGTH) {
        setMessage(String.format(PASSWORD_TOO_SHORT, MINLENGTH));
        violationsCount++;
    }

    if (this.value.length() > MAXLENGTH) {
        setMessage(String.format(PASSWORD_TOO_LONG, MAXLENGTH));
        violationsCount++;
    }

    if (!doesContainDigit()) {
        violationsCount++;
        setMessage(PASSWORD_NO_DIGIT);
    }

    if (!doesContainLowercase()) {
        violationsCount++;
```

```
            setMessage(PASSWORD_NO_LOWERCASE);
        }

        if (!doesContainUppercase()) {
            violationsCount++;
            setMessage(PASSWORD_NO_UPPERCASE);
        }

        if (!doesContainSpecialSymbol()) {
            violationsCount++;
            setMessage(PASSWORD_NO_SPECIAL_SYMBOL);
        }

                if (violationsCount == 0) {
            return true;
        } else {
            violationsCount = 0;
            return false;
        }

        private void setMessage(String message) {
            this.context.buildConstraintViolationWithTemplate(message).addConstraintViolation();
        }
    }
```



Вариант 3 – валидация с анотация `@Valid` в Web частта

https://www.baeldung.com/spring-mvc-custom-validator

```
@PostMapping(value = "/users/register")
    public String doRegister(@Valid RegistrationDTO dto, BindingResult validationResult) {
//        RegistrationDTO dto = new RegistrationDTO("user", "pass", "pass", "mail@abv.bg");
        if (validationResult.hasErrors()) {
            return "user/register";
        }

        userService.register(dto);

        return "user/login";
    }
}
```

Creating @Beans = annotating as @Component - example

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration  //Indicates that a class declares one or more @Bean methods and may be processed by the Spring container to generate bean definitions and service requests for those beans at runtime, for example:
public class ApplicationBeanConfiguration {
    @Bean   //Indicates that a method produces a bean to be managed by the Spring container
    public Gson gson() {
        return new GsonBuilder()
```

```
                .excludeFieldsWithoutExposeAnnotation()
                .setPrettyPrinting()
                .create();
    }

    @Bean //Indicates that a method produces a bean to be managed by the Spring container
    public ModelMapper modelMapper() {
        ModelMapper modelMapper = new ModelMapper();

        modelMapper.addConverter(new Converter<String, LocalDate>() {
            @Override
            public LocalDate convert(MappingContext<String, LocalDate> mappingContext) {
                return LocalDate.parse(mappingContext.getSource(),
DateTimeFormatter.ofPattern("dd/MM/yyyy"));
            }
        });

        modelMapper.addConverter(new Converter<String, LocalDateTime>() {
            @Override
            public LocalDateTime convert(MappingContext<String, LocalDateTime> mappingContext) {
                return LocalDateTime.parse(mappingContext.getSource(),
DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss"));
            }
        });

modelMapper.addConverter(new Converter<String, LocalTime>() {
    @Override
    public LocalTime convert(MappingContext<String, LocalTime> mappingContext) {
        return LocalTime.parse(mappingContext.getSource(),
DateTimeFormatter.ofPattern("HH:mm:ss"));
    }
});


        return modelMapper;
    }
}


@Component
public class ValidationUtilImpl implements ValidationUtil {
    private Validator validator;

    public ValidationUtilImpl() {
        this.validator = Validation.buildDefaultValidatorFactory().getValidator();
    }

    @Override
    public <E> boolean isValid(E dto) {
        Set<ConstraintViolation<E>> validationErrors = this.validator.validate(dto);

        if (validationErrors.isEmpty()) {
            return true; //no errors, we can input this object ~~(no such dto in the database)~~
        } else {
            return false; //errors present, cannot input this object into the db ~~(there is such a
dto already in the database)~~
        }
    }
}
```

```
@Component
public class XmlParserImpl implements XmlParser {

    @Override
    @SuppressWarnings("unchecked")
    public <T> T fromFile(String filePath, Class<T> tClass) throws JAXBException,
FileNotFoundException {
        JAXBContext jaxbContext = JAXBContext.newInstance(tClass);
        Unmarshaller unmarshaller = jaxbContext.createUnmarshaller();

        return (T) unmarshaller.unmarshal(new FileReader(filePath));
    }
}
```

Можем и така за валидатора – без Component:

```
public class MyValidator {
    private final Validator validator;

    public MyValidator() {
        this.validator = Validation.buildDefaultValidatorFactory().getValidator();
    }

    public <E> boolean isValid(E dto) {
        Set<ConstraintViolation<E>> validationErrors = this.validator.validate(dto);

        if (validationErrors.isEmpty()) {
            return true; //no such dto in the database
        } else {
            return false; //there is such a dto already in the database
        }
    }
}


@Configuration
public class BeanConfiguration {
    @Bean
    public MyValidator getValidator(){
        return new MyValidator();
    }

}
```

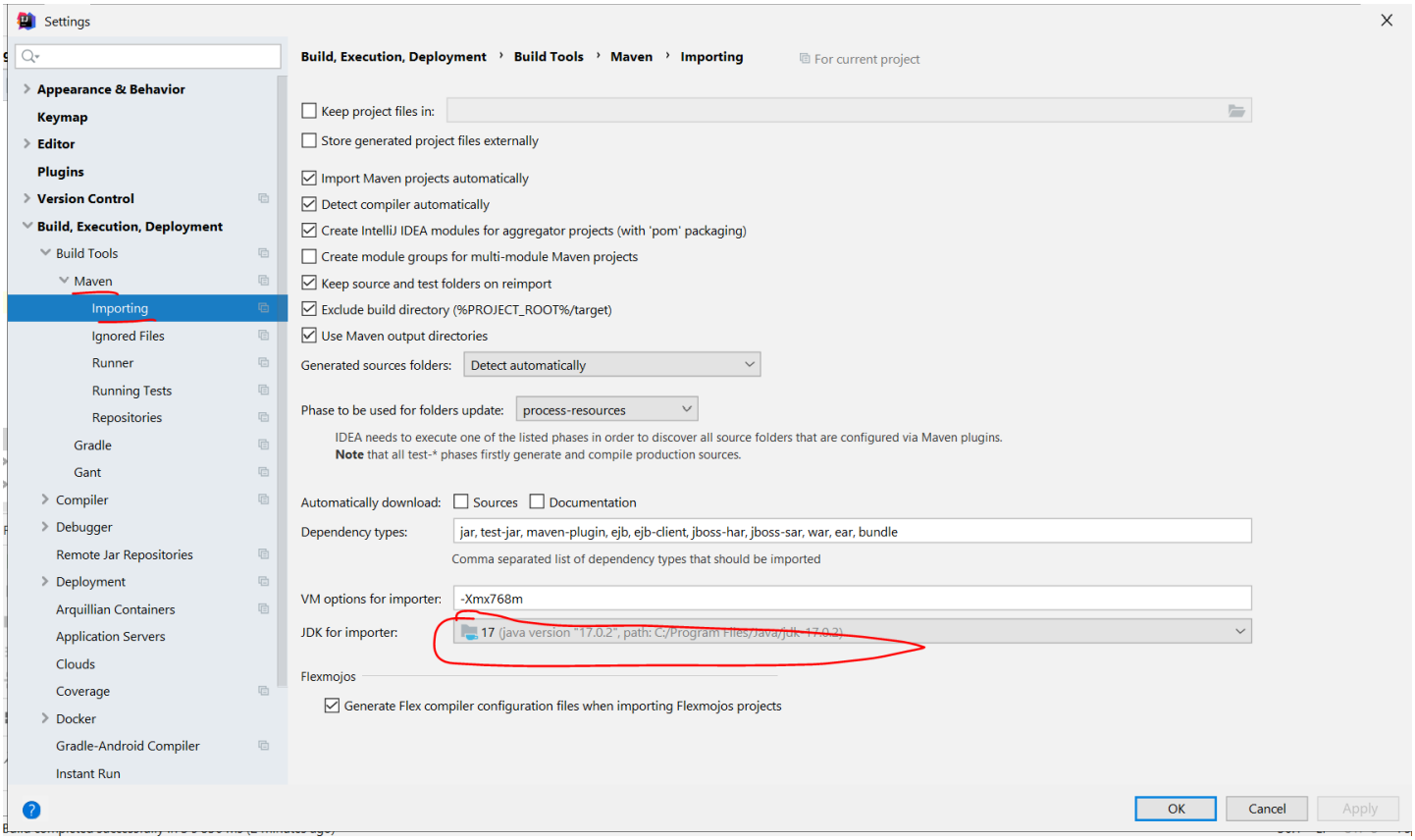
Два modelMappers

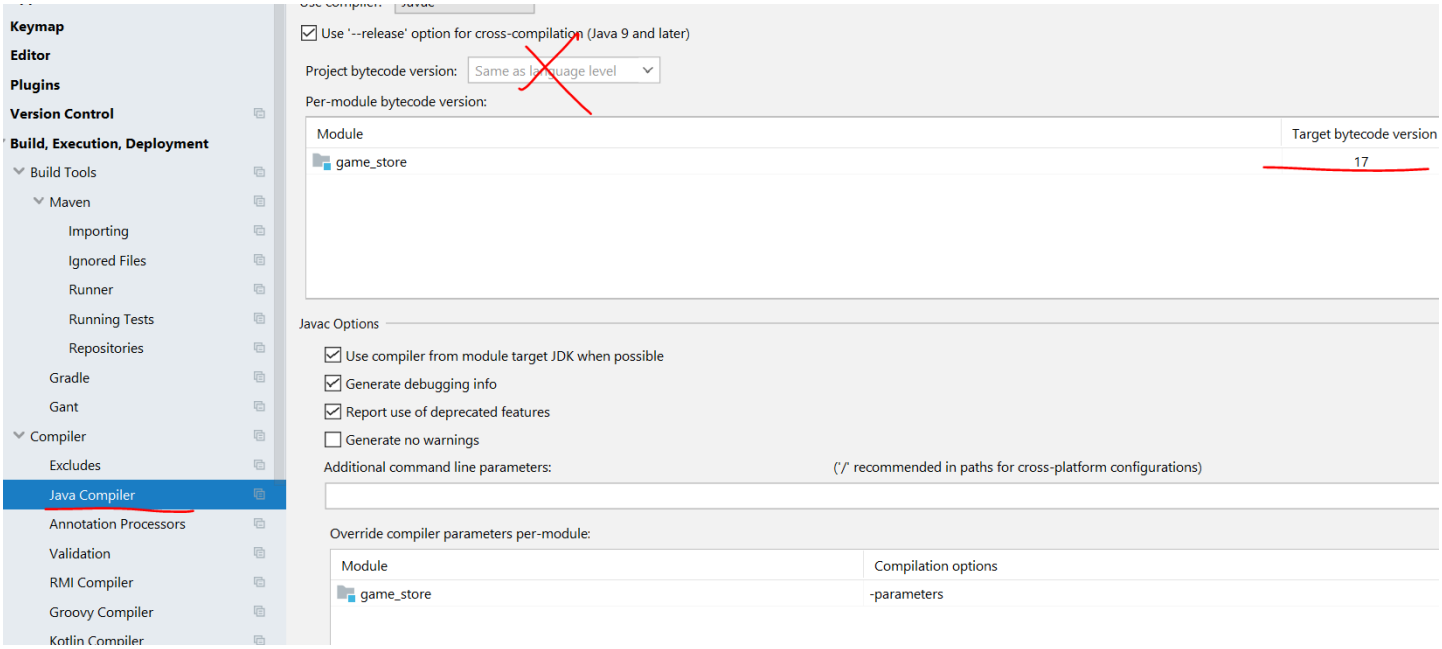
Когато не искаме всеки път modelMapper-а да има едни и същи екстра converter-и – за много обекти може да не се налага всеки път по един и съши начин да се convert-в и това е начина да разграничим.

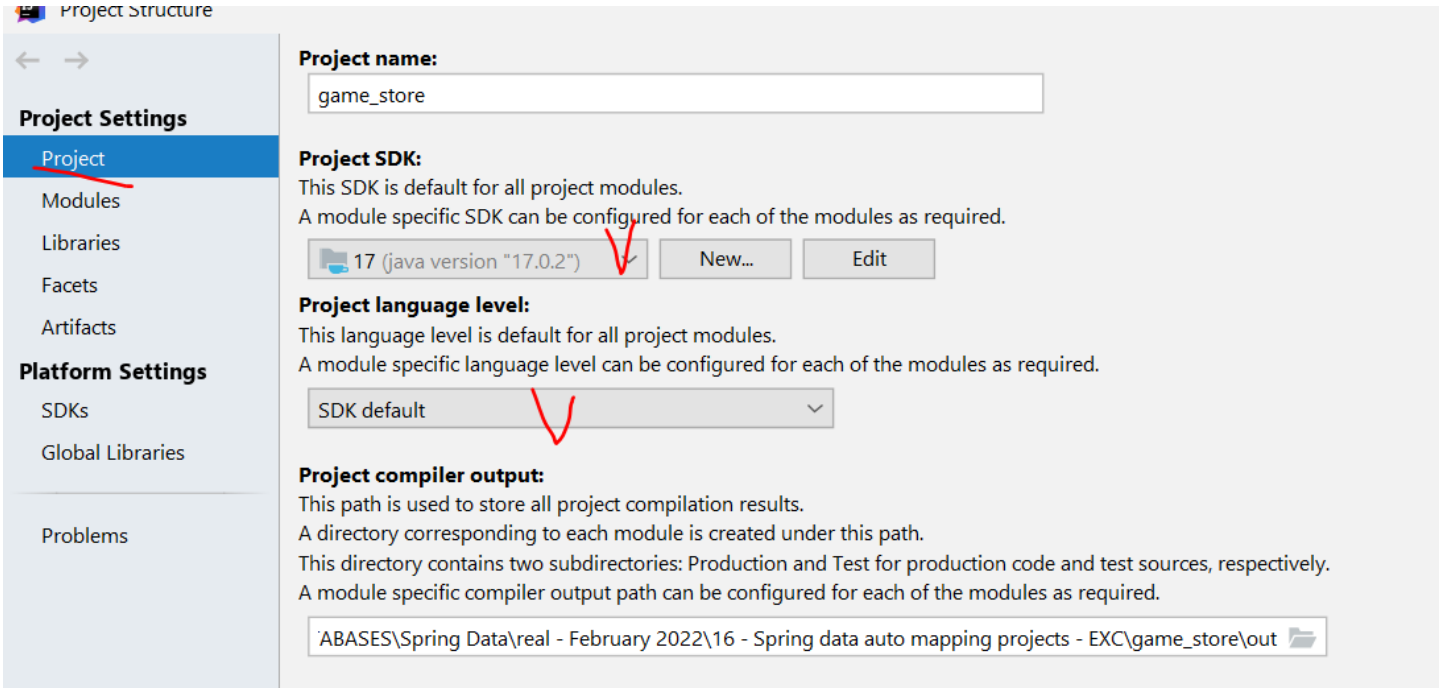
```
@Configuration
public class BeanConfiguration {
    @Bean(name = "default")
        @Primary //ако не сме оказали Qualifier, то винаги ползвай тази инстанция

    public ModelMapper getModelMapper(){
        return new ModelMapper();
    }
```

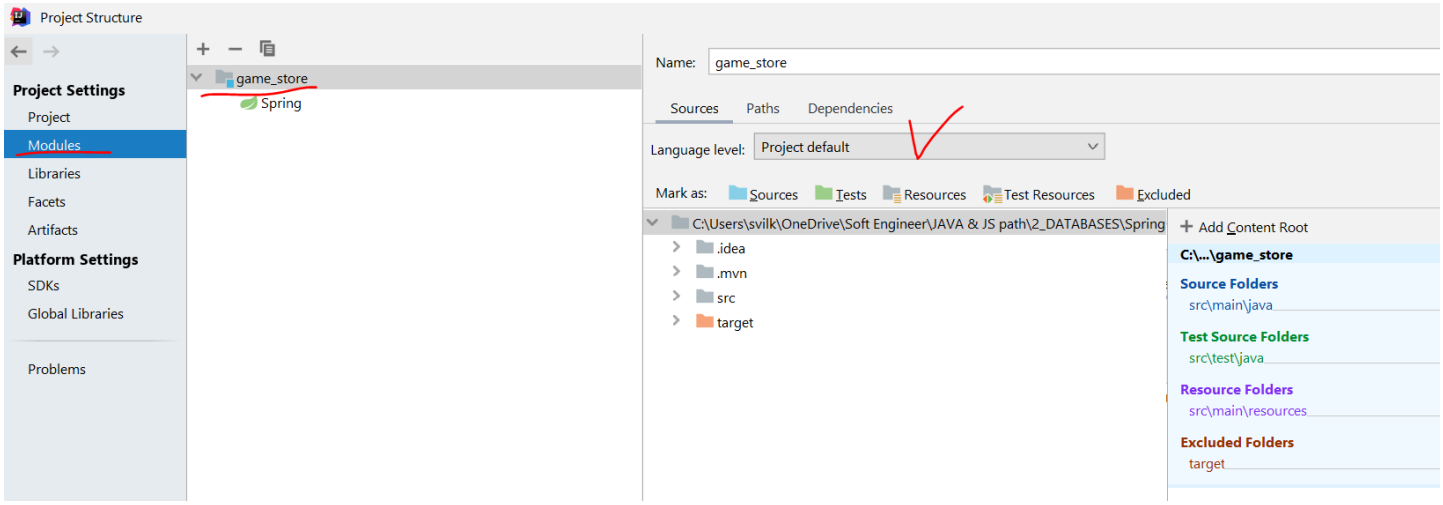
```
    @Bean(name = "alternative")
    public ModelMapper getAlternativeModelMapper(){
        return new ModelMapper();
    }
}


@Service
public class UserService {
    private ModelMapper modelMapper;
    private UserRepository userRepository;

    @Autowired
    public UserService(@Qualifier(value = "alternative") ModelMapper modelMapper, UserRepository
userRepository) {
        this.modelMapper = modelMapper;
        this.userRepository = userRepository;
    }
}
```

Java 17 да се подкара със Switch expression:




Keymap
Editor
Plugins
Version Control
Build, Execution, Deployment
Build Tools
Maven
Importing
Ignored Files
Runner
Running Tests
Repositories
Gradle
Gant
Compiler
Excludes
Java Compiler
Annotation Processors
Validation
RMI Compiler
Groovy Compiler
Kotlin Compiler
Use '--release' option for cross-compilation (Java 9 and later)
Project bytecode version:
Same as language level
Per-module bytecode version:
Module
Target bytecode version
game_store
17
Javac Options
Use compiler from module target JDK when possible
Generate debugging info
Report use of deprecated features
Generate no warnings
Additional command line parameters:
('/' recommended in paths for cross-platform configurations)
Override compiler parameters per-module:
Module
Compilation options
game_store
-parameters


Project Structure
Project Settings
Project
Modules
Libraries
Facets
Artifacts
Platform Settings
SDKs
Global Libraries
Problems
Project name:
game_store
Project SDK:
This SDK is default for all project modules.
A module specific SDK can be configured for each of the modules as required.
17 (java version "17.0.2")
New...
Edit
Project language level:
This language level is default for all project modules.
A module specific language level can be configured for each of the modules as required.
SDK default
Project compiler output:
This path is used to store all project compilation results.
A directory corresponding to each module is created under this path.
This directory contains two subdirectories: Production and Test for production code and test sources, respectively.
A module specific compiler output path can be configured for each of the modules as required.
ABASES\Spring Data\real - February 2022\16 - Spring data auto mapping projects - EXC\game_store\out

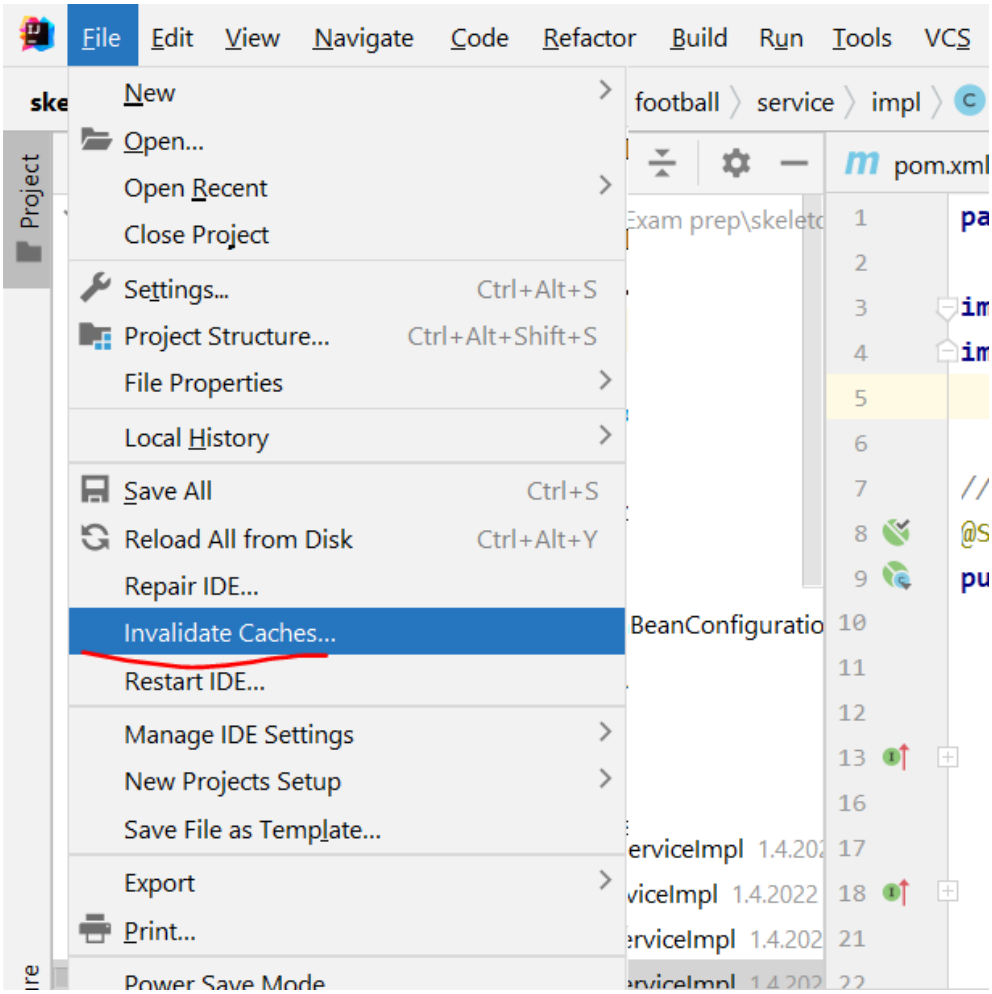


Pom.xml

```xml
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.6.4</version>
        <relativePath/> <!-- lookup parent from repository -->
    </parent>
    <groupId>com.example</groupId>
    <artifactId>demo</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <name>demo</name>
    <description>Demo project for Spring Boot</description>

    <properties>
        <java.version>17</java.version>
    </properties>

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

        <dependency>
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId>
            <scope>runtime</scope>
        </dependency>

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

        <dependency>
            <groupId>org.modelmapper</groupId>
            <artifactId>modelmapper</artifactId>
            <version>3.0.0</version>
        </dependency>

    </dependencies>

    <build>
```

```
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
        </plugins>
    </build>

</project>
```

Invalidate caches



LAZY и EAGER

.lazy = on demand – използва се когато имаме връзка 1 към 1 000 000 примерно. Когато detach-нем обект, тези 1 000 000 записа се губят като достъп.
.eager – когато detach-нем обекта, тези 1 000 000 записа са взети вече и стоят в хеша.

```
@OneToMany   e  default fetch type LAZY
@ManyToMany
```

`@ManyToOne  e  default fetch type EAGER`

Избягвай EAGER loading-а. Ако ти трябва в някой случай, в който извличаш ентитито, и негова колекция, по-добре напиши `@Query` в репозиторито, което прави `JOIN FETCH`. Например:

```
@Query("SELECT u FROM User u JOIN FETCH u.roles r WHERE u.username = :username");
User getByUsername(String username);
```

UUID

UUID – A universally unique identifier (UUID) is a 128-bit label used for information in computer systems. The term globally unique identifier (GUID) is also used. When generated according to the standard methods, UUIDs are, for practical purposes, unique.

```
@Id
@GeneratedValue(generator = "uuid-string")
@GenericGenerator(name = "uuid-string",
        strategy = "org.hibernate.id.UUIDGenerator")
public String getId() { return id; }

public void setId(String id) { this.id = id; }
```

Не е добра идея да слагаме UUID като primary key – прекалено бавно стават insert-е поради проблем с A-B дърво алгоритми.

Other

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

В Spring класове анотирани с @Configuration заместват .xml конфигурациите

csv – comma separated value file (при адресни книги го има)

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

Интеграционни тестове – тестове за повече от един unit/метод

DTO vs DAO – нямат нищо общо
Data Transfer Object – обикновено свързано със заявка на потребител
Data Access Object (DAO) – сходно на Repository – отново отговарят да комуникират с базата

Embeddable JPA Javadoc – вместо вградени класове
javax.persistence

Composite primary key in JPA

Option 1 – with two @Id and implements Serializable

```
public class UserPK implements Serializable {
        @GeneratedValue(strategy = GenerationType.IDENTITY)
    private long id;

        @Column(nullable = false, unique = true)
    private String username;
}


@Entity
@Table(name = "users")
@IdClass(UserPK.class)
public class User implements Serializable {
    @Id  //визуално да са тук
    private long id;

    @Id
    private String username;
```

Option 2 – using `@Embeddable`

```
import javax.persistence.Embeddable;
import java.io.Serializable;

@Embeddable
public class UserPK implements Serializable {
    private long id;
    private String username;
}



@Entity
@Table(name = "users")
public class User implements Serializable {
    @EmbeddedId
    private UserPK primaryKey;


@AttributeOverride()
private String username;


@Entity
@Table(name = "routes")
public class Route {
    @ManyToOne
    @MapsId("id")
    private User author;
```

WARN: Щом ще имаме композитен ключ в MySQL, то добре е да имплементираме в рамките на Java hasCode() и equals().

```
2022-06-09 19:35:12.668  WARN 18312 --- [  restartedMain] org.hibernate.mapping.RootClass          : HHH000038: Composite-id class does not override equals(): bg.softuni
 .pathfinder.model.UserPK
2022-06-09 19:35:12.668  WARN 18312 --- [  restartedMain] org.hibernate.mapping.RootClass          : HHH000039: Composite-id class does not override hashCode(): bg.softuni
 .pathfinder.model.UserPK
2022-06-09 19:35:13.080  WARN 18312 --- [  restartedMain] o.h.t.s.i.ExceptionHandlerLoggedImpl     : GenerationTarget encountered exception accepting command : Error
 executing DDL "
```

Composite unique key, but not primary

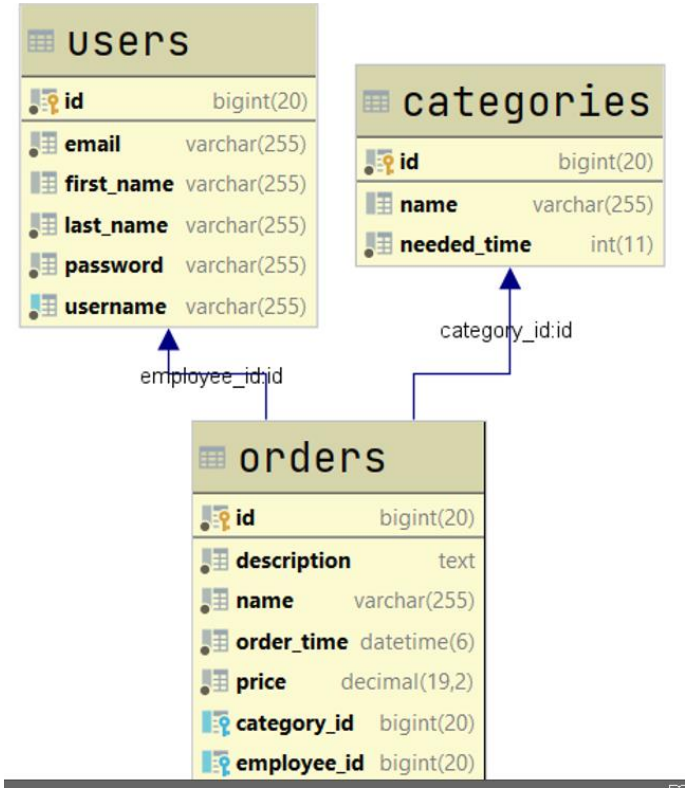
We may use this more often:

```
@Entity
@Table(name = "users",
uniqueConstraints = @UniqueConstraint(columnNames = {"username", "email"}))
public class User implements Serializable {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private long id;

    @Column(nullable = false, unique = true)
    private String username;

    @Column(nullable = false)
    private String password;

    @Column(unique = true)
    private String email;
```

Schema DB in IntelliJ - JDBC connecting with our DB

Със синичко е unique

С долна точка е задължително not null



ObjectMapper на Jackson на Spring

```
@Test
public void createComment_sampleData_commentIsReturnedAsExpected() throws Exception {
    when(commentService.createComment(any())).thenAnswer(interaction -> {
        CommentCreationDto commentCreationDto = interaction.getArgument( 0);
        return new CommentDisplayView( id: 1L, commentCreationDto.getUsername(), commentCreationDto.getMessage());
    });
    CommentMessageDto commentMessageDto = new CommentMessageDto("This is comment #1");
    ObjectMapper objectMapper = new ObjectMapper();

    mockMvc.perform(post( urlTemplate: "/api/" + ROUTE_ID + "/comments")
            .content(objectMapper.writeValueAsString(commentMessageDto)))
            .andExpect(status().is( status: 201))
            .andExpect(jsonPath( expression: "$.message", is( value: "This is comment #1")));
}
```

Flyway

```
-------------
#Flyway properties
flyway:
 enabled: true
 schemas: migrations
 validate-on-migrate: true
 locations:
   classpath:db.migration.mysql
 baseline-on-migrate: false
```

```
<dependency>
    <groupId>org.flywaydb</groupId>
    <artifactId>flyway-core</artifactId>
</dependency>
<dependency>
    <groupId>org.flywaydb</groupId>
    <artifactId>flyway-mysql</artifactId>
</dependency>
<dependency>
    <groupId>org.flywaydb</groupId>
    <artifactId>flyway-maven-plugin</artifactId>
    <version>9.16.0</version>
</dependency>
```