# Konfiguracja bazy danych H2

To baza danych przechowywana w pamięci. Resetuje się za każdym razem gdy uruchamiamy nasz program od nowa.

1)dodanie do POM xml odpowiedniej zależności

<!-- https://mvnrepository.com/artifact/com.h2database/h2 -->

<dependency>

<groupId>com.h2database</groupId>

<artifactId>h2</artifactId>

<version>1.0.60</version>

</dependency>

2)podłączenie do bazy danych

**Klasa AppH2**

**private static** String *h2Driver*=**"org.h2.Driver"**;  
**private static** String *h2Address*=**"jdbc:h2:mem:test\_mem;DB\_CLOSE\_DELAY=-1"**;  
*//domyślne wartości dla bazy danych w pamięci***private static** String *user*=**""**;  
**private static** String *password*=**""**;

By default, closing the last connection to a database closes the database. For an in-memory database, this means the content is lost. To keep the database open, add ;DB\_CLOSE\_DELAY=-1 to the database URL. To keep the content of an in-memory database as long as the virtual machine is alive, use jdbc:h2:mem:test;DB\_CLOSE\_DELAY=-1.

**public static** Connection getConnection() **throws** ClassNotFoundException, SQLException {  
 Connection connection=**null**;  
 Class.*forName*(*h2Driver*);  
 connection=DriverManager.*getConnection*(*h2Address*,*user*,*password*);  
 **return** connection;

}

**public static void** main(String[] args) **throws** SQLException, ClassNotFoundException {  
 Connection connection=*getConnection*();  
 **if** (connection!=**null**){  
 System.***out***.println(**"SUKCES"**);  
 }  
}

# JDBC

JDBC polega na przesyłaniu zapytań do bazy.

Będziemy chcieli wrzucić obiekty klasy Student do bazy danych.

@AllArgsConstructor  
@Getter  
@Setter  
@NoArgsConstructor

@ToString  
**public class** Student {  
 **private int id**;  
 **private** String **name**;  
}

Utworzenie tabeli USERS w bazie danych w pamięci (H2)

**private static void** createTableForStudent() **throws** SQLException, ClassNotFoundException {  
 Connection connection=AppH2.*getConnection*();  
 Statement statement=connection.createStatement();  
 String query=**"CREATE TABLE STUDENTS(id int primary key,name varchar(255))"**;  
 statement.execute(query);  
 connection.commit();  
}

Wstawienie do bazy danych obiektów klasy Student

**private static void** insertStudent(Student student) **throws** SQLException, ClassNotFoundException {  
 Connection connection=AppH2.*getConnection*();  
 Statement statement=connection.createStatement();  
 String query=**"INSERT INTO STUDENTS VALUES("**+  
 student.getId()+**",\'"**+student.getName()+**"\')"**;  
 statement.execute(query);  
 connection.commit();  
}

Pobranie studentów

**public static** List<Student> getStudent() **throws** SQLException, ClassNotFoundException {  
 List<Student> students=**new** ArrayList<>();  
 Connection connection=AppH2.*getConnection*();  
 Statement statement=connection.createStatement();  
 String query=**"SELECT** *\** **FROM STUDENTS"**;  
 ResultSet resultSet=statement.executeQuery(query);  
 **while** (resultSet.next()){  
 **int** id=resultSet.getInt(**"id"**);  
 String name=resultSet.getString(**"name"**);  
 students.add(**new** Student(id,name));  
 }  
 **return** students;  
}

Jak widzimy musi tu być **ResultSet resultSet=statement.executeQuery(query);**  
bo odbieramy wyniki

# JPA

Java Persistence API – zestaw interfejsów, które wspomagają programistę w operowaniu na bazie danych. Najpopularniejszą biblioteką implementującą standard JPA jest Hibernate. Operacje są cachowane. JPA decyduje kiedy dane mają być przesłane.

# JPA/Hibernate

Dodajemy zależność w POM.xml

<**dependency**>  
 <**groupId**>org.hibernate</**groupId**>  
 <**artifactId**>hibernate-core</**artifactId**>  
 <**version**>5.2.16.Final</**version**>  
</**dependency**>

Rozwiązanie problemu gdy nie ma folderu resources

<**resources**>  
 <**resource**>  
 <**directory**>src/main/resources</**directory**>  
 </**resource**>  
</**resources**>

Należy to w <build> wpisać

**Konfiguracja JPA**

1)utworzyć katalog **META-INF** w resources a w nim plik **persistent.xml –** standardowy plik konfiguracyjny JPA.

Można również użyć pliku **hibernate.config.xml**

Hibernate jest zwykła biblioteką z której można korzystać bezpośrednio używając klas oraz korzystać z pliku konfiguracyjnego hibernate.config.xml. Jednak gdy chcemy przejść na wyższy poziom abstrakcji to od tego jest JPA i plik konfiguracyjny persistent.xml. Wtedy korzystamy z paczek JPA, a nie z paczek Hibernate. Zaletą jest fakt, że podczas implementacji naszego programu można podmienić bibliotekę Hibernate na inną która również implementuje standard JPA. Jest to zaleta tylko teoretyczna. Nie zdarza się taka sytuacja. W praktyce korzysta się z JPA tam gdzie można.

Przykładowy plik persistence.xml

*<?***xml version="1.0" encoding="UTF-8"** *?>*<**persistence xmlns="http://java.sun.com/xml/ns/persistence"  
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
 xsi:schemaLocation="http://java.sun.com/xml/ns/persistence  
 http://java.sun.com/xml/ns/persistence/persistence\_1\_0.xsd" version="1.0"**>  
  
 <**persistence-unit name="persystencja" transaction-type="RESOURCE\_LOCAL"**>  
 <**provider**>org.hibernate.jpa.HibernatePersistenceProvider</**provider**>  
  
 <**properties**>  
 <**property name="connection.driver\_class" value="org.h2.Driver"**/>  
 <**property name="hibernate.connection.url" value="jdbc:h2:./db/repository"**/>  
 <**property name="hibernate.dialect" value="org.hibernate.dialect.H2Dialect"**/>  
 <**property name="hibernate.hbm2ddl.auto" value="create-drop"**/>  
 <**property name="hibernate.show\_sql" value="true"** />  
 </**properties**>  
 </**persistence-unit**>  
</**persistence**>

JPA operuje na dwóch podstawowych obiektach

**EntityManagerFactory** – singleton. Używa się głównie do tworzenia EntityManagera.

**EntityManager** – to nie jest singleton. Odpowiada za jednorazowe połączenie z bazą danych. Komunikuje się bezpośrednio z JPA, a pośrednio z bazą danych.

**public class** AppJPA {  
 **public static void** main(String[] args) {  
 EntityManagerFactory factory= Persistence.*createEntityManagerFactory*(**"persystencja"**);  
 EntityManager manager=factory.createEntityManager();  
 }  
}

# Encja

Encja to obiekt javowy, który posiada pola i metody.

Aby ze Student zrobić encję należy dodać adnotację @Entity i inne zgodnie z tym co poniżej

Należy pamiętać, że Hibernate wymaga pustego konstruktora, nawet prywatnego.

@Entity(name = **"students"**)  
**public class** Student {  
 @Id  
 **private int id**;  
 @Column(name = **"name"**,nullable = **false**)  
 **private** String **name**;  
}

@Transient oznacza, aby tego pola nie mapować.

# Crud

Create, Read, Update, Delete

## Create

W ten sposób można **dodać studenta** do bazy

**public static void** createStudent(){  
 Student student1=**new** Student(15,**"Jan"**);  
 ***manager*.persist(student1);**  
}

Jednak nie zawsze od razu widać wynik zapytania, bo operacje przechowywane **są w cachu**. Aby zmusić JPA do działania

**public static void** createStudent(){  
 Student student1=**new** Student(15,**"Jan"**);  
 ***manager*.getTransaction().begin();**  
 *manager*.persist(student1);  
 ***manager*.getTransaction().commit();**  
}

@@LOGI

INFO: HHH000476: Executing import script 'org.hibernate.tool.schema.internal.exec.ScriptSourceInputNonExistentImpl@70a36a66'

Hibernate: insert into students (name, id) values (?, ?)

@@LOGI

**Transakcja** to wrapper na jakiś algorytm. **Transkacja** musi wykonać cała albo w ogóle.

## Read

Jak **odczytać** Studenta?

**public static void** readStudent(){  
 Student student=***manager*.find**(Student.**class**,15);  
 System.***out***.println(student);  
}

@@LOGI

Hibernate: insert into students (name, id) values (?, ?)

Student(id=15, name=Jan)

@@LOGI

A jeśli chcemy **pobrać więcej studentów?**

Musimy napisać zapytanie i użyć pseudojęzyka JPQL, o którym będzie poniżej

A najlepsza wersja jest taka

**@Entity  
@Table(name = "students")**  
**public class** Student {  
 @Id  
 **private int id**;  
 @Column(name = **"name"**,nullable = **false**)  
 **private** String **name**;  
}

Oraz

**public static void** readStudents(){  
 String query=**"FROM Student"**;  
 List<Student> studenci=*manager*.createQuery(query).getResultList();  
 studenci.forEach(System.***out***::println);  
}

## Update

Aktualizacja danych rekordu (Studenta)

**public static void** updateStudent(**int** id){  
 Student student=*manager*.find(Student.**class**,id);  
 student.setName(**"Tomasz"**);  
 *manager*.getTransaction().begin();  
 ***manager*.merge(student);**  
 *manager*.getTransaction().commit();  
}

merge – jeśli nie ma studenta, to zapisuje go do bazy danych. Merge można użyć jeszcze wtedy gdy chcemy, aby obiekt był zwrócony od razu.

@@LOGI

Student(id=15, name=Jan)

Student(id=11, name=Józef)

Student(id=1, name=Tomasz)

@@LOGI

## Delete

Kasowanie

**public static void** removeStudent(**int** id){  
 Student student=*manager*.find(Student.**class**,id);  
 *manager*.getTransaction().begin();  
 *manager*.remove(student);  
 *manager*.getTransaction().commit();  
}

# Embedded

A co jeśli chcemy w naszej klasie zrobić kompozycję?

**@Embeddable** – klasa ta jest wstawiana do innej tabeli(klasy). I wtedy w klasie w której występuje należy użyć **@Embedded** przed nazwą zmiennej

@AllArgsConstructor  
@NoArgsConstructor  
@Embeddable  
**public class** Address {  
 **private** String **street**;  
 **private** String **postalCode**;  
 **private** String **city**;  
}

@AllArgsConstructor  
@Getter  
@Setter  
@NoArgsConstructor  
@ToString  
@Entity  
@Table(name = **"students"**)

**public class** Student {  
 @Id  
 **private int id**;  
 @Column(name = **"name"**,nullable = **false**)  
 **private** String **name**;  
 @Embedded  
 **private** Address **address**;  
}

# OneToOne

Połączymy tabelę Students z tabelą Indeks

W tabeli Students

@OneToOne  
**private** Indeks **indeks**;

Wtedy w main

*manager*.getTransaction().begin();  
Student jan=**new** Student(15,**"Jan"**); // lub merge  
Indeks indeks=**new** Indeks(1,**"indeks1"**); //lub merge  
jan.setIndeks(indeks);  
*manager*.getTransaction().commit();  
System.***out***.println(jan);

UWAGA: indeks musi być już w bazie danych

To jest relacje 1do1 jednostronna bo student ma indeks, ale Indeks nie ma nic o Studencie

**Jak zrobić w drugą stronę?**

W klasie Indeks

**public class** Indeks {  
 @Id  
 **private int id**;  
 **private** String **number**;  
  **@OneToOne(mappedBy = "indeks")**  
 **private** Student **student**;  
}

i na koniec

@AllArgsConstructor  
@Getter  
@Setter  
@NoArgsConstructor  
@ToString  
@Entity  
@Table(name = **"students"**)  
**public class** Student {  
 @Id  
 **private int id**;  
 @Column(name = **"name"**,nullable = **false**)  
 **private** String **name**;  
 @OneToOne  
 **@ToString.Exclude**  
 **private** Indeks **indeks**;  
 **public** Student(**int** id,String name){  
 **this**.**id**=id;  
 **this**.**name**=name;  
 }  
*// @Embedded  
// private Address address;*}

Main

**public static void** main(String[] args) {  
 *manager*.getTransaction().begin();  
 Student jan=*manager*.merge(**new** Student(15,**"Jan"**));  
 Indeks indeks=*manager*.merge(**new** Indeks(1,**"indeks1"**));  
 jan.setIndeks(indeks);  
 jan=*manager*.merge(jan);  
 indeks.setStudent(jan);  
 *manager*.merge(indeks);  
 *manager*.getTransaction().commit();  
 System.***out***.println(jan);  
  
 Indeks ind=*manager*.find(Indeks.**class**,1);  
 System.***out***.println(ind);

}

dff

# Generowanie unikalnego Id

# Operacje kaskadowe