Java Hibernate

Aplikacje przetwarzające duże ilości danych wykorzystują bazy danych. Przeważnie są to relacyjne bazy danych, jednak coraz częściej wykorzystuje się także tzw. płaskie bazy (jak csv, XML, JSON) czy też całe rozwiązania tzw. NoSQL, jak Apache Cassandra, MongoDB, IgniteDB (zarówno proste, jak i wielowymiarowe/obiektowe rozwiązania nie wykorzystujące relacji).

O ile aplikacje dzielą dane na kilka/kilkanaście tabel i/lub plików, operacje na danych nie będą skomplikowane. Bezproblemowo można wykorzystać własne metody do dodawania, modyfikowania i zarządzania danymi w bazach danych. Wystarczy zastosować odpowiednie sterowniki pozwalające na utrzymywanie połączenia do baz danych.

Jeżeli jednak aplikacja jest bardziej rozbudowana, to zarządzanie zebranymi przez nią informacjami może stać się ciężkie. Do tego dochodzi dodatkowy nakład zapoznawania się z rozkładem poszczególnych pól. Same zaś operacje pobierania/zapisywania/aktualizacji danych mogą w takich przypadkach nadmiernie wykorzystywać transfer baza danych <=> aplikacja, co znacznie wpłynie na efektywność działania całego rozwiązania.

Jednym z rozwiązań tego problemu jest framework Hibernate. Pozwala on na bezpośrednie rzutowanie danych pomiędzy klasami tworzonego programu komputerowego, a układem tabel w bazie danych. Rozwiązanie to nosi nazwę Object-Relational Mapping (ORM, O/R mapping). Ponadto rozwiązanie to, poprzez swoje wewnętrzne operacje, niweluje problemy nadmiarowego ruchu danych pomiędzy programem a serwerem baz danych.

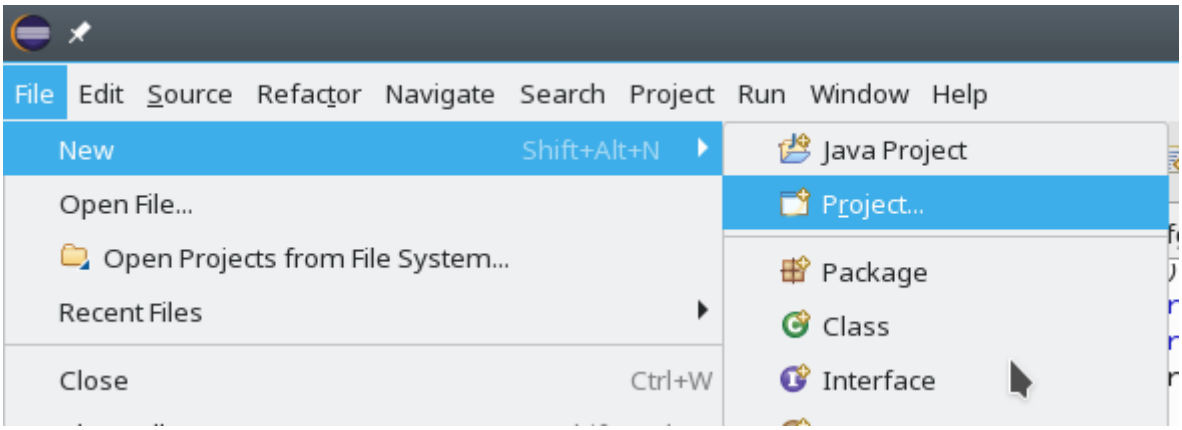
Niniejszy materiał pokaże w jaki sposób można utworzyć projekt aplikacji bazodanowej wykorzystującej Hibernate. Ponadto pokazane zostanie zestawienie działania aplikacji wykorzystującej klasyczne rozwiązania (surowy sterownik) z technologią Hibernate.

1. Wstępne rozpoczęcie projektu.

Projekt rozpoczynamy od utworzenia nowego projektu. Projekt utworzymy w środowisku Eclipse, jednak w sieci można znaleźć tworzenie projektu w NetBeans oraz bezpośrednio w linii poleceń (materiały w odnośnikach).

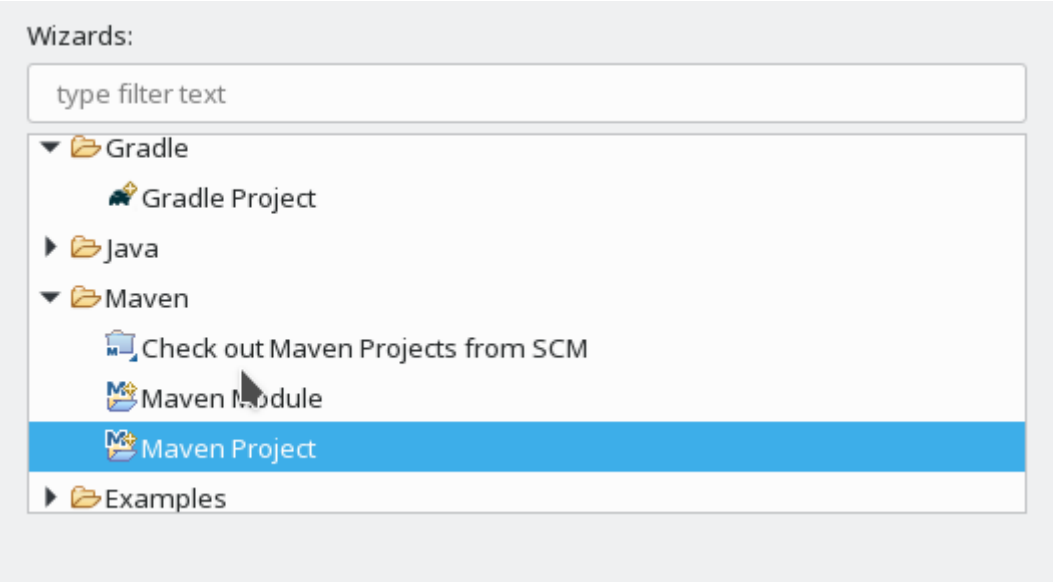
Po uruchomieniu środowiska Eclipse wybieramy tworzenie nowego projektu:

W



nowym oknie wybieramy jeden z dwóch projektów: Maven lub Gradle.

Czym są te projekty i dlaczego właśnie z nich



będziemy korzystać? Oba należą do grupy narzędzi automatyzujących budowę projektów programistycznych. Oba zostały stworzone z myślą o języku Java. W naszym przypadku głównym celem użycia jednego z tychże narzędzi ma zapewnienić działanie projektu poprzez pobranie odpowiednich bibliotek, od których projekt Hibernate będzie zależny. Jednak narzędzia te potrafią wykonywać także inne operacje, np.:

- przeprowadzać testy działania naszej aplikacji (kod z katalogu test)

- instalować nasz projekt w repozytorium Maven (naszym, lokalnym), w wyniku czego nasze pozostałe projekty będą mogły korzystać z niego jako zależności

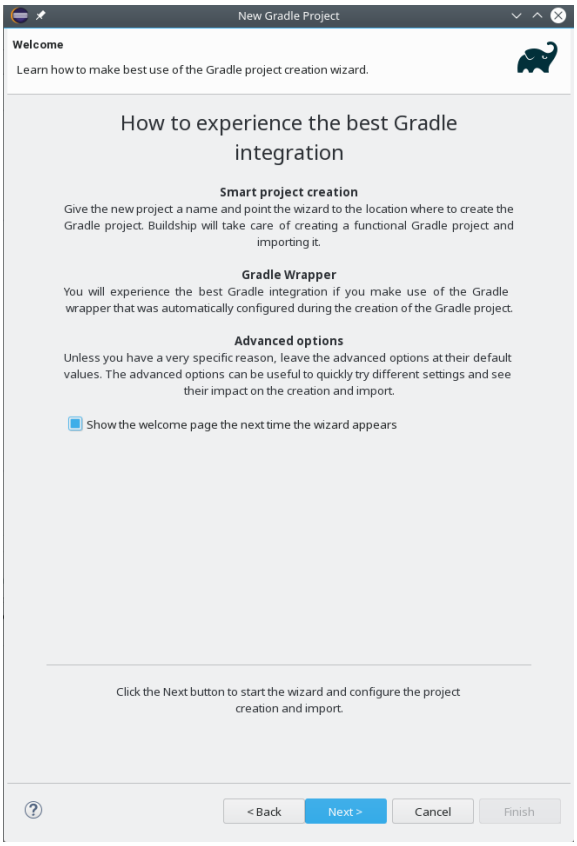
- budować tzw. paczki dystrybucyjne (gotowe oprogramowanie, ze wszystkimi zależnościami)

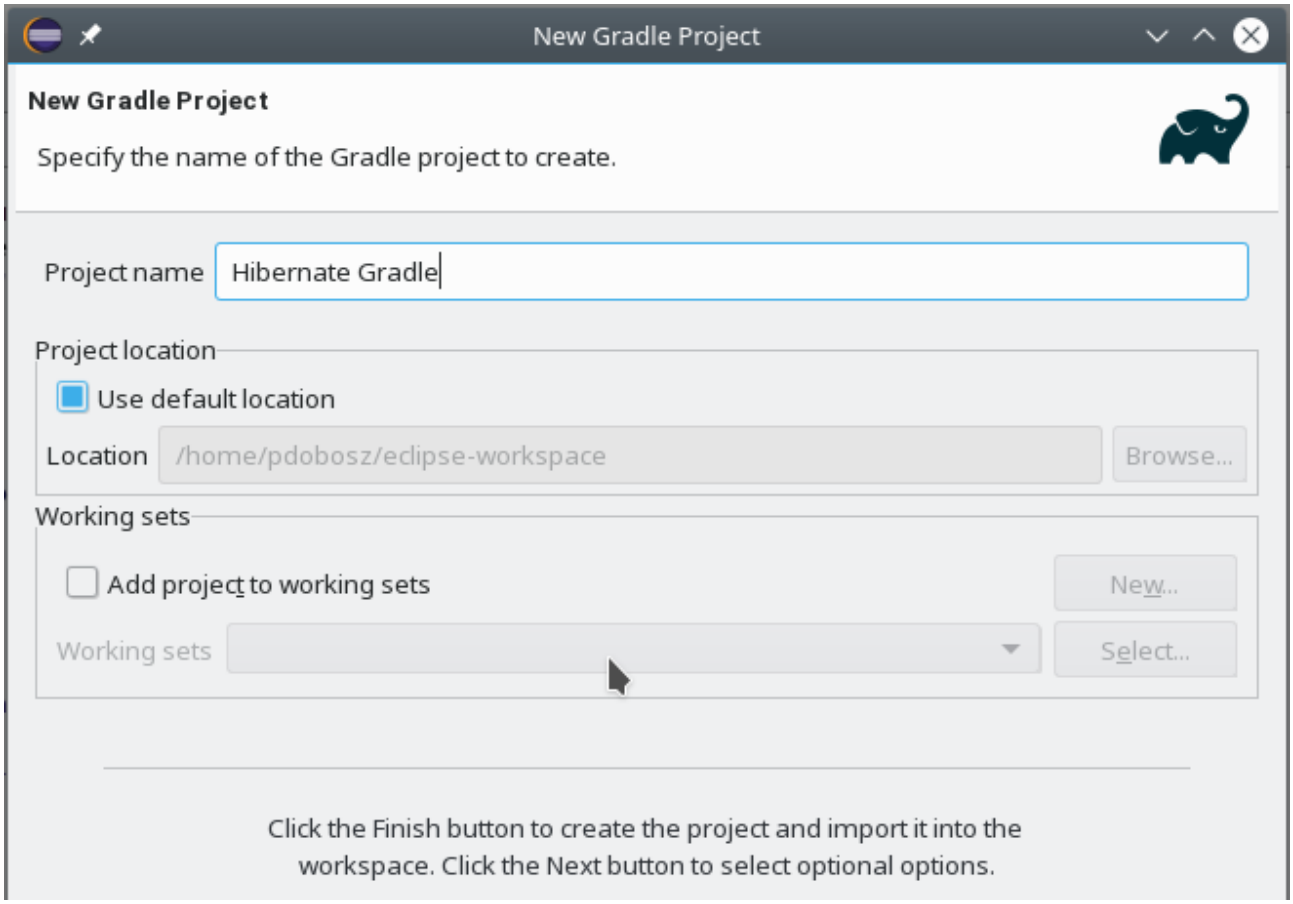
- sprawdzać projekt pod kątem poprawności i wykonywania

Obecnie może się wydawać, iż to Gradle jest bardziej popularnym rozwiązaniem dla języka Java (wykorzystywany jest chociażby przez Android Studio). Maven jest jednak prostszy w konstrukcji (większość konfiguracji zamyka się w jednym pliku XML) tak więc ostateczny wybór narzędzia zależy od programisty.

a) tworzenie projektu Gradle

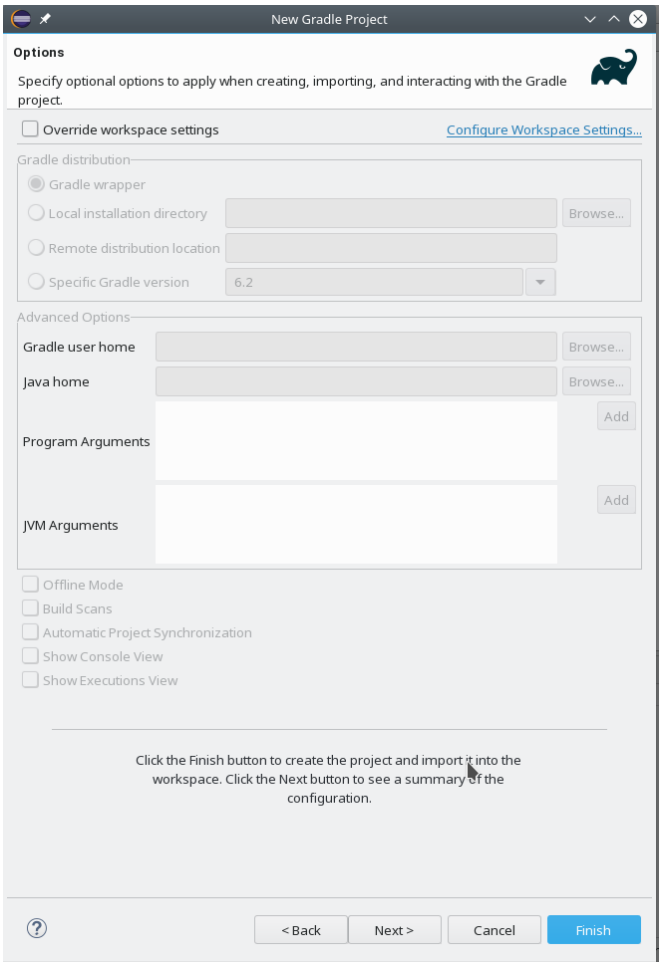
Po wyborze projektu Gradle otrzymamy informacje wstępne o działaniu narzędzia (tzw. ekran powitalny). Możemy odznaczyć jego pojawianie się w przyszłości. Tak czy inaczej przechodzimy dalej.





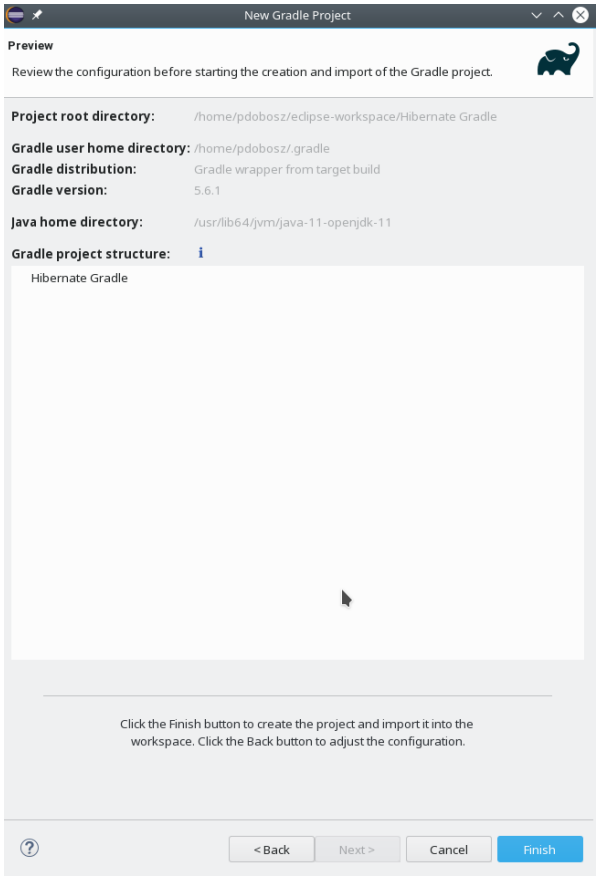
Podajemy nazwę naszego projektu. Możemy ponadto zmienić domyślny katalog projektu (przeważnie mamy jeden folder na nasze projekty). W przypadku opcji Working sets możemy ustalić katalogi, w których przechowujemy dodatkowe pliki konfiguracyjne i/lub klasy Java, z których będziemy chcieli korzystać w naszym projekcie (zostaną do niego dołączone). Ponieważ jednak tworzymy czysty projekt, ta druga opcja nie będzie nas teraz dotyczyć.

Na tym etapie możemy zakończyć konfigurowanie projektu. Możemy jednak też kliknąć przycisk dalej by zobaczyć dodatkowe ustawienia.



Jeżeli klikniemy na przycisk dalej, będziemy mieli możliwość zamiany domyślnych opcji naszej przestrzeni roboczej, podmienić domyślne katalogi dla środowiska Java, dodać nowe zmienne środowiskowe czy też włączyć dodatkowe opcje narzędzia Gradle. Na tym etapie możemy zakończyć działanie konfiguracji bądź zobaczyć podsumowanie w kolejnym widoku.

Niezależnie od naszego wyboru narzędzie skonfiguruje nasz projekt i będziemy mogli przejść do tworzenia aplikacji bazodanowej.



UWAGA: Jeżeli pierwszy raz tworzymy projekt Gradle, pobieranie dodatkowych, wymaganych modułów może zająć nieco więcej czasu. Jest to normalne i nie należy się tym przejmować. Ponadto proces ten może zakończyć się błędem lecz nie należy się tym przejmować – kolejne uruchomienie narzędzia (uruchamia się je np. poprzez ponowną kompilację projektu) powinno naprawić powstałe problemy.

Po ukończeniu tego etapu musimy dokonać aktualizacji pewnych zależności projektu. Przede wszystkim nasze narzędzie będzie musiało dociągnąć odpowiednią wersję wtyczek dla bazy danych (wykorzystujemy MariaDB/MySQL) oraz obsługę technologii Hibernate (obecna wersja to 5.4.x). Ponadto Java od wersji 9 nie posiada domyślnie obsługi oznakowań oraz podtrzymywania sesji połączenia z bazą, przez do musimy dołożyć tę zależność ręcznie. Aby tego dokonać, w pliku build.gladle dopisujemy następujące elementy:

```
api 'org.hibernate:hibernate-core:5.4.11.Final'

api 'mysql:mysql-connector-java:8.0.13'

api 'javax.activation.activation:1.1.1'

api 'javax.xml.bind:jaxb-api:2.3.0'

api 'com.sun.xml.bind:jaxb-core:2.3.0'

api 'com.sun.xml.bind:jaxb-impl:2.3.0'
```

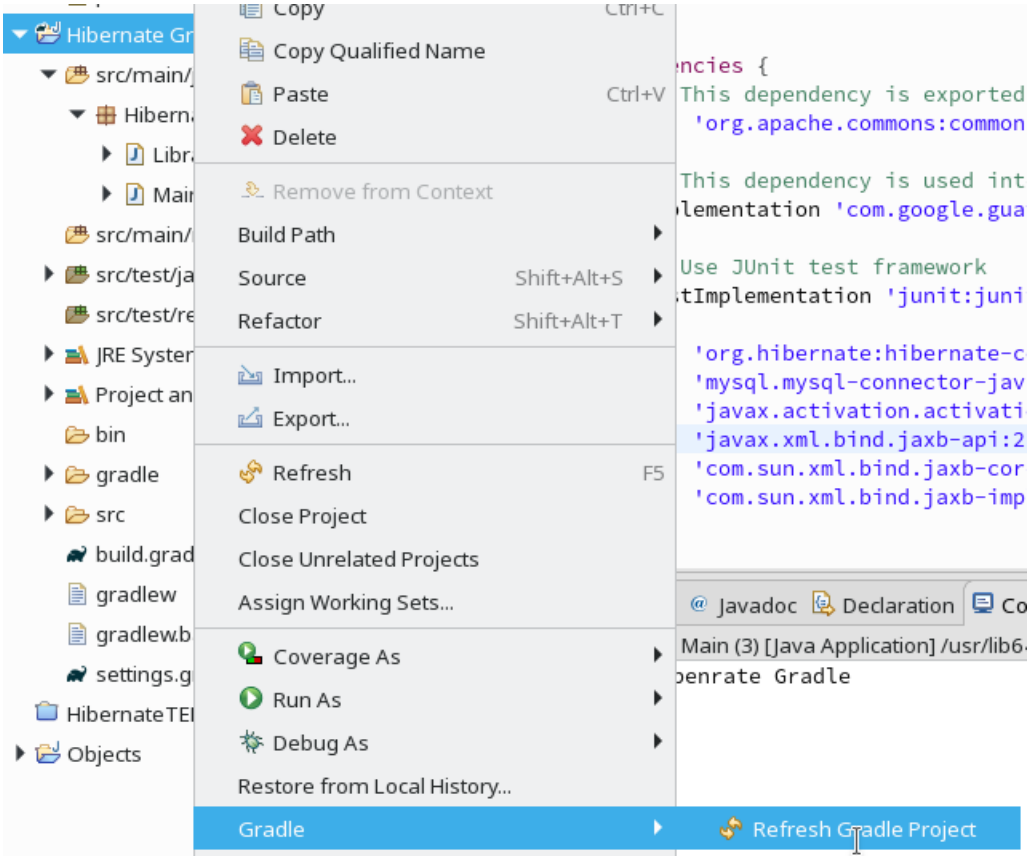
UWAGA! Powyższe wersje zależności są poprawne dla dnia 23.02.2020, dla wersji OpenJDK 11/13. Należy pamiętać, że kolejne wydania Java oraz poszczególnych technologii mogą wymagać aktualizacji wersji zależności!

Poniżej zrzut ekranu przestawiający umiejscowienie wskazanych instrukcji:

```
/*
 * This file was generated by the Gradle 'init' task.
 *
 * This generated file contains a sample Java Library project to get you started.
 * For more details take a look at the Java Libraries chapter in the Gradle
 * User Manual available at https://docs.gradle.org/5.6.1/userguide/java_library_plugin.html
 */

plugins {
    // Apply the java-library plugin to add support for Java Library
    id 'java-library'
}

repositories {
    // Use jcenter for resolving dependencies.
    // You can declare any Maven/Ivy/file repository here.
    jcenter()
}

dependencies {
    // This dependency is exported to consumers, that is to say found on their compile classpath.
    api 'org.apache.commons:commons-math3:3.6.1'

    // This dependency is used internally, and not exposed to consumers on their own compile classpath.
    implementation 'com.google.guava:guava:28.0-jre'

    // Use JUnit test framework
    testImplementation 'junit:junit:4.12'

    api 'org.hibernate:hibernate-core:5.4.11.Final'
    api 'mysql:mysql-connector-java:8.0.13'
    api 'javax.activation:activation:1.1.1'
    api 'javax.xml.bind:jaxb-api:2.3.0'
    api 'com.sun.xml.bind:jaxb-core:2.3.0'
    api 'com.sun.xml.bind:jaxb-impl:2.3.0'
}

```

Po dodaniu elementów możemy odświeżyć nasze zależności poprzez kliknięcie prawym przyciskiem myszy na projekcie, wybraniu sekcji Gradle i odświeżeniu projektu:

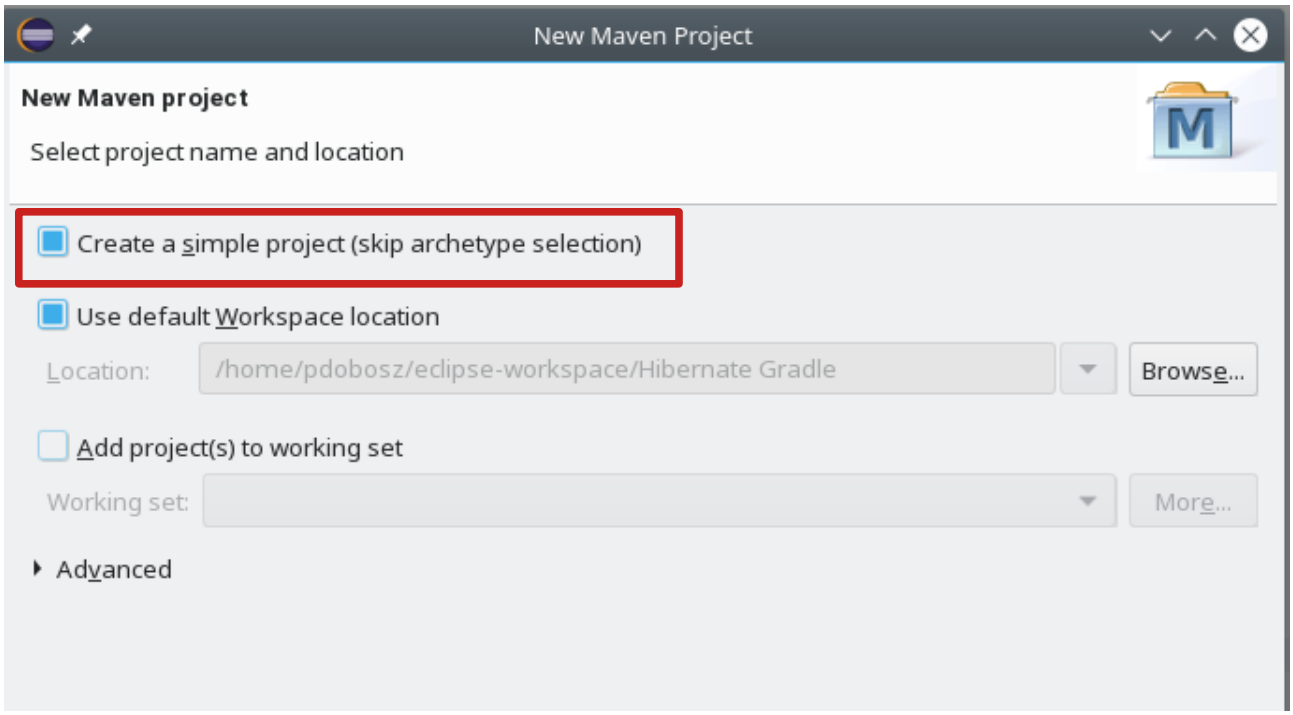
Teraz możemy



kontynuować pracę nad projektem Hibernate. W kolejnej sekcji opisane zostanie utworzenie

b) tworzenie projektu Maven

Projekty Maven, w przeciwieństwie do Gradle, posiadają swoje tzw. archetypy. Archetyp to po prostu szablon projektu, dzięki któremu dołączane są domyślne określone wtyczki i zależności projektu w taki sposób, by programista musiał spędzić jak najmniej czasu na konfiguracji środowiska, a jak najwięcej na kodowaniu. Ponieważ jednak archetypy nie posiadają odpowiedniej konfiguracji dla naszego projektu, wybierzemy opcję projektu prostego, przez co większość konfiguracji spadnie na nas. W związku z tym opcja zaznaczona na czerwono musi zostać zaznaczona.



Po kliknięciu przycisku Dalej będziemy zobligowani do podania informacji dotyczących naszego projektu:

- Group Id – będzie to nazwa naszej paczki w Java, skąd będzie pobierany kod źródłowy aplikacji (tzw. główna paczka). Możemy podać dowolną nazwę w konwencji paczek, np. samo słowo maven, maven.hibernate, teb.hibernate.maven czy dowolną inną kombinację. Obowiązują tutaj takie same ograniczenia jak w przypadku „tradycyjnych” paczek, tj. nie można nazywać ich wielką literą.

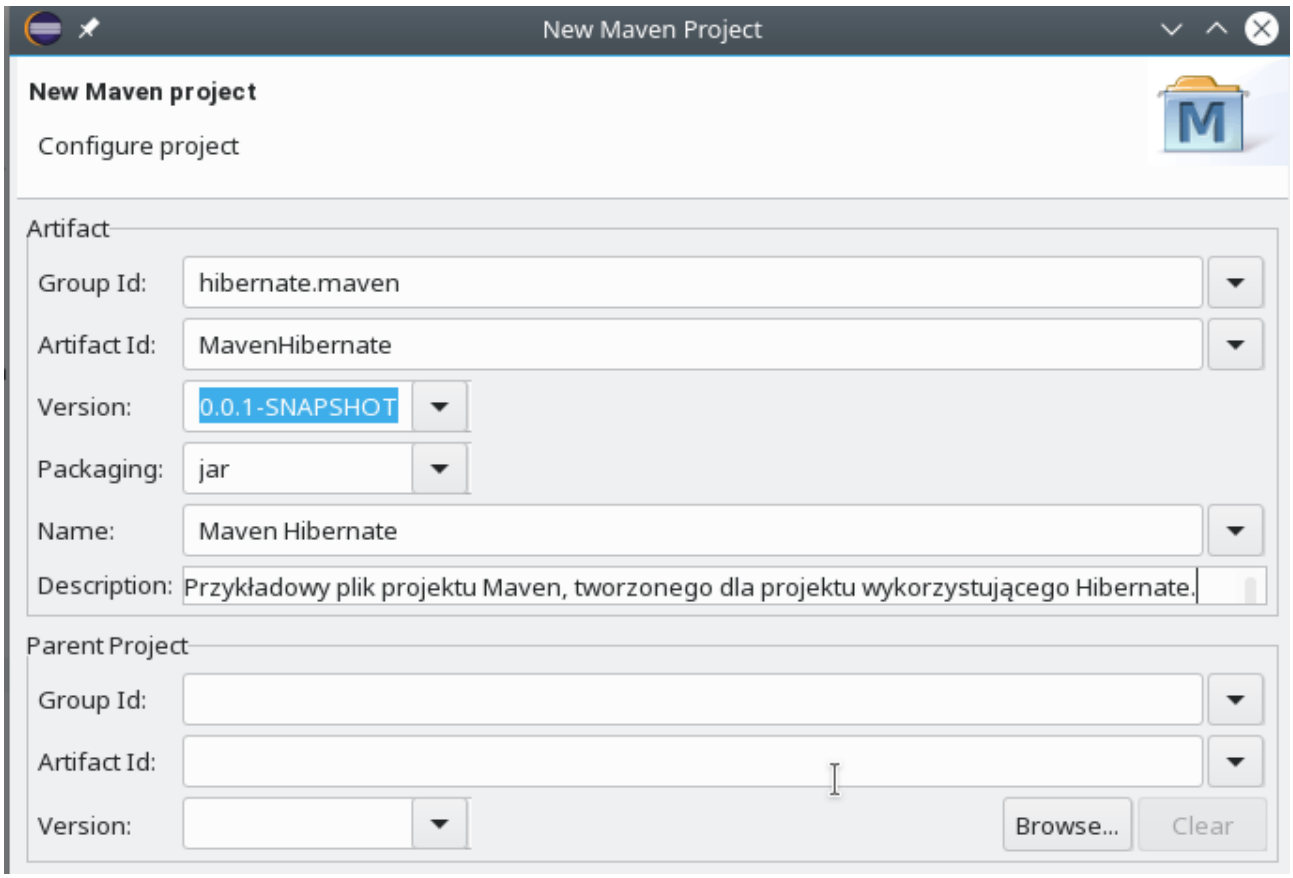
- Artifact Id – nazwa, pod którą będzie występować nasz projekt w lokalnym repozytorium Maven. Należy pamiętać, że grupa może posiadać więcej niż jeden projekt, zaś nazwa Artifact Id musi być unikatowa (jeżeli nie chcemy spowodować konfliktów pomiędzy projektami)

- Version – tutaj podczas konfiguracji mamy tylko do dyspozycji wartość 0.0.1-SNAPSHOT. Wersja określa stan naszego projektu w repozytorium (obecnie Maven określa go jako startowy). Numery wersji oraz stan projektu (SNAPSHOT oznacza projekt, który jest w fazie „burzliwego” rozwoju i korzystamy z jego obecnej wersji, z błędami i niedociągnięciami) będziemy mieli możliwość zmiany ręcznie w pliku pom.xml.

- Pachaging – określamy sposób dystrybucji naszego projektu. Jar będzie tutaj najlepszym wyborem, gdyż dzięki temu nasz projekt będzie mógł być uruchamiany na dowolnej maszynie z zainstalowanym JVM. Pozostałe opcje przeznaczone są dla repozytoriów bibliotek.

- Name – możemy określić nazwę projektu, pod jaką będzie on widniał w naszej przestrzeni roboczej wszystkich projektów. Nazwa projektu nie musi być tożsama z Artifact Id ani Group Id (innymi słowy może być dowolna)

- Description – dodatkowy opis projektu, może być pominięty



Opcje ParentProject zarezerwowane są dla przypadku, gdybyśmy tworzyli projekt zależny od innego, gotowego już projektu (albo projektu przez nas rozwijanego). Ponieważ tworzymy projekt świeży, opcje te pomijamy.

Teraz musimy dodać zależności projektowe. W pliku pom.xml dokładamy następującą sekcję (domyślnie nie jest ona dokładana przez pusty projekt):

```
<dependencies>
  <dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-core</artifactId>
    <version>5.2.6.Final</version>
  </dependency>
  <dependency>
     <groupId>mysql</groupId>
     <artifactId>mysql-connector-java</artifactId>
      <version>8.0.13</version>
  </dependency>
    <dependency>
    <groupId>javax.activation</groupId>
    <artifactId>activation</artifactId>
    <version>1.1.1</version>
  </dependency>
```

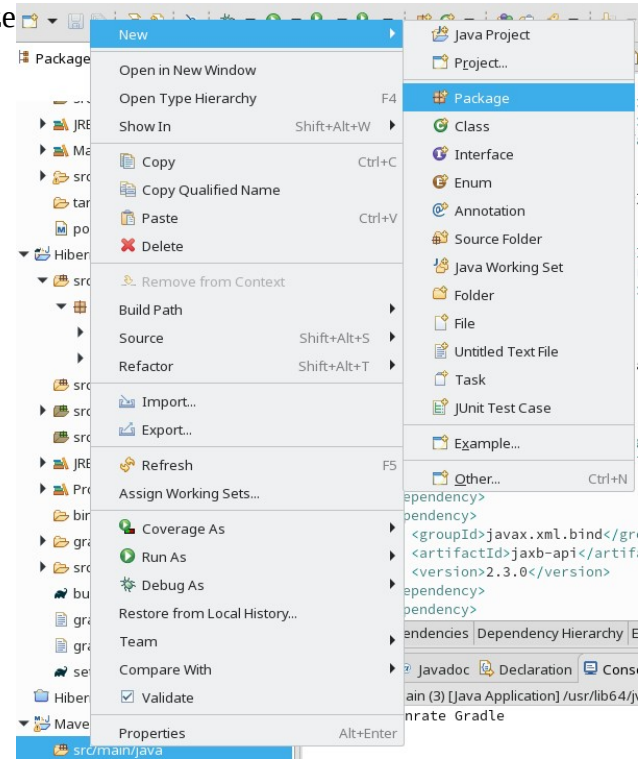
```
  <dependency>
    <groupId>javax.xml.bind</groupId>
    <artifactId>jaxb-api</artifactId>
    <version>2.3.0</version>
  </dependency>
  <dependency>
    <groupId>com.sun.xml.bind</groupId>
    <artifactId>jaxb-core</artifactId>
    <version>2.3.0</version>
  </dependency>
  <dependency>
     <groupId>com.sun.xml.bind</groupId>
     <artifactId>jaxb-impl</artifactId>
     <version>2.3.0</version>
  </dependency>
</dependencies>
```

UWAGA! Wersje podane powyżej są aktualne na dzień 23.02.2020, dla wersji OpenJDK 11/13. W późniejszym czasie mogą pojawić się nowe wersje bibliotek.

Cały plik pom.xml, będący plikiem konfiguracji projektu, może wyglądać jak na zrzucie:

```
<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>hibernate.maven</groupId>
  <artifactId>MavenHibernate</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <name>Maven Hibernate</name>
  <description>Przykładowy plik projektu Maven, tworzonego dla projektu wykorzystującego Hibernate.</description>
  <dependencies>
    <dependency>
      <groupId>org.hibernate</groupId>
      <artifactId>hibernate-core</artifactId>
      <version>5.2.6.Final</version>
    </dependency>
    <dependency>
      <groupId>mysql</groupId>
      <artifactId>mysql-connector-java</artifactId>
      <version>8.0.13</version>
    </dependency>
    <dependency>
        <groupId>javax.activation</groupId>
        <artifactId>activation</artifactId>
        <version>1.1.1</version>
    </dependency>
    <dependency>
        <groupId>javax.xml.bind</groupId>
        <artifactId>jaxb-api</artifactId>
        <version>2.3.0</version>
    </dependency>
    <dependency>
        <groupId>com.sun.xml.bind</groupId>
        <artifactId>jaxb-core</artifactId>
        <version>2.3.0</version>
    </dependency>
    <dependency>
       <groupId>com.sun.xml.bind</groupId>
       <artifactId>jaxb-impl</artifactId>
       <version>2.3.0</version>
    </dependency>
  </dependencies>
</project>
```

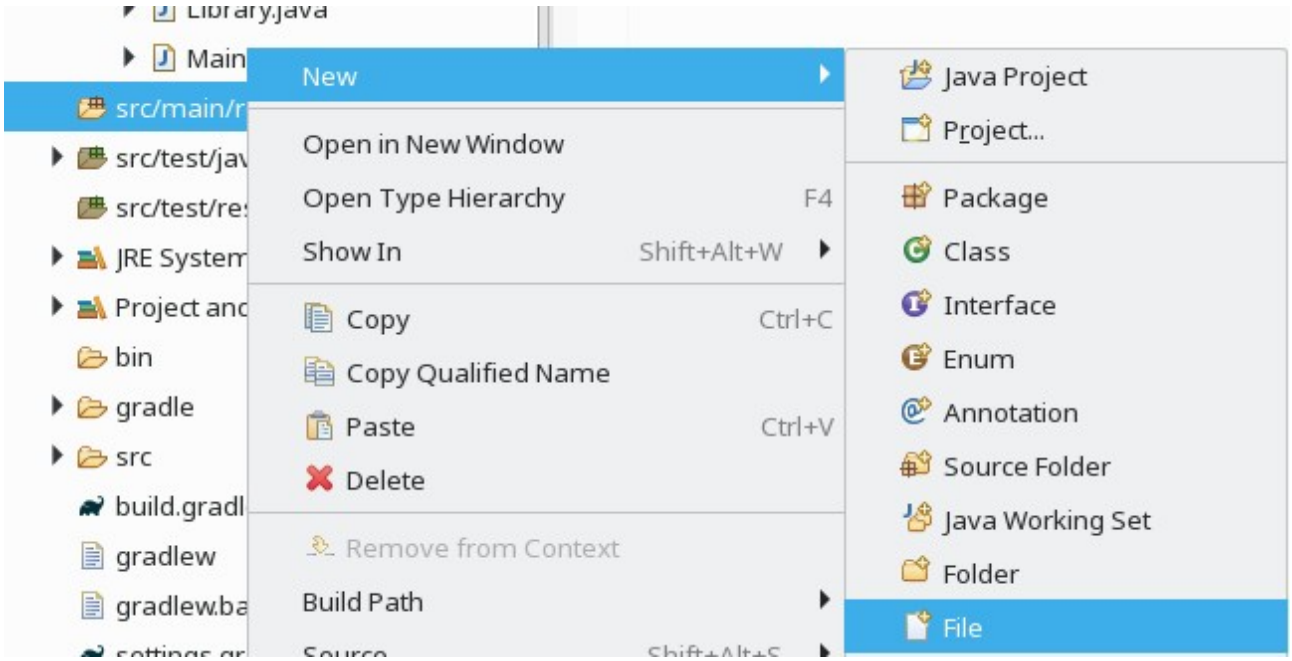
W przypadku projektu Maven musimy jeszcze w katalogu src/main/java dołożyć paczkę grupy, którą podaliśmy jako group ID projektu:

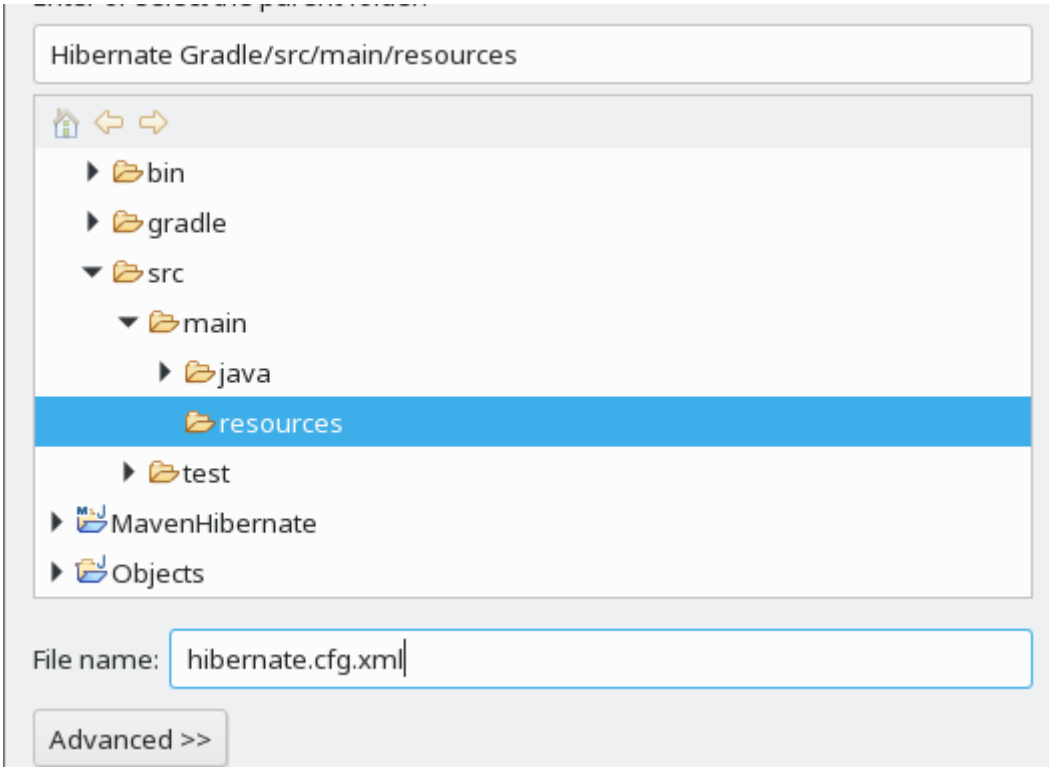


W innym wypadku projekt może nie być poprawnie kompilowany.

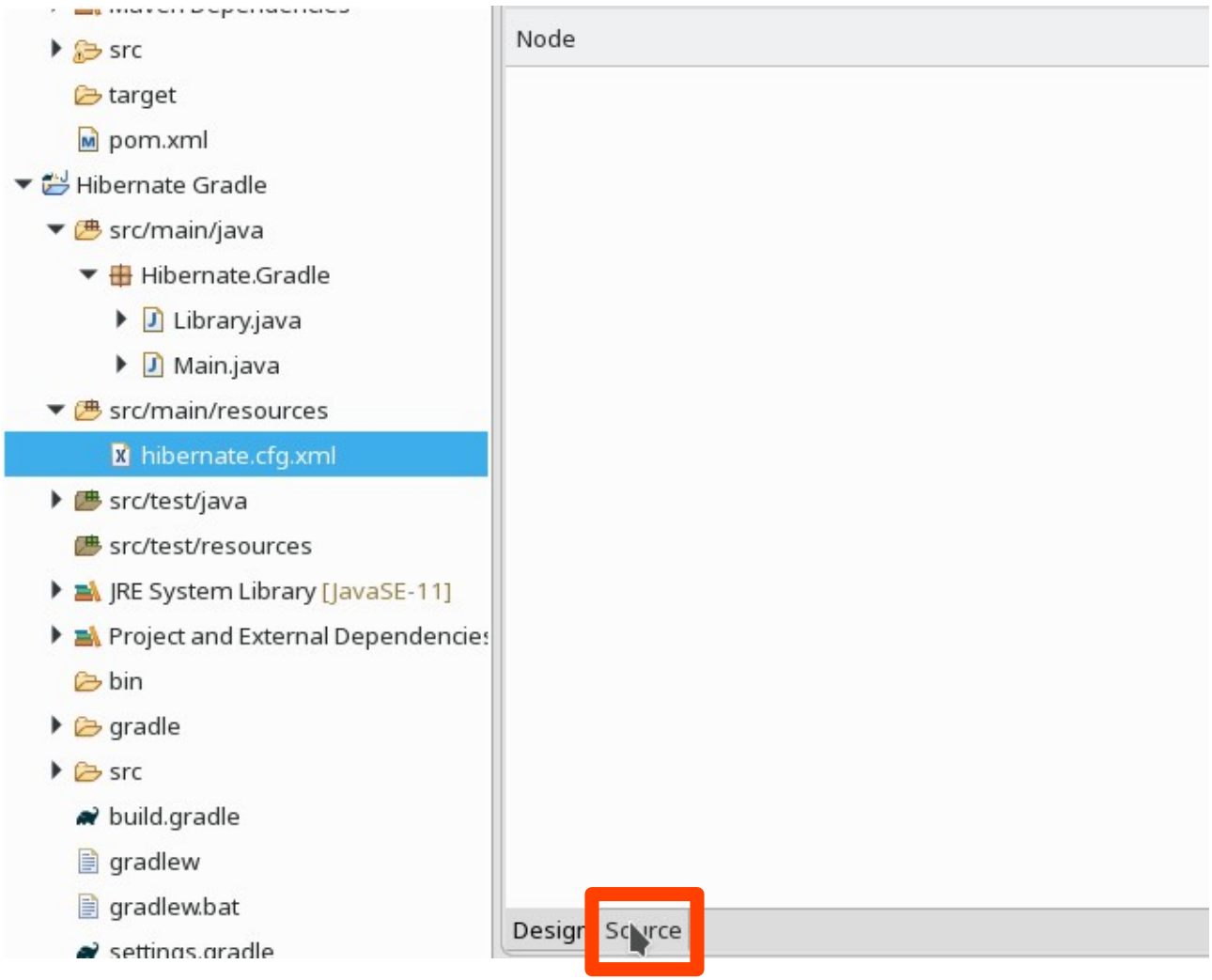
2. Utworzenie konfiguracji projektu Hibernate.

Niezależnie od tego, z którego narzędzia skorzystaliśmy, możemy stworzyć nowy plik konfigurujący nasze połączenie z bazą danych. Plik ten posiada najczęściej nazwę hibernate.cfg.xml i powinien zostać umieszczony w folderze src/main/resources





Po otwarciu pliku przechodzimy do pliku i upewniamy się, że mamy podgląd na jego kod:



Plik musi zawierać pewną minimalną ilość wartości konfiguracyjnych. Najprostsza konfiguracja przedstawiona zostanie poniżej:

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE hibernate-configuration PUBLIC
"-//Hibernate/Hibernate Configuration DTD 4.0//EN"
"http://hibernate.net/hibernate-configuration-4.0.dtd">
<hibernate-configuration>
   <session-factory>
      <property name="hibernate.connection.driver_class">com.mysql.cj.jdbc.Driver</property>
      <property name="hibernate.connection.url">jdbc:mysql://localhost:3306/baza</property>
      <property name="hibernate.connection.password">test</property>
      <property name="hibernate.connection.username">test</property>
      <property name="hibernate.dialect">org.hibernate.dialect.MySQLDialect</property>
      <property name="show_sql">true</property>
   </session-factory>
</hibernate-configuration>
```

Do sekcji hibernate-configration mamy do czynienia ze standardową składnią pliku XML. Określamy jego wersję, typ dokumentu oraz tzw. słownik znaczników XML (w jaki sposób mają być one interpretowane).

W sekcji <hibernate-configuration></hibernate-configuration> zawarta jest konfiguracja naszego połączenia z bazą danych. Obecnie plik posiada tylko jedną sekcję – session-factory (współczynniki/elementy sesji połączeniowej).

Poszczególne właściwości zawarte w tejże sekcji:

- hibernate.connection.driver_class – określa z jakiego sterownika korzystamy podczas połączenia naszej aplikacji. Wybrany tutaj pozwala na połączenie do serwera baz MySQL/MariaDB (dla innych serwerów będzie to inny sterownik)

- hibernate.connection.url – podajemy tutaj jednoznaczny adres zasobu (URL), pod którym dostępna będzie nasza baza; w tym wypadku nasze połączenie wskazuje na localhost (adres 127.0.01/::1 w IPv6), na port 3306 (domyślny port serwera MySQL, pod którym przyjmuje on połączenia od klientów), na którym to serwerze dostępna jest baza danych o nazwie `baza`

- hibernate.connection.password – w tym miejscu podajemy hasło do połączenia; ze względu na otwarty charakter hasła dobrym rozwiązaniem byłoby utworzenie użytkownika posiadającego odpowiednie uprawnienia tylko do określonej bazy danych (z ograniczonymi, bądź żadnymi prawami do pozostałych elementów serwera baz).

- hibernate.connection.username – nazwa użytkownika w bazie danych, który będzie wykorzystywany do połączenia (dla którego podaliśmy wcześniej hasło)

- hibernate.dialect- wskazujemy Hibernate, w jakim „narzeczu” ma porozumiewać się z serwerem SQL; w naszym wypadku będzie to MYSQLDialect

- show_sql – opcja pozwala na pokazywanie zapytań, jakie generuje hibernate do bazy danych. Parametr niezwykle przydatny podczas pisania aplikacji (można zobaczyć czemu coś nie działa jak należy), w przypadku wersji produkcyjnych można go pominąć.

Dodatkowo dobrze jest dołączyć następującą opcję:

<property name="hbm2ddl.auto">update</property>

Hibernate podejmuje dzięki temu decyzję dotyczącą schematu działania z bazą danych, dla języka DDL (Data Definition Language). Domyślnie raz utworzony schemat zostaje zapisany i wykorzystywany do pracy z przesyłem danych (domyślna opcja to validate).

Wszystkie opcje właściwości:

- validate – schemat jest sprawdzany pod kątem poprawności

- update – podczas tworzenia sesji z bazą następuje sprawdzenie i zaktualizowanie schematu

- create – każdorazowo tworzony jest nowy schemat

- create-drop – jak poprzednio, jednak w przypadku zatrzymania aplikacji schemat jest niszczony

- none – nic nie rób ze schematem

Pozostałe parametry i ich wykorzystanie można znaleźć w oficjalnej dokumentacji Hibernate, odnośniki do niej dostępne są w źródłach.

3. Konfiguracja mapowania tabel do klas.

Załóżmy, że posiadamy definicję takiej oto tabeli:

```
CREATE TABLE `osoby` (
  `id_osoba` bigint(20) UNSIGNED NOT NULL,
  `imie` varchar(30) NOT NULL,
```

```
  `nazwisko` varchar(40) NOT NULL,
  `nick` varchar(20) NOT NULL,
  `data_urodzenia` date NOT NULL,
  `emial` varchar(100) NOT NULL,
  `telefon` varchar(12) NOT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;
```

Klasa, która będzie w stanie odwzorować ją w kodzie Java przyjmie następującą postać:

```
package Hibernate.Gradle;

public class Osoby {
	private long id;
	private String name;
	private String surname;
	private String nick;
	private String birthdate;
	private String email;
	private String telephone;

	public void setId(long id) {
		this.id=id;
	}
	public long getId() {
		return id;
	}
	public void setName(String name) {
		this.name=name;
	}
	public String getName() {
		return name;
	}
	public void setSurname(String surname) {
		this.surname=surname;
	}
	public String getSurname() {
		return surname;
	}
```

```
	public void setBirthdate(String birthdate) {
		this.birthdate=birthdate;
	}
	public String getBirthdate() {
		return birthdate;
	}
	public void setNick(String nick) {
		this.nick=nick;
	}
	public String getNick() {
		return nick;
	}
	public void setEmail(String email) {
		this.email=email;
	}
	public String getEmail() {
		return email;
	}
	public void setTelephone(String telephone) {
		this.telephone=telephone;
	}
	public String getTelephone() {
		return telephone;
	}
}
```

Przy wszystkim należy pamiętać, że nazwy dla funkcji MUSZĄ odpowiadać nazwom zmiennych zadeklarowanych jako pola klasy! Inaczej nie uda się poprawnie zmapować klasy do tabeli! Należy zauważyć, że dopuszcza się nazywanie funkcji wedle konwencji:

```
private String email;
public void setEmail(String email) { this.email=email; }
public String getEmail() { return email; }
```

Rozwiązanie to poprawia czytelność nazw metod get/set.

Mając zdefiniowane tabelę oraz klasę, trzeba wskazać rolę przyporządkowania jednego do drugiego poprzez Hibernate. W tym celu można utworzyć plik o dowolnej nazwie zawierający definicję relacji pomiędzy dwoma obiektami.

Wedle konwencji plik powinien zawierać nazwę klasy, wskazanie iż definiuje relację Hibernate oraz rozszerzenie języka znaczników xml (jako, że w tymże języku następuje definicja).

W naszym wypadku plik przyjmie nazwę

Osoby.hbm.xml

Jego kod będzie wyglądał następująco:

```
<?xml version = "1.0" encoding = "utf-8"?>
<!DOCTYPE hibernate-mapping PUBLIC
"-//Hibernate/Hibernate Mapping DTD//EN"
"http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
  <class name = "Hibernate.Gradle.Osoby" table = "osoby">
    <meta attribute = "class-description">
      This class contains the employee detail.
    </meta>
    <id name = "id" type = "long" column = "id_osoba">
      <generator class="native"/>
    </id>
    <property name = "name" column = "imie" type = "string"/>
    <property name = "surname" column = "nazwisko" type = "string"/>
    <property name = "nick" column = "nick" type = "string"/>
    <property name = "birthdate" column = "data_urodzenia" type = "string"/>
    <property name = "email" column = "emial" type = "string"/>
    <property name = "telephone" column = "telefon" type = "string"/>
  </class>
</hibernate-mapping>
```

Pierwsze linie, do <hibernate-mapping>, wyglądają podobnie jak to miało miejsce w pliku konfiguracji opisywanej technologii.

Znacznik class definiuje przepisanie klasy naszego programu w języku Java do nazwy tabeli w bazie, z którą będziemy przeprowadzać transakcje. Tutaj należy pamiętać, że NAZWA KLASY MUSI ZAWIERAĆ PEŁNĄ ŚCIEŻKĘ PACZKI! Bez tej ścieżki będziemy otrzymywać komunikat o braku powiązania encji! Za polem nazwy znajduje się pole tabeli, gdzie podajemy jej pełną nazwę (zachowując wielkość liter).

Znacznik id zarezerwowany jest dla pola identyfikującego jednoznacznie każdy wiersz zapisanych danych w naszej tabeli. Tutaj również jako nazwę podajemy nazwę pola w naszej tabeli, zaś pod atrybutem kolumny podajemy nazwę pola w tabeli (wielkość znaków musi być 1:1). Dodatkowo wskazujemy typ danych – w naszym wypadku jest to wartość long (w MySQL odpowiada za BigInt). Znacznik generator pozwala ustalić, w jaki sposób nadawane są nowe wartości identyfikatora.

W kolejnych znacznikach właściwości (property) podawane są nazwy zmiennych w klasie i ich odpowiedników w bazie. Każda z właściwości musi posiadać swój typ. Należy zauważyć, że typ daty w bazie danych (data_urodzenia) w przypadku mapowania posiada typ string, nie date (typ datowy w Javie znacznie różni się od typu w bazie danych).

Ostatnim etapem powiązania jest wskazanie dla Hibernate ścieżki do pliku hbm.xml. W tym wypadku do pliku hibernate.cfg.xml należy dodać dodatkowy wpis:

<mapping resource="Osoby.hbm.xml"></mapping>

lub

<mapping resource="Osoby.hbm.xml"/>

Przy czym trzeba pamiętać, że ścieżka do pliku może być względna lub bezwzględna. Powyższy wpis będzie poprawny, jeżeli plik hbm.xml będzie znajdował się dokładnie w tym samym katalogu co cfg.xml.

Poniżej zaprezentowane zostaną zrzuty ekranowe dokumentujące ułożenie plików w projekcie oraz ich strukturę celem lepszego zapoznania się z konfiguracją projektu:

Plik hibernate.cfg.xml:

build.gradle | Library.java | Main.java | *hibernate.cfg.xml | Osoby.java | Role.java | RoleMap.java

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE hibernate-configuration PUBLIC
"-//Hibernate/Hibernate Configuration DTD 4.0//EN"
"http://hibernate.net/hibernate-configuration-4.0.dtd">
<hibernate-configuration>
    <session-factory>
        <property name="hibernate.connection.driver_class">com.mysql.cj.jdbc.Driver</property>
        <property name="hibernate.connection.url">jdbc:mysql://192.168.1.220:3306/esport</property>
        <property name="hibernate.connection.password">test</property>
        <property name="hibernate.connection.username">test</property>
        <!--  <property name="hibernate.connection.usessl">false</property>-->
        <property name="hibernate.dialect">org.hibernate.dialect.MySQLDialect</property>
        <property name="show_sql">true</property>
        <property name="hbm2ddl.auto">update</property>

        <mapping resource="Osoby.hbm.xml"></mapping>
        <mapping resource="Role.hbm.xml"></mapping>
    </session-factory>
</hibernate-configuration>
```

Plik Osoby.hbm.xml:

build.gradle | Library.java | Main.java | hibernate.cfg.xml | Osoby.java | Role.java | RoleMap.java | Osoby.hbm.xml

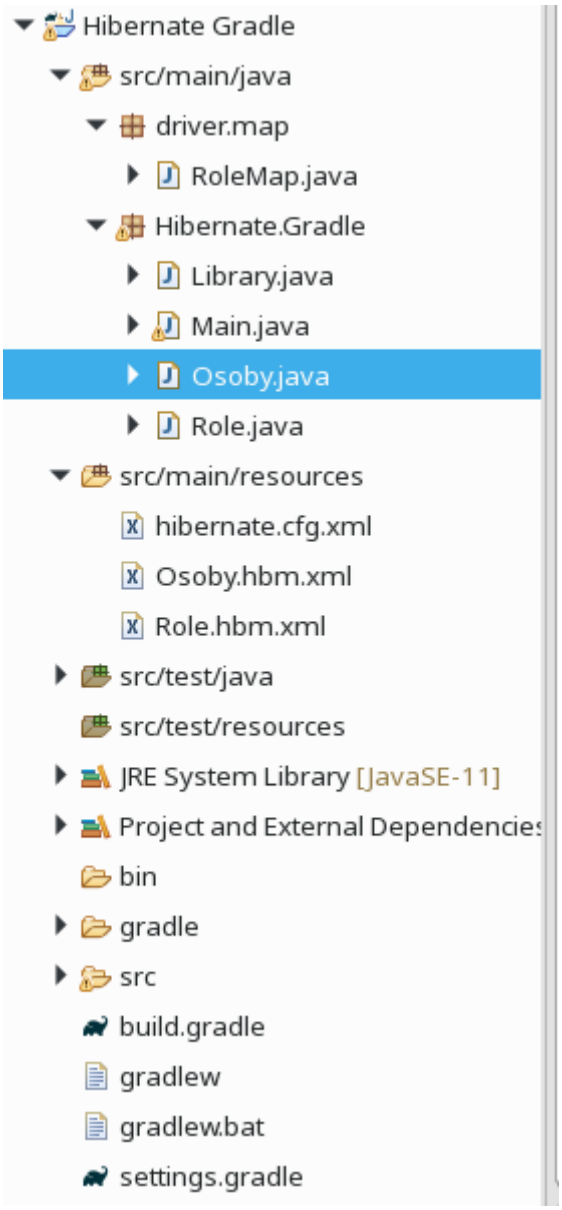
```
<?xml version = "1.0" encoding = "utf-8"?>
<!DOCTYPE hibernate-mapping PUBLIC
"-//Hibernate/Hibernate Mapping DTD//EN"
"http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">

<hibernate-mapping>
   <class name = "Hibernate.Gradle.Osoby" table = "osoby">

      <meta attribute = "class-description">
         This class contains the employee detail.
      </meta>

      <id name = "id" type = "long" column = "id_osoba">
         <generator class="native"/>
      </id>

      <property name = "name" column = "imie" type = "string"/>
      <property name = "surname" column = "nazwisko" type = "string"/>
      <property name = "nick" column = "nick" type = "string"/>
      <property name = "birthdate" column = "data_urodzenia" type = "string"/>
      <property name = "email" column = "emial" type = "string"/>
      <property name = "telephone" column = "telefon" type = "string"/>
   </class>
</hibernate-mapping>
```

Plik Osoby.java (na zrzucie brak ostatniego nawiasu zamykającego całą klasę):

build.gradle | Library.java | Main.java | hibernate.cfg.xml | *Osoby.java

```java
package Hibernate.Gradle;
public class Osoby {
    private long id;
    private String name;
    private String surname;
    private String nick;
    private String birthdate;
    private String email;
    private String telephone;
    public void setId(long id) {
        this.id=id;
    }
    public long getId() {
        return id;
    }
    public void setName(String name) {
        this.name=name;
    }
    public String getName() {
        return name;
    }
    public void setSurname(String surname) {
        this.surname=surname;
    }
    public String getSurname() {
        return surname;
    }
    public void setBirthdate(String birthdate) {
        this.birthdate=birthdate;
    }
    public String getBirthdate() {
        return birthdate;
    }
    public void setNick(String nick) {
        this.nick=nick;
    }
    public String getNick() {
        return nick;
    }
    public void setEmail(String email) {
        this.email=email;
    }
    public String getEmail() {
        return email;
    }
    public void setTelephone(String telephone) {
        this.telephone=telephone;
    }
```

Drzewo katalogów naszego projektu, pokazujące rozlokowanie plików:



Powyższa konfiguracja umożliwia podłączenie się i operacje na naszej bazie danych.

4. Uruchomienie pierwszego przykładu, przygotowanie odpowiednich zmiennych i metod.

Mając już przygotowaną konfigurację oraz pierwszą klasę można utworzyć szkielet aplikacji bazodanowej. Na razie aplikacja na sztywno będzie dodawała jeden wiersz danych i będzie odczytywać pojedyncze informacje z bazy, a także wszystkie dane z bazy do jednej listy.

Wszystko należy zacząć od utworzenia pliku klasy z metodą main(). W tym celu możemy utworzyć klasę o nazwie Main.java. W celu sprawdzenia działania aplikacji zastosujemy prosty kod, który umożliwi nam zapis nowego wiersza do zmapowanej tabeli.

Ponieważ klasa należeć będzie do paczki, podajemy jej nazwę:

```
package Hibernate.Gradle;
```

Następnie definiujemy nazwę naszej klasy:

```
public class Main {
//tutaj będzie kod naszej aplikacji
}
```

Jak widać niczym nie różni się od definicji innych klas. W jej wnętrzu umieszczamy definicję metody main():

```
public static void main(String[] args) {
        //tutaj będzie kod połączenia do bazy
}
```

Wewnątrz niej umieszczamy kod połączenia:

```
SessionFactory polaczenieHibernate;                                          //1
final StandardServiceRegistry registry = new StandardServiceRegistryBuilder()  //2
                                                .configure()                 //3
                                                .build();                    //4
try {
        polaczenieHibernate = new MetadataSources(registry)                  //5
                                .BuildMetadata()                             //6
                                .buildSessionFactory();                      //7

} catch (Exception ex) {
        StandardServiceRegistryBuilder.destroy(registry);                    //8
        throw new RuntimeException(ex);                                      //9
}
```

W pierwszej linii tworzymy zmienną, która będzie przechowywać dla nas uchwyt do połączenia Hibernate.

W linii drugiej tworzymy zmienną zawierającą konfigurację Hibernate. Tworzona jest z pliku hibernate.cfg.xml, którego zawartość wczytuje metoda z linii trzeciej (metoda automatycznie szuka tego pliku w katalogu resources; plik można również podać jako parametr – jeżeli posiada inną nazwę lub znajduje się w innej lokalizacji).

Czwarta linia powoduje zbudowanie konfiguracji z wczytanego pliku przez linię trzecią.

Dalszy kod musi wykonać się w klauzuli try...catch (obowiązkowa do obsługi wyjątków wyrzucanych przez metodę tworzenia połączenia).

Piąta linia przypisuje do naszej zmiennej-uchwytu wartość sesji łączącej ze wskazanym serwerem baz danych do konkretnej bazy (w naszym wypadku o nazwie baza). Linia szósta i siódma wywołuje odpowiednie metody tworzące niezbędne elementy połączenia.

Linia ósma odpowiada za zniszczenie wartości konfiguracji połączenia w przypadku gdyby utworzenie połączenia nie było możliwe (np. brak bazy danych, niewłaściwe wartości połączenia, złe hasło itp.).

Ponieważ może się zdarzyć, że niszczenie zmiennej registry będzie próbowało niszczyć zmienną typu null (np. został podany niewłaściwy plik konfiguracyjny/brak pliku) linia 9 wywoła pojawienie się kolejnego wyjątku do obsłużenia przez nasz program. Dzięki temu będziemy mogli ze szczegółami dowiedzieć się jaki błąd w naszym programie wystąpił i w jaki sposób go wyeliminować.

Jeżeli klauzula try przeszła bez problemów, możemy wykonać zapis do naszej bazy danych. W tym celu dodamy następujące linie do naszego kodu:

```
Session s = polaczenieHibernate.openSession();      //1
s.beginTransaction();                               //2
Osoby o = new Osoby();
o.setName("Marianna");
o.setSurname("Wilczkiewicz");
o.setNick("pantera");
o.setEmail("niemam@nazwa.pl");
o.setDate("1995-10-25");
o.setTelephone("777-888-999");
s.save(o);                                          //3
s.getTransaction().commit();                        //4
s.close();                                          //5
```

Najważniejsze linie kodu zostały ponumerowane.

Pierwsza ustanawia utworzenie nowej sesji połączenia z bazą danych. O ile polaczenieHibernate to tylko „luźne” powiązanie do naszej bazy, o tyle zmienna s nawiązuje je celem podjęcia akcji na naszej bazie.

Druga linia zakłada rozpoczęcie transakcji z bazą danych. Do chwili, kiedy nie nastąpi zatwierdzenie zmian (linia 4) w bazie nie pojawi się nowy zapis. Zabezpiecza to bazę przed dodaniem niepełnej/niewłaściwej treści.

Trzecie linia dodaje do tabeli „na brudno” nową wartość, jaka zapisana została do naszego obiektu utworzonego z klasy Osoba.

Piąta linia powoduje zamknięcie sesji. Połączenie do bazy jest nadal aktywne, jednak możliwe stają się inne operacje, np. odczytu bądź zapisu do bazy przez inne aplikacje bazodanowe związane z tą bazą.

W ostatniej fazie zamykamy połączenie z bazą:

```
polaczenieHibernate.close();
```

Całość kodu naszej aplikacji będzie się prezentować następująco:

build.gradle | Library.java | *Main.java | hibernate.cfg.x | Role.java | RoleMap.java | Osoby.hbm.xml

```java
package Hibernate.Gradle;

import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.boot.MetadataSources;
import org.hibernate.boot.registry.StandardServiceRegistry;
import org.hibernate.boot.registry.StandardServiceRegistryBuilder;

public class Main {
    public static void main(String[] args) {
        SessionFactory polaczenieHibernate;
        final StandardServiceRegistry registry = new StandardServiceRegistryBuilder()
                .configure() // configures settings from hibernate.cfg.xml
                .build();
        try {
            polaczenieHibernate = new MetadataSources(registry).buildMetadata().buildSessionFactory();
            Session s = polaczenieHibernate.openSession();
            s.beginTransaction();
            Osoby o = new Osoby();
            o.setName("Marianna");
            o.setSurname("Wilczkiewicz");
            o.setNick("pantera");
            o.setEmail("niemam@nazwa.pl");
            o.setBirthdate("1995-10-25");
            o.setTelephone("777-888-999");
            s.save(o);
            s.getTransaction().commit();
            s.close();
            polaczenieHibernate.close();
        } catch (Exception ex) {
            StandardServiceRegistryBuilder.destroy(registry);
        }
    }
}
```

Proszę zauważyć, że w tym układzie zapis do bazy również objęty jest klauzulą try...catch; w innym wypadku moglibyśmy próbować otwierać sesję z niezainicjowanego połączenia.

Wpisów import możemy dokonać ręcznie lub zostaną dodane przez Eclipse/kliknięcie przez nas rozwiązywania zależności. Należy pamiętać, żeby wykorzystać paczki org.hibernate (mogą się pojawić wpisy z innych paczek, które nie są kompatybilne).

Program w postaci przedstawionej powyżej ma oczywiście mało sensu z kilku powodów. Oto one:

- brak możliwości odczytu danych z tabeli

- brak możliwości aktualizacji danych

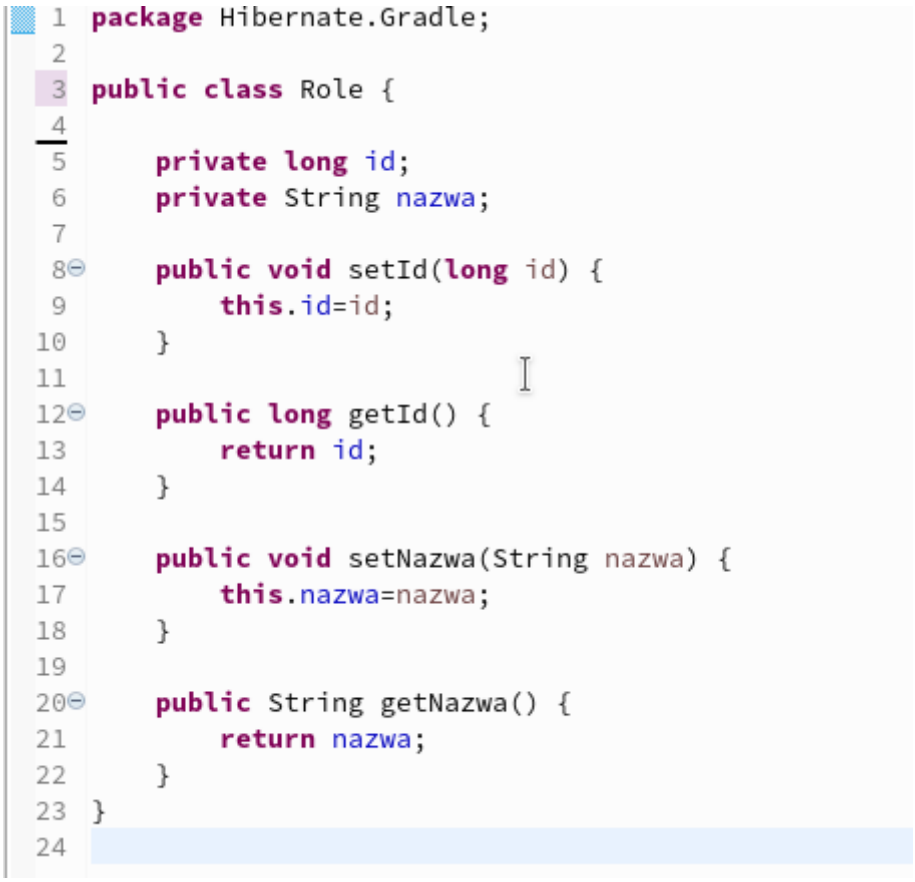
- brak możliwości usuwania danych

Do każdej operacji musielibyśmy na nowo tworzyć kod z klauzuli try i wykonywać tylko jedną, określoną operację. Rozwiązanie to byłoby mało elastyczne zważywszy na fakt, że możemy posiadać znacznie więcej tabel niż jedna. To powodowałoby konieczność pisania ogromnej ilości kodu, który byłby powielany.

Dlatego też, na podstawie innej, mniejszej tabeli, zostanie pokazane rozwiązanie znacznie bardziej elastyczne, pozwalające na elastyczny dostęp do poszczególnych wpisów w tabeli.

Po pierwsze tworzymy plik klasy zawierający pola takie same jak mapowana tabela:

```
package Hibernate.Gradle;
public class Role {
	private long id;
	private String nazwa;
	public void setId(long id) {
		this.id=id;
	}
	public long getId() {
		return id;
	}
	public void setNazwa(String nazwa) {
		this.nazwa=nazwa;
	}
	public String getNazwa() {
		return nazwa;
	}
}
```



```
1  package Hibernate.Gradle;
2
3  public class Role {
4
5      private long id;
6      private String nazwa;
7
8      public void setId(long id) {
9          this.id=id;
10     }
11
12     public long getId() {
13         return id;
14     }
15
16     public void setNazwa(String nazwa) {
17         this.nazwa=nazwa;
18     }
19
20     public String getNazwa() {
21         return nazwa;
22     }
23 }
24
```

Kod tabeli, na podstawie której utworzona została klasa:

```
CREATE TABLE `role` (
`id_rola` SERIAL,
`nazwa` VARCHAR(30))
```

Następnie tworzymy, jak poprzednio, plik hbm.xml:

```
<?xml version = "1.0" encoding = "utf-8"?>
<!DOCTYPE hibernate-mapping PUBLIC
"-//Hibernate/Hibernate Mapping DTD//EN"
"http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
  <class name = "Hibernate.Gradle.Role" table = "role">
    <meta attribute = "class-description">
      This class contains the employee detail.
    </meta>
    <id name = "id" type = "long" column = "id_rola">
      <generator class="native"/>
    </id>
    <property name = "nazwa" column = "nazwa" type = "string"/>
  </class>
</hibernate-mapping>
```

```
1  <?xml version = "1.0" encoding = "utf-8"?>
2  <!DOCTYPE hibernate-mapping PUBLIC
3  "-//Hibernate/Hibernate Mapping DTD//EN"
4  "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">
5
6  <hibernate-mapping>
7     <class name = "Hibernate.Gradle.Role" table = "role">
8
9        <meta attribute = "class-description">
10          This class contains the employee detail.
11       </meta>
12
13       <id name = "id" type = "long" column = "id_rola">
14          <generator class="native"/>
15       </id>
16
17       <property name = "nazwa" column = "nazwa" type = "string"/>
18
19    </class>
20 </hibernate-mapping>
```

Teraz tworzymy nowy plik np. o nazwie RoleMap.java. Dobrze byłoby, gdyby plik był w innej paczce niż dotychczasowe klasy, by łatwiej rozróżnić klasy modelu danych (odwzorowanie danych po stronie bazy i programu) od ich sterownika (gdzie będą wykonywane operacje przesyłania danych w obie strony). W poniższym przykładzie nowa klasa została umieszczona w paczce driver.map. Sam plik zaczynamy od deklaracji klasy:

```
public class RoleMap {
    //tutaj będzie kod sterownika
}
```

Tworzymy zmienną Session; będzie ona przechowywać dostęp do sesji połączenia z tabelą podczas wykonywania operacji zapisu/odczytu/aktualizacji/usuwania danych. Sesja ta będzie zestawiana dla każdej z operacji osobno.

```
private Session session;
```

Celem upewnienia się, że będziemy posiadali zestawioną sesję, utworzenie klasy wymusi na nas przekazanie odpowiedniej wartości:

```
    public RoleMap(Session session) {
        this.session=session;
    }
```

Zapis do bazy zostanie zamknięty w następującej metodzie:

```
    public void save(String nazwa) {
        session.beginTransaction();
        Role rola = new Role();
        rola.setNazwa(nazwa);
        session.save(rola);
        session.getTransaction().commit();
    }
```

Tym sposobem możemy wywoływać dowolną ilość razy zapis do naszej tabeli. Należy zwrócić uwagę, że metoda nie kończy sesji, a jedynie zatwierdza wprowadzone zmiany. Jeżeli ta linia nie pojawiłaby się w metodzie to nasze dane nie zostałyby dodane do bazy (a jedynie zapisane w jej roboczej wersji).

Wybranie danych z tabeli po jednoznacznym polu jest stosunkowo proste:

```
public Role read(long id) {
        Role rola = session.get(Role.class, id);
        return rola;
    }
```

Ta metoda nie rozpoczyna transakcji, tak samo jak jej nie zatwierdza. Dzieje się tak dlatego, że bazy danych nie wymagają mechanizmów chwilowego blokowania zasobów przy odczycie (wręcz wiele połączeń może naraz czytać dane zgromadzone w tabelach bazy danych).

Nieco bardziej skomplikowanie będzie wyglądać sytuacja w chwili, gdy zechcemy wyciągnąć dane po zmiennych nieunikatowych/nie posiadających jakiegokolwiek indeksu. W tym wypadku zajdzie

potrzeba skorzystania ze standardowego zapytania SQL, zgodnego z wybranym przez nas silniku bazodanowym. Poniżej prezentacja metody wyciągającej wiersze z danymi, w których widnieje określona nazwa:

```
public Role read(String nazwa) {
    @SuppressWarnings("unchecked")
            List<Object> role = session.createSQLQuery("SELECT id_rola FROM role WHERE nazwa='"+nazwa+"'").list();
    if (role.isEmpty())
       return new Role();
    Role rola = new Role();
    BigInteger bi = (BigInteger) role.get(0);
    rola.setId(bi.longValue());
    rola.setNazwa(nazwa);
            return rola;
       }
```

Pierwsza linijka wskazuje, by kompilator nie informował nas o ewentualnej możliwości pozostawienia naszej zmiennej role bez jakiegokolwiek wiersza.

Do zmiennej role przypisujemy wynik zapytania pobierającego dla nas zawartości id_rola ze wszystkich wierszy, w których występuje wskazana nazwa.

Ponieważ chcemy wybrać tylko jeden, pierwszy wynik, pobieramy pierwszy wynik z listy. Ponieważ mamy do czynienia z liczbą typu BigInteger, musimy przekonwertować ją do wartości long (bi.longValue()).

Na koniec zwracamy obiekt Role z wypełnionymi danymi.

Aktualizacja danych przebiega w bardziej „złożony” sposób:

```
public boolean update(String staraNazwa, String nowaNazwa) {
            return update(staraNazwa, nowaNazwa, -1);
    }

    boolean update(String staraNazwa, String nowaNazwa, long id) {
            Role rola = (id!=-1) ? read(id): read(staraNazwa);
            if (rola == null)
                    return false;
            System.out.println(rola.getNazwa());
            rola.setNazwa(nowaNazwa);
            session.beginTransaction();
            session.update(rola);
            session.getTransaction().commit();
            return true;
    }
```

Powyżej widzimy dwie metody. Jedna z nich przyjmuje dwa argumenty – obecnie występującą nazwę w tabeli oraz nazwę, jaką pierwotna wartość ma zostać zastąpiona.

Możliwe jest też podanie trzeciego parametru, identyfikatora wiersza z nazwą. Wtedy też wywoła się druga z przedstawionych metod (w przypadku pominięcia trzeciego parametru przyjmie on wartość -1).

W pierwszej linii właściwej metody aktualizacyjnej tworzymy obiekt na podstawie podanego id/wyciągniętego id na podstawie podanej starej nazwy.

Jeżeli obiekt o podanym id/nazwie nie istnieje – zostanie zwrócona wartość false (nie dokonano modyfikacji w jakimkolwiek wierszu). W przeciwnym wypadku zostanie wywołana metoda update i zwrócona zostanie wartość true.

Analogicznie będzie wyglądała sytuacja z kasowaniem danych:

```
public boolean delete(String nazwa) {
		return delete(nazwa,-1);
	}

	public boolean delete(long id) {
		return delete("",id);
	}

	boolean delete(String nazwa, long id) {
		Role rola = (id!=-1) ? read(id): read(nazwa);
		if (rola == null)
			return false;
		session.beginTransaction();
		session.delete(rola);
		session.getTransaction().commit();
		return true;
	}
```

W tym miejscu występują dwa przeciążenia metod. Programista może wywołać kasowanie po nazwie lub po identyfikatorze. Ostatnia metoda nie jest jawna (nie można jej wywołać z obiektu). To w niej pobierane są brakujące dane naszego obiektu Rola i następuje usunięcie wiersza, który będzie zawierał dokładnie te dane.

Nasza klasa będzie zawierała jeszcze jedną metodę, pozwalającą na pobranie wszystkich wierszy danych z mapowanej tabeli:

```
public List<Role> getAll() {
	CriteriaBuilder builder = session.getCriteriaBuilder();
	CriteriaQuery<Role> criteria = builder.createQuery(Role.class);
    criteria.from(Role.class);
    List<Role> role = session.createQuery(criteria).getResultList();
		return role; }
```

W pierwszej linijce tworzymy zmienną budowy kryteriów zapytania.

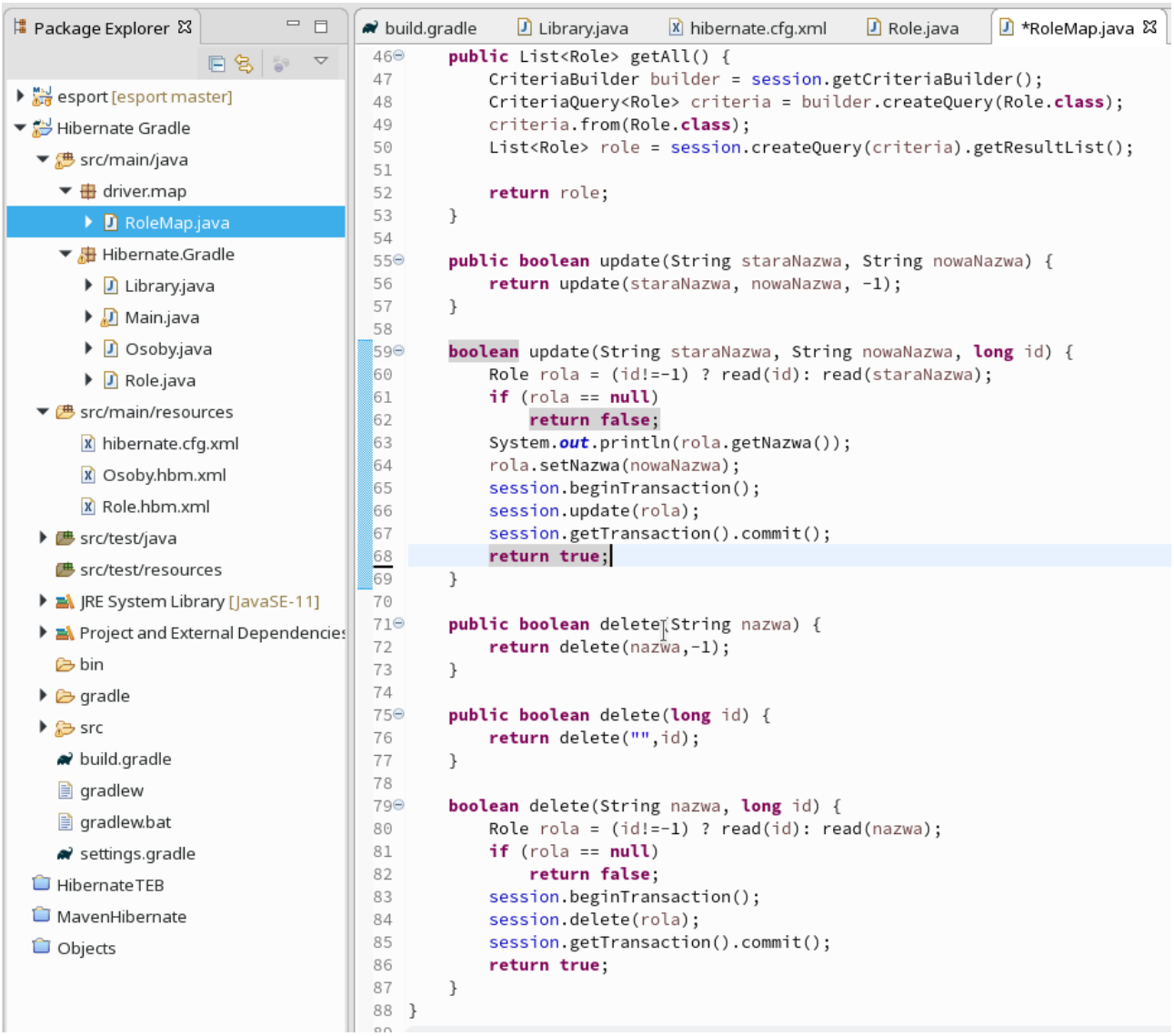
W linii numer dwa tworzymy zapytanie na podstawie całej klasy (wszystkie pola).

Linia trzy wykonuje pobranie danych na podstawie kryteriów.

Linia czwarta wykonuje zapytanie i zapisuje wszystkie wyniki w liście.

Całość kodu klasy będzie się prezentować następująco:

```java
package driver.map;

import java.math.BigInteger;
import java.util.List;

import javax.persistence.criteria.CriteriaBuilder;
import javax.persistence.criteria.CriteriaQuery;

import Hibernate.Gradle.Role;

import org.hibernate.Session;

public class RoleMap {

    private Session session;

    public RoleMap(Session session) {
        this.session=session;
    }

    public void save(String nazwa) {
        session.beginTransaction();
        Role rola = new Role();
        rola.setNazwa(nazwa);
        session.save(rola);
        session.getTransaction().commit();
    }

    public Role read(String nazwa) {
        @SuppressWarnings("unchecked")
        List<Object> role = session.createSQLQuery("SELECT id_rola FROM role WHERE nazwa='"+nazwa+"'").list();
        if (role.isEmpty())
            return new Role();
        Role rola = new Role();
        BigInteger bi = (BigInteger) role.get(0);
        rola.setId(bi.longValue());
        rola.setNazwa(nazwa);
        return rola;
    }

    public Role read(long id) {
        Role rola = session.get(Role.class, id);
        return rola;
    }
```



Wykorzystanie modelu oraz sterownika w pewnym stopniu wymaga od nas przebudowy kody klasy Main. W tej chwili będzie się on prezentował następująco:

```java
public class Main {

    private SessionFactory polaczenieHibernate;

    protected void setup() {

        final StandardServiceRegistry registry = new StandardServiceRegistryBuilder()
                .configure() // configures settings from hibernate.cfg.xml
                .build();
        try {
            polaczenieHibernate = new MetadataSources(registry).buildMetadata().buildSessionFactory();
        } catch (Exception ex) {
            StandardServiceRegistryBuilder.destroy(registry);
            throw new RuntimeException(ex);
        }
    }

    protected void exit() {
        if (polaczenieHibernate != null) {
            polaczenieHibernate.close();
            System.out.println("Zakonczenie");
        }
    }

    protected SessionFactory getPolaczenie() {
        return polaczenieHibernate;
    }

    public static void main(String[] args) {
        System.out.println("Projekt Hibenrate Gradle");
        Main hibernate = new Main();
        hibernate.setup();
        System.out.println("Stanowisko o id 2 " + new RoleMap(hibernate.getPolaczenie().openSession()).read(2).getNazwa());
        System.out.println("Stanowisko id dla Trener " + new RoleMap(hibernate.getPolaczenie().openSession()).read("Trener").getId());
        RoleMap rMapa = new RoleMap(hibernate.getPolaczenie().openSession());
        rMapa.update("Techniczny","Asystent");
        rMapa.save("Sprzątacz");
        rMapa.save("Komentator");
        rMapa.delete("Sprzątacz");
        hibernate.exit();
    }
}
```

Po pierwsze utworzona została zmienna przechowująca stan połączenia z bazą danych i Hibernate:

private SessionFactory polaczenieHibernate;

Dodana została metoda, której jedynym zadaniem jest inicjalizacja powyższej zmiennej:

```
protected void setup() {
                final StandardServiceRegistry registry = new StandardServiceRegistryBuilder()
                    .configure() // configures settings from hibernate.cfg.xml
                    .build();
                try {
                        polaczenieHibernate = new
MetadataSources(registry).buildMetadata().buildSessionFactory();
                } catch (Exception ex) {
                   StandardServiceRegistryBuilder.destroy(registry);
                   throw new RuntimeException(ex);
                }
   }
```

Jak można zauważyć, jest to modyfikacja poprzednio przygotowanej metody main (wyciągnięcie z niej kodu inicjalizacji połączenia).

Utworzona została również metoda kończąca połączenie:

```
        protected void exit() {
        if (polaczenieHibernate != null) {
                polaczenieHibernate.close();
        }
    }
```

Metoda ta sprawdza, czy połączenieHibernate istnieje. Jeżeli tak – zamykamy je. Jeżeli natomiast połączenie nie istnieje – nie ma sensu go zamykać (próba zamknięcia nieistniejącego połączenia wywoła błąd).

W samej metodzie main, zamiast tworzyć ogromy kod try...catch, inicjalizacja połączenia zamknie się w tych dwóch linijkach kodu:

```
Main hibernate = new Main();
hibernate.setup();
```

Pierwsza z nich tworzy zmienną obiektową z klasy Main (tej samej, którą obecnie modyfikujemy). Następnie tworzymy połączenie do bazy.

Testujemy połączenie poprzez wybranie określonych danych z bazy:

```
System.out.println("Stanowisko o id 2 " + new
RoleMap(hibernate.getPolaczenie().openSession()).read(2).getNazwa());
        System.out.println("Stanowisko id dla Trener " + new
RoleMap(hibernate.getPolaczenie().openSession()).read("Trener").getId());
```

Raz wybieramy dane po id, innym razem po nazwie. Zaproponowane rozwiązanie ma jednak wadę – za każdym razem tworzy nowy obiekt, co zabiera cenne zasoby sprzętowe.

Lepszym rozwiązaniem jest utworzenie zmiennej-uchwytu do sterownika:

```
RoleMap rMapa = new RoleMap(hibernate.getPolaczenie().openSession());
```

od tego momentu możemy wywoływać wszystkie dostępne metody z rMap nie obciążając dodatkowo zasobów sprzętowych maszyn, na których będzie działał nasz program.

5. Inne sposoby mapowania tabel.

Tworzenie mapowania może odbywać się także poprzez wstawki @ (tzw. persistence). Dzięki temu nie musimy tworzyć plików hbm.xml. Zamiast tego w klasach mapowanych do tabel można dopisać wskazanie, które pole odpowiada któremu polu w klasie. Jeżeli nazwa pola w klasie jest identyczna do nazw pola w tabeli wskazanie można pominąć. Podobnie ma się sprawa z nazwami klasy i nazwami łączonych tabel.

Wykorzystując stworzony przez nas przykład mapowana klasa osoby miała by następujący wygląd:

```
package Hibernate.Gradle;
import javax.persistence.*;                    //1
@Entity                                        //2
@Table(name = "osoby")                         //3
public class Osoby {
        @Id                                                              //4
        @Column(name = "id_osoba")                                       //5
        @GeneratedValue(strategy = GenerationType.IDENTITY)              //6
        private long id;
        @Column(name = "imię")                                           //7
        private String name;
        @Column(name = "nazwisko")                                       //8
        private String surname;
        //tego pola nie trzeba łączyć; nazwa pola w klasie oraz tabeli jest taka sama
        private String nick;                                             //9
        @Column(name = "data_urodzenia")                                 //10
        private String birthdate;
        @Column(name = "emial")                                          //11
        private String email;
        @Column(name = "telefon")                                        //12
        private String telephone;
        public Osoby() {

        }
                public void setId(long id) {
                this.id=id;
        }
        public long getId() {
                return id;
        }
        public void setName(String name) {
                this.name=name;
        }
        public String getName() {
                return name;
        }
        public void setSurname(String surname) {
```

```
		this.surname=surname;
	}
	public String getSurname() {
		return surname;
	}
	public void setBirthdate(String birthdate) {
		this.birthdate=birthdate;
	}
	public String getBirthdate() {
		return birthdate;
	}
	public void setNick(String nick) {
		this.nick=nick;
	}
	public String getNick() {
		return nick;
	}
	public void setEmail(String email) {
		this.email=email;
	}
	public String getEmail() {
		return email;
	}
	public void setTelephone(String telephone) {
		this.telephone=telephone;
	}
	public String getTelephone() {
		return telephone;
	}
}
```

Pierwsza linia kodu dodaje obsługę znaczników potrzebnych dla mapowania Hibernate.

Druga linia wskazuje Hibernate, że wskazana klasa jest jednostką odwzorowywaną do tabeli w bazie. Bez tej adnotacji mapowanie nie zadziała.

Trzecia linia jest wymagana w przypadku, gdy nazwa mapowanej tabeli jest inna niż naszej klasy. Jeżeli przyjmujemy konwencję nazw tabel inną niż konwencja nazw klas w Java – musimy tutaj wskazać dokładną nazwę tabeli odwzorowywanej.

Linia czwarta wskazuje pole w tabeli, które jest polem identyfikującym – kluczem głównym (podstawowym). Hibernate będzie wiedział by tego pola nie wymagać przy zapisie do bazy (będzie uzupełnianie wedle reguły podanej poniżej).

Linia piąta wskazuje nazwę, jaką to pole posiada w tabeli.

Linia szósta określa metodę, wedle jakiej pole będzie uzupełniane o nowe wartości. Wybrany sposób identyfikuje nowe wartości jako jednostki (jak w bazie danych).

Pozostałe linie, poza dziewiątą, stanowią odwzorowanie pól klas – pól tabel.

Linia dziewiąta jest jedyną, w której nie następuje adnotacja. Dzieje się tak, ponieważ pole w tabeli ma dokładnie tę samą nazwę co pole w klasie.

Main.java | Przepis.java | Skladnik.java | Kategorie.java | baza.sql | KategorieSterownik.java

```java
package Hibernate.Gradle;
import javax.persistence.*;

@Entity
@Table(name = "osoby")
public class Osoby {
    @Id
    @Column(name = "id_osoba")
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private long id;
    @Column(name = "imie")
    private String name;
    @Column(name = "nazwisko")
    private String surname;
    //tego pola nie trzeba łączyć; nazwa pola w klasie oraz tabeli jest taka sama
    private String nick;
    @Column(name = "data_urodzenia")
    private String birthdate;
    @Column(name = "emial")
    private String email;
    @Column(name = "telefon")
    private String telephone;

    public Osoby() {

    }

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

    public long getId() {
        return id;
    }

    public void setName(String name) {
        this.name=name;
    }

    public String getName() {
        return name;
    }

    public void setSurname(String surname) {
        this.surname=surname;
    }

    public String getSurname() {
```

Powyżej przedstawiony został fragment kodu z adnotacjami bazy danych. Reszta klasy pozostaje bez zmian.

Tworząc konfigurację Hibernate w ten sposób nadal możemy korzystać a pliku cfg.xml celem konfiguracji opcji Hibernate. Plik będzie się różnić w kwestii podpięcia mapowania klas. Zamiast:

<mapping resource="Osoby.hbm.xml"></mapping>

<mapping resource="Role.hbm.xml"></mapping>

wprowadzimy:

<mapping class="Hibernate.Gradle.Osoby"></mapping>

<mapping class="Hibernate.Gradle.Role"></mapping>

W związku z tym plik będzie wyglądał następująco:

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE hibernate-configuration PUBLIC
"-//Hibernate/Hibernate Configuration DTD 4.0//EN"
"http://hibernate.net/hibernate-configuration-4.0.dtd">
<hibernate-configuration>
    <session-factory>
        <property name="hibernate.connection.driver_class">com.mysql.cj.jdbc.Driver</property>
        <property name="hibernate.connection.url">jdbc:mysql://192.168.1.220:3306/esport</property>
        <property name="hibernate.connection.password">test</property>
        <property name="hibernate.connection.username">test</property>
        <property name="hibernate.dialect">org.hibernate.dialect.MySQLDialect</property>
        <property name="show_sql">true</property>
        <property name="hbm2ddl.auto">update</property>
        <mapping class="Hibernate.Gradle.Osoby"></mapping>
        <mapping class="Hibernate.Gradle.Role"></mapping>
    </session-factory>
</hibernate-configuration>
```

Ostatnią możliwością jest wyeliminowanie całego pliku cfg.xml. W tym momencie, najlepiej w pliku głównym projektu, możemy dodać nową zmienną obiektową tworzoną z klasy Configuration:

```
Configuration cfg = new Configuration();
cfg.setProperty("hibernate.connection.username", "test");
cfg.setProperty("hibernate.connection.password", "test");
cfg.setProperty("hibernate.connection.driver_class", "com.mysql.cj.jdbc.Driver");
cfg.setProperty("hibernate.connection.url",
"jdbc:mysql://192.168.1.220:3306/esport");
cfg.setProperty("hbm2ddl.auto", "update");
cfg.setProperty("hibernate.dialect", "org.hibernate.dialect.MySQLDialect");
cfg.setProperty("show_sql", "true");
cfg.addAnnotatedClass(Osoby.class);
```

```
cfg.addAnnotatedClass(Role.class);
```

Tłumaczenie linia po linii:

- tworzymy nową zmienną z klasy Configuration
- dodajemy nazwę użytkownika bazy danych
- dodajemy hasło użytkownika bazy danych
- podajemy nazwę klasy, przez którą będziemy realizować połączenie do naszej bazy danych
- podajemy pełny adres połączeniowy do naszej bazy danych (w tym wypadku jeden z adresów lokalnych)
- ustawiamy strategię aktualizacji na aktualizację (zmienna została omówiona przy pliku cfg.xml)
- wskazujemy jakiego języka SQL będziemy używać (parametr opcjonalny)
- ponieważ działamy na wersji roboczej (nie produkcyjnej) możemy sobie włączyć pokazywanie zapytań SQL (ten parametr jest w pełni opcjonalny)
- linia dodaje klasę Osoby jako mapowaną (czytanie adnotacji)
- linia dodaje klasę Role jako mapowaną (czytanie adnotacji)

Po utworzeniu konfiguracji wczytujemy ją poprzez następującą linię:

```
polaczenieHibernate = cfg.buildSessionFactory();
```

Cała funkcja setup będzie miała następującą postać:

```
protected void setup() {
		Configuration cfg = new Configuration();
		cfg.setProperty("hibernate.connection.username", "test");
		cfg.setProperty("hibernate.connection.password", "test");
		cfg.setProperty("hibernate.connection.driver_class", "com.mysql.cj.jdbc.Driver");
		cfg.setProperty("hibernate.connection.url",
"jdbc:mysql://192.168.1.220:3306/esport");
		cfg.setProperty("hbm2ddl.auto", "update");
		cfg.setProperty("hibernate.dialect", "org.hibernate.dialect.MySQLDialect");
		cfg.setProperty("show_sql", "true");
		cfg.addAnnotatedClass(Osoby.class);
		cfg.addAnnotatedClass(Role.class);
		try {
			polaczenieHibernate = cfg.buildSessionFactory();
		} catch (Exception ex) {
		   throw new RuntimeException(ex);
		}
   }
```

Wersja ze zrzutu ekranu:

```java
package Hibernate.Gradle;

import org.hibernate.SessionFactory;

import org.hibernate.cfg.Configuration;
import driver.map.RoleMap;

public class Main {

    private SessionFactory polaczenieHibernate;

    protected void setup() {
        Configuration cfg = new Configuration();
        cfg.setProperty("hibernate.connection.username", "test");
        cfg.setProperty("hibernate.connection.password", "test");
        cfg.setProperty("hibernate.connection.driver_class", "com.mysql.cj.jdbc.Driver");
        cfg.setProperty("hibernate.connection.url", "jdbc:mysql://192.168.1.220:3306/esport");
        cfg.setProperty("hbm2ddl.auto", "update");
        cfg.setProperty("hibernate.dialect", "org.hibernate.dialect.MySQLDialect");
        cfg.setProperty("show_sql", "true");
        cfg.addAnnotatedClass(Osoby.class);
        cfg.addAnnotatedClass(Role.class);
        try {
            polaczenieHibernate = cfg.buildSessionFactory();
        } catch (Exception ex) {
            throw new RuntimeException(ex);
        }
    }

    protected void exit() {
        if (polaczenieHibernate != null) {
            polaczenieHibernate.close();
            System.out.println("Zakonczenie");
        }
    }

    protected SessionFactory getPolaczenie() {
        return polaczenieHibernate;
    }

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        System.out.println("Projekt Hibenrate Gradle");
        Main hibernate = new Main();
        hibernate.setup();

        System.out.println("Stanowisko o id 2 " + new RoleMap(hibernate.getPolaczenie().openSession()).read(2).getNazwa());
        System.out.println("Stanowisko id dla Trener " + new RoleMap(hibernate.getPolaczenie().openSession()).read("Trener").getId());
```

Na zrzucie widoczne są także metody używane w poprzedniej wersji po to, by zorientować się w jaki sposób zmienił się kod.

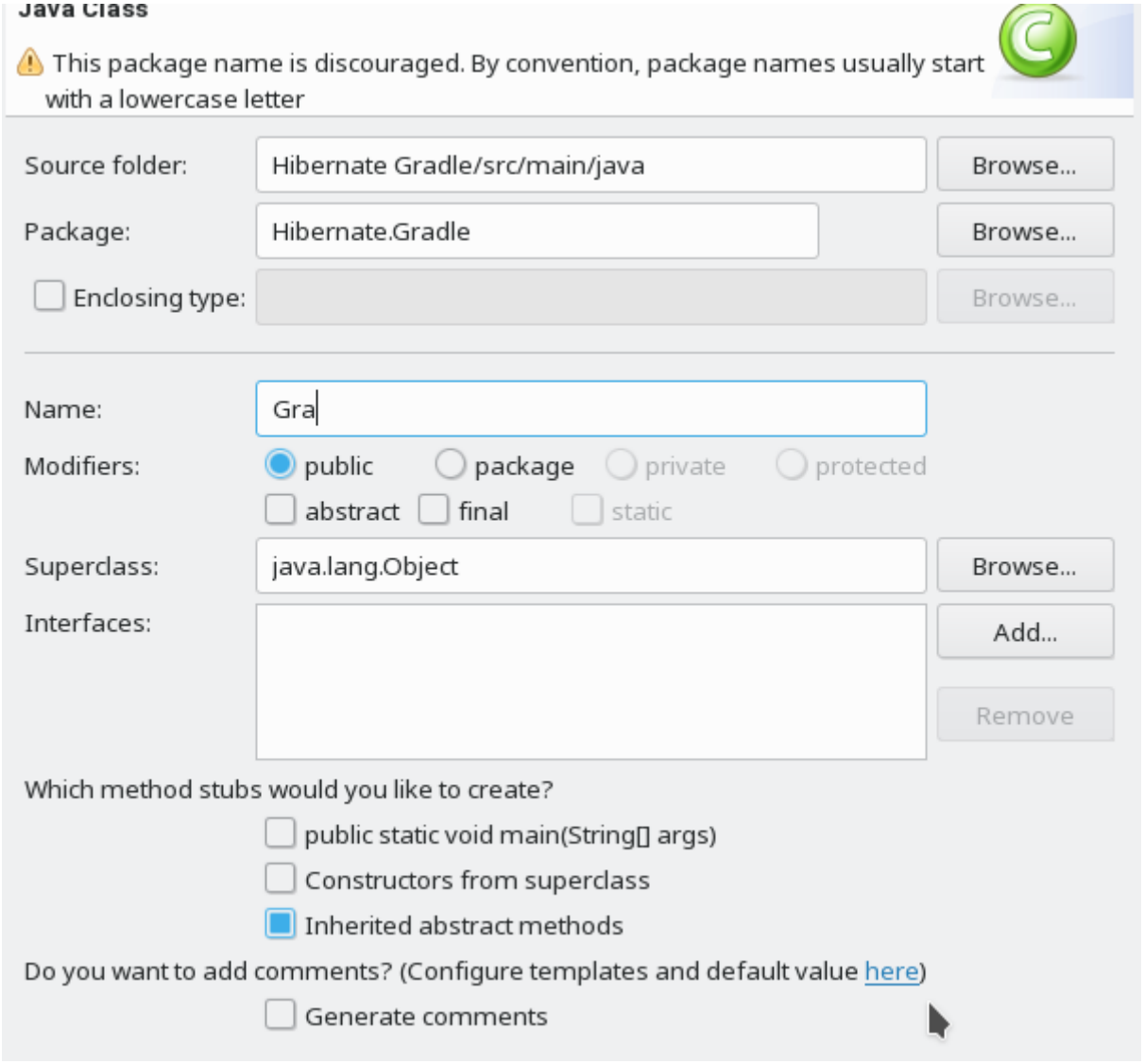
6. Aplikacja nie wykorzystująca Hibernate.

Jako kontrę wyobraźmy sobie aplikację, w której wszystko wykonujemy sami – całą logikę połączenia oraz komunikacji z bazą danych. Porównajmy obciążenie pracy programisty. W kolejnych krokach spróbujemy wykonać połączenie do bazy danych esport, jednak uzupełnimy nową tabelę – gry. Tabela ma następującą budowę:

```
CREATE TABLE `gry` (
  `id_gra` bigint(20) UNSIGNED NOT NULL,
  `nazwa` varchar(40) NOT NULL,
  `gatunek` varchar(15) NOT NULL,
  `tryb_gry` varchar(15) NOT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;
```

Tabelę odwzorujemy w programie za pomocą klasy odpowiadającej naszej tabeli oraz klasy-sterownika, odpowiadającemu większej licznie rekordów (zdolnej obsługiwać nawet wszystkie wpisy w tabeli).

a) tworzymy nową klasę Gra.java w paczce Hibernate.Gradle:



b) ponieważ używamy projektu Gradle (lub Maven) nie musimy ręcznie dodawać biblioteki łączącej do serwera MySQL. Wystarczy upewnić się, że w konfiguracji jest linijka:

api 'mysql:mysql-connector-java:8.0.13' //Gradle

w przypadku Maven:

```
<dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <version>8.0.13</version>
</dependency>
```

c) tworzymy w klasie typ enumeryczny wskazujący na używane pole. Typ ten będzie potrzebny przy przeszukiwaniu pól celem wybrania odpowiedniej gry z listy danych, modyfikacji oraz usuwania. Jego pełne możliwości zostaną zaprezentowane później.

```
public static enum TYP {
        ID(0),
        NAZWA(1),
        GATUNEK(2),
        TRYB(3);

        private int x;

        private TYP(int x) {this.x=x;}

        public int get() {return x;}
        public static TYP typ(int i) {
                switch (i) {
                        case 0: return TYP.ID;
                        case 1: return TYP.NAZWA;
                        case 2: return TYP.GATUNEK;
                        case 3: return TYP.TRYB;
                }
                return null;
        }
}
```

d) dotworzone zostaną stałe statyczne (final static) odzwierciedlające identyfikatory pól w bazie danych; pozwoli to na łatwiejsze przemieszczanie się w polach danych:

```
public final static int ID = 0;
public final static int NAZWA = 1;
public final static int GATUNEK = 2;
public final static int TRYB = 3;
```

Fragment kodu w pliku prezentuje się następująco:

```java
package Hibernate.Gradle;

public class Gra {

    public static enum TYP {
        ID(0),
        NAZWA(1),
        GATUNEK(2),
        TRYB(3);

        private int x;

        private TYP(int x) {this.x=x;}

        public int get() {return x;}
        public static TYP typ(int i) {
            switch (i) {
                case 0: return TYP.ID;
                case 1: return TYP.NAZWA;
                case 2: return TYP.GATUNEK;
                case 3: return TYP.TRYB;
                //default: return null;
            }
            return null;
        }
    }

    public final static int ID = 0;
    public final static int NAZWA = 1;
    public final static int GATUNEK = 2;
    public final static int TRYB = 3;

    private long id;
    private String nazwa;
    private String gatunek;
    private String tryb;

```

e) dodajemy zmienne odwzorowujące pola z tabeli (pojedynczego wpisu); pola widoczne na zrzucie powyżej

```
private long id;
private String nazwa;
private String gatunek;
private String tryb;
```

f) tworzymy własne konstruktory klasy by móc dodawać zarówno „puste” wpisy jak i te „pełne”. Tylko jeden z nich definiuje przypisanie polom wartości – pozostałe stanowią jedynie odwołania.

```
public Gra() {
    this(null,null,null);
}
```

```
public Gra(String nazwa) {
        this(nazwa,null,null);
}

public Gra(String nazwa, String gatunek) {
        this(nazwa,gatunek,null);
}

public Gra(String nazwa, String gatunek, String tryb) {
        this(nazwa,gatunek,tryb,-1);
}

public Gra(String nazwa, String gatunek, String tryb, long id) {
        this.nazwa=nazwa;
        this.gatunek=gatunek;
        this.tryb=tryb;
        this.id=id;
}
```

```
32
33      private long id;
34      private String nazwa;
35      private String gatunek;
36      private String tryb;
37
38      public Gra() {
39          this(null,null,null);
40      }
41
42      public Gra(String nazwa) {
43          this(nazwa,null,null);
44      }
45
46      public Gra(String nazwa, String gatunek) {
47          this(nazwa,gatunek,null);
48      }
49
50      public Gra(String nazwa, String gatunek, String tryb) {
51          this(nazwa,gatunek,tryb,-1);
52      }
53
54      public Gra(String nazwa, String gatunek, String tryb, long id) {
55          this.nazwa=nazwa;
56          this.gatunek=gatunek;
57          this.tryb=tryb;
58          this.id=id;
59      }
60
```

g) tworzymy pola get i set dla pól naszej klasy; nie dodajemy pola setId gdyż to pole będzie automatycznie modyfikowane w bazie i nie będziemy mogli samodzielnie nim manipulować:

```
public long getId() {
	return id;
}

public String getNazwa() {
	return nazwa;
}

public String getGatunek() {
	return gatunek;
}

public String getTryb() {
	return tryb;
}

public void setNazwa(String nazwa) {
	this.nazwa=nazwa;
}

public void setGatunek(String gatunek) {
	this.gatunek=gatunek;
}

public void setTryb(String tryb) {
	this.tryb=tryb;
}
```

```java
public Gra(String nazwa, String gatunek, String tryb, long id) {
    this.nazwa=nazwa;
    this.gatunek=gatunek;
    this.tryb=tryb;
    this.id=id;
}

public long getId() {
    return id;
}

public String getNazwa() {
    return nazwa;
}

public String getGatunek() {
    return gatunek;
}

public String getTryb() {
    return tryb;
}

public void setNazwa(String nazwa) {
    this.nazwa=nazwa;
}

public void setGatunek(String gatunek) {
    this.gatunek=gatunek;
}

public void setTryb(String tryb) {
    this.tryb=tryb;
}

public void setGra(Gra gra) {
    this.gatunek=gra.getGatunek();
    this.nazwa=gra.getNazwa();
    this.tryb=gra.getTryb();
    this.id=gra.getId();
}
```

h) na zrzucie powyżej widać tzw. metodę kopiującą. Jej zadaniem jest zapisanie do naszego obecnego obiektu wartości podanych w innym obiekcie. Obiekty pomiędzy sobą będą niezależne (modyfikacja jednego nie wpłynie na wartości drugiego).

```
public void setGra(Gra gra) {
        this.gatunek=gra.getGatunek();
        this.nazwa=gra.getNazwa();
        this.tryb=gra.getTryb();
        this.id=gra.getId();
```

}

i) na koniec nasza klasa zostanie wzbogacona o funkcję wyszukującą. Pozwoli ona na ustalenie, czy podana przez użytkownika nazwa istnieje w naszym spisie wartości. Jeżeli tak – zostanie zwrócony numer kolumny, do której mapowana będzie wskazana wartości. Jeżeli wartość nie istnieje – zwrócona zostanie wartość -1:

```
public int zawiera(Object o) {
        if (o.getClass().isPrimitive()) {
            if ((long)o == this.id)
                return ID;
        }
        else if (o.getClass() == String.class) {
            if (o.toString().compareTo(this.gatunek)==0)
                return GATUNEK;
            if (o.toString().compareTo(this.nazwa)==0)
                return NAZWA;
            if (o.toString().compareTo(this.tryb)==0)
                return TRYB;
        }
        return -1;
}
```

```
85      public void setTryb(String tryb) {
86          this.tryb=tryb;
87      }
88
89      public void setGra(Gra gra) {
90          this.gatunek=gra.getGatunek();
91          this.nazwa=gra.getNazwa();
92          this.tryb=gra.getTryb();
93          this.id=gra.getId();
94      }
95
96      public int zawiera(Object o) {
97          if (o.getClass().isPrimitive()) {
98              if ((long)o == this.id)
99                  return ID;
100         }
101         else if (o.getClass() == String.class) {
102             if (o.toString().compareTo(this.gatunek)==0)
103                 return GATUNEK;
104             if (o.toString().compareTo(this.nazwa)==0)
105                 return NAZWA;
106             if (o.toString().compareTo(this.tryb)==0)
107                 return TRYB;
108         }
109         return -1;
110     }
111 }
```

W ten oto sposób kończymy definicję klasy potrzebnej do odwzorowania JEDNEGO wpisu w tabeli. Teraz zajmiemy się plikiem obsługującym zarządzanie całą listą wpisów.

a) tworzymy w paczce driver.map plik klasy GryDriver.java (analogicznie do pliku Gra.java)

b) dodajemy niezbędne pliki obsługujące połączenie z bazą danych oraz plik naszej klasy Gra:

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.ArrayList;
import java.util.List;

import Hibernate.Gradle.Gra;
```

c) tworzymy zmienną przechowującą rekordy informacji o grach. Wykorzystamy listę, ze względu na łatwość zarządzania wpisami:

```
private List<Gra> gry;
```

d) tworzymy zmienną służącą do połączenia z bazą danych. Można pominąć definicję tej zmiennej i wstawić literał bezpośrednio w metodzie połączenia, jednak dzięki temu rozwiązaniu, w razie potrzeb, możemy szybko zmienić parametry dostępu do serwera czy bazy danych.

```
private String poleczenieAdres="jdbc:mysql://192.168.1.220:3306/esport?
user=test&password=test";
```

Alternatywnie można nawet poszczególne składowe połączenia rozbić na stałe celem czystszego zapisu:

```
    private final String serwer = "192.168.1.220";
    private final String baza = "esport";
    private final String port = "3306";
    private final String uzytkownik = "test";
    private final String haslo = "test";
```

Po czym tworzymy ciąg połączeniowy na jeden z dwóch sposobów. Albo tworzymy ciąg poprzez metodę format:

```
private final String poleczenieAdres = String.format("jdbc:mysql://%s:%s/%s?user=%s&password=
%s", serwer,port,baza,uzytkownik,haslo);
```

Albo poprzez operator łączenia (plus):

```
private final String poleczenieAdres = "jdbc:mysql://"+serwer+":"+port+"/"+baza+"?
user="+uzytkownik+"&password="+haslo;
```

e) na koniec tworzymy zmienną, która będzie przechowywała uchwyt połączenia do bazy danych:

```
private Connection polaczenie = null;
```

f) tworzymy konstruktory w taki sposób, by za każdym razem tworzyła się przynajmniej nowa lista wartości gier.

```
public GryDriver() {
		this(new ArrayList<Gra>());
	}

	public GryDriver(Gra gra) {
		this.gry = new ArrayList<Gra>();
		this.gry.add(gra);
	}

	public GryDriver(List<Gra> gry) {
		this.gry = gry;
	}
```

g) tworzymy metody połączenia z bazą oraz ewentualnego rozłączenia z nią:

```
	public boolean otworzPolaczenie() {
		return otworzPolaczenie(false);
	}

	public boolean otworzPolaczenie(boolean test) {
		try {
			polaczenie = DriverManager.getConnection(poleczenieAdres);
			return true;
		}
		catch (SQLException ex) {
			if (test)
```

```
                        System.err.println("Połączenie nie może zostać ustanowione. Kod błędu " + ex.getErrorCode() + ", komunikat: " + ex.getMessage());
                }
                return false;
        }

        public boolean zamknijPolaczenie() {
                return zamknijPolaczenie(false);
        }

        public boolean zamknijPolaczenie(boolean test) {

                        try {
                                polaczenie.close();
                                return true;
                        } catch (SQLException e) {
                                if (test)
                                        System.err.println("Połączenie nie zostało zamknięte ponieważ nigdy nie zostało otwarte.");
                        }

                return false;
        }
```

W powyższym przykładzie najważniejsza dla połączenia jest następująca instrukcja:

```
polaczenie = DriverManager.getConnection(poleczenieAdres);
```

Metoda getConnection powoduje nawiązanie połączenia do bazy, o ile ciąg połączeniowy wskazuje na właściwy serwer. Jeżeli połączenie nie będzie mogło być zrealizowane wykonana zostanie operacja z wyjątku (catch).

h) Pierwszym etapem będzie załadowanie danych z tabeli. W tym celu wywołamy odpowiednie zapytanie SQL:

```
"SELECT * FROM `gry`;"
```

Utworzymy do tego celu odpowiednią metodę zaladuj(). Sama metoda będzie mogła być wywołana z maksymalnie dwoma parametrami. Pierwszy z nich będzie wskazywał, czy listę załadowanych gier wczytujemy od nowa, czy tylko ją aktualizujemy (synchronizacja ze stanem bazy danych).

Drugi parametr będzie przydatny podczas projektowania programu – jeżeli będzie ustawiony to metoda zwróci informacje o ewentualnych problemach z połączeniem.

```
	public void zaladuj() {
		zaladuj(false, false);
	}

	public void zaladuj(boolean aktualizacja) {
		zaladuj(aktualizacja, false);
	}

	public void zaladuj(boolean aktualizacja, boolean test) {
		try {
			Statement polecenie = polaczenie.createStatement();
			ResultSet wynik = polecenie.executeQuery("SELECT * FROM `gry`;");
			while (wynik.next()) {
				Gra gtmp=new
Gra(wynik.getString("nazwa"),wynik.getString("gatunek"),wynik.getString("tryb_gry"),wynik.getL
ong(1));
				if (aktualizacja && gry.indexOf(gtmp)>-1)
					continue;
				gry.add(gtmp);
			}
		} catch (SQLException ex) {
			if (test)
				System.err.println("Pobieranie danych nie zakończyło się poprawnie.
Kod błędu " + ex.getErrorCode() + ", komunikat: " + ex.getMessage());
		}
	}
```

W linii:

```
Gra gtmp=new
Gra(wynik.getString("nazwa"),wynik.getString("gatunek"),wynik.getString("tryb_gry"),wynik.getL
ong(1));
```

tworzymy nową zmienną obiektową z wartościami pobranymi z bazy. Jeżeli tylko aktualizujemy stan listy, a pobrany rekord jest już załadowany do pamięci programu to pomijamy dodanie go do naszej listy poprzez linię:

```
if (aktualizacja && gry.indexOf(gtmp)>-1)
        continue;
```

Wszystko co będzie za słowem continue nie wykona się, a sam program przejdzie do kolejnego kroku wykonania pętli (do następnego pobranego elementu).

i) mając pobrane dane będziemy zapewne chcieli mieć do nich dostęp – np. pobrać określone informacje i wyświetlić je użytkownikom programu. W tym celu wykorzystamy odpowiednie metody pobierz:

```
public List<Gra> pobierzWszystko() {
        return gry;
    }

    public List<Gra> pobierz(Object szukany) {
        return pobierz(szukany,Gra.TYP.ID);
    }

    public Gra pobierz(int id) {
        return gry.get(id);
    }

    public int pobierzIndex(Object szukany) {
        for (Gra gra:gry) {
            if (gra.zawiera(szukany)==Gra.NAZWA) {
                return gry.indexOf(gra);
            }
        }
        return -1;
    }

    public List<Gra> pobierz(Object szukany, int typ) {
        return pobierz(szukany, Gra.TYP.typ(typ));
    }

    @SuppressWarnings("static-access")
    public List<Gra> pobierz(Object szukany, Gra.TYP typ) {
```

```
		List<Gra> g = new ArrayList<Gra>();
		for (Gra gra : gry) {
			if (gra.zawiera(szukany) != -1) {
				if (typ.get() == gra.zawiera(szukany))
					g.add(gra);
				if (typ.get() == gra.zawiera(szukany))
					g.add(gra);
				if (typ.get() == gra.zawiera(szukany))
					g.add(gra);
				if (typ.get() == gra.zawiera(szukany))
					g.add(gra);
			}
		}
		return g;
	}
```

Jak widać w kodzie, wszystkie zadeklarowane metody korzystają z jednej zbiorczej funkcji o nazwie pobierz (wyjątkiem są pobierzWszystko() oraz pobierz(int id), które mają niezależny kod). W tym momencie przydają nam się wcześniej utworzone typy enumeryczne oraz stałe utworzone w klasie Gra.java. Na tym etapie jesteśmy bowiem w stanie określić w której kolumnie chcemy znaleźć poszukiwaną frazę/wartość. Domyślnie szukamy po id w bazie, można jednak wybierać dowolne pozostałe pole.

Metoda pobierzWszystko() zwraca po prostu wszystkie możliwe wyniki (całą listę). Metoda pobierz(int id) zwraca natomiast wynik nie identyfikatora z bazy lecz identyfikator pozycji w liście w programie (to nie są tożsame wartości!).

j) niekiedy może zajść potrzeba zamiany jednego elementu na inny; w tym celu potrzebna będzie odpowiednia metoda, taka jak ta poniżej:

```
public void zamien(Gra gra, int id) {
		gry.get(id).setGra(gra);
	}
```

k) dodanie nowego elementu do listy będzie realizowane przez metodę dodaj(). Pozwala ona na wprowadzenie samej nazwy lub, jeżeli zajdzie potrzeba, także dodatkowych informacji o wprowadzanej grze.

```
public boolean dodaj(String nazwa) {
		return dodaj(nazwa, "","");
	}
```

```
	public boolean dodaj(String nazwa, String gatunek) {
		return dodaj(nazwa,gatunek,"");
	}

	public boolean dodaj(String nazwa, String gatunek, String tryb) {
		if (pobierz(nazwa,Gra.TYP.NAZWA).isEmpty()) {
			gry.add(new Gra(nazwa,gatunek,tryb));
			return true;
		}
		return false;
	}
```

Jak można zauważyć, dodawanie jest proste i może dopuścić do powielenia danych. Jednak w naszym przykładzie nie będzie to istotne.

l) zapisanie danych do bazy będzie realizowała odpowiednia funkcja zapisz:

```
public boolean zapisz() {
		String vals="";
		for (Gra gra: gry)
			if (gra.getId()==-1)
				vals+=" ('"+gra.getNazwa()+"','"+gra.getGatunek()
+"','"+gra.getTryb()+"'),";                                          //1
		if (!vals.isEmpty()) {                                       //2
			vals=vals.substring(0, vals.length()-1);                 //3
			try {
				polaczenie.createStatement().execute("INSERT INTO `gry`
(`nazwa`,`gatunek`,`tryb_gry`) VALUES"+vals+";");                    //4
				return true;
			} catch (SQLException ex) {
				System.err.println("Zapis do bazy zakończył się błędem. Kod błędu "
+ ex.getErrorCode() + ", komunikat: " + ex.getMessage());
			}
		}
		return false;
	}
```

Linia pierwsza wywoływana jest tylko w przypadku gdy id danej gry jest równe -1 (czyli została dodana podczas obecnej sesji i nie została załadowana z bazy). Do wartości vals dodany zostanie ciąg wyglądający mniej więcej tak:

(‘Starcraft II’,’strategia’,’RTS’),

jeżeli którekolwiek pole będzie puste, będą otwarte i zamknięte apostrofy (wyznaczniki ciągu znakowego w zapytaniach do bazy danych). Należy zauważyć, że na końcu każdego ciągu dodawany jest przecinek, mający na celu oddzielać kolejne wartości do zapisania (jak to ma miejsce w zapytaniu SQL).

Warunek pozwalający na wywołanie instrukcji wywołuje się z pętli for, która przeskakuje po wszystkich elementach listy!

Druga zaznaczona linia wykonuje się już po wskazanej pętli. Jeżeli zmienna vals będzie posiadała chociaż jeden wpis (vals nie będzie puste → isEmpty()) ze zmiennej zostanie skasowany ostatni znak - dzięki funkcji substring. Pierwszy parametr wskazuje odkąd zaczynamy wycinanie znaków z bazowego ciągu, ostatni zaś wskazuje, na której pozycji chcemy zakończyć wycinanie. Ponieważ usuwamy tylko ostatni znak (chodzi o wyrzucenie ostatniego przecinka), jako ostatni znak wskazujemy długość obecnego ciągu pomniejszoną o jeden; dzięki temu ostatni znak nie pozostanie dodany do nowego ciągu.

Na koniec, oznaczonej linii numer 4, wykonujemy próbę dodania do bazy wartości ze zmiennej vals (dodając ją do zapytania po słowie VALUES). Jeżeli z jakichś powodów zapytanie nie będzie mogło być przetworzone to wykonają się instrukcje z klauzuli catch.

m) kolejnym krokiem będzie aktualizacja już istniejących danych. W tym celu zostały utworzone odpowiednie funkcje:

```
public boolean aktualizuj(long id) {
        return aktualizuj((int)id);                    //1
    }

    public boolean aktualizuj(int id) {
        return aktualizujWszystko(id, "");             //2
    }

    public boolean aktualizujWszystko() {
        return aktualizujWszystko(-1, "");             //3
    }

    public boolean aktualizuj(String name) {
        return aktualizujWszystko(-1, name);           //4
    }

    public boolean aktualizujWszystko(int id, String name) {
        for (Gra gra: gry) {
```

```
                String q="";
                if (gra.getId()==id || (id == -1 && name=="")) {      //5
                        q="UPDATE `gry` SET `nazwa`='"+gra.getNazwa()
+"',`gatunek`='"+gra.getGatunek()+"',`tryb`='"+gra.getTryb()+"' WHERE `id_gra`="+gra.getId()
+";";
                }
                if (gra.getNazwa()==name && name!="") {           //6
                        q="UPDATE `gry` SET `nazwa`='"+gra.getNazwa()
+"',`gatunek`='"+gra.getGatunek()+"',`tryb`='"+gra.getTryb()+"' WHERE `nazwa`="+name+";";
                }
                if (q!="")
                        try {
                                polaczenie.createStatement().execute(q);      //7
                                if (id!=-1)
                                        return true;
                        } catch (SQLException ex) {
                                System.err.println("Zapis do bazy zakończył się błędem. Kod
błędu " + ex.getErrorCode() + ", komunikat: " + ex.getMessage());
                                return false;
                        }
        }
        return true;
    }
```

Kolejne metody pozwalają na odpowiednie obsłużenie aktualizacji danych istniejących już w bazie. Pierwsza metoda (oznaczona jedynką) pozwala na zaktualizowanie danych o podanym id jako typ long – odwołuje się do metody aktualizuj przyjmującej typ int (użyto odpowiedniego rzutowania danych).

Druga, trzecia i czwarta metoda odowłują się do jednej, zbiorczej metody o nazwie aktualizujWszystko. Każda z nich pozwala na bądź to na zaktualizowanie danych po id w bazie, bądź to po nazwie aktualnie wybranej gry. Można również wywołać metodę zaktualizujWszystko by zakutualizować wszystkie dane w bazie.

Sama metoda aktualizacja posiada dwa istotne waruki wywołujące się dla każdego elementu. Jeżeli id lub nazwa będą pasować do podanych przez programistę – nastąpi aktualizacja danych poprzez zapytanie tymczasowe q (linia oznaczona jako siódma). Jeżeli nastąpi jakikolwiek błąd – obsługę zapytania przejmie klauzula catch.

n) ostatnim elementem obsługi bazy danych jest usuwanie danych z tabeli. W tym celu posłużymy się następującymi metodami:

```
public boolean wyczysc() {
        return usun(-1,"");                     //1
```

```
}

public boolean usun() {
	return wyczysc();			//2
}

public boolean usun(String nazwa) {
	return usun(-1,nazwa);			//3
}

public boolean usun(long id) {
	return usun(id,"");			//4
}

public boolean usun(long id, String nazwa) {
	String zapytanie = "DELETE FROM `gry`";		//5
	long idIN=-1;					//6
	String nazwaIN="(";				//7
	for (Gra gra: gry) {
		if (gra.getId()==id)
			idIN=gra.getId();		//8
		if (gra.getNazwa().compareTo(nazwa)==0)
			nazwaIN+="'"+nazwa+"',";	//9
	}
	if (nazwaIN.length()>1)
		nazwaIN=nazwaIN.substring(0, nazwaIN.length()-1)+")";	//10
	else
		nazwaIN="";
	if (nazwaIN.length()>1||idIN!=-1)
		zapytanie+=" WHERE ";				//11
	if (idIN!=-1)
		zapytanie+="`id_gra`="+idIN;
	if (nazwaIN.length()>1&&idIN!=-1)
		zapytanie+=" AND ";				//12
	else if (zapytanie.contains("id_gra"))
		zapytanie+=";";					//13
```

```
        if (nazwaIN.length()>1)
                zapytanie+="`nazwa` IN "+nazwaIN+";";
        if (zapytanie.indexOf(";")==-1 && id==-1 && nazwa.isEmpty())
                zapytanie+=";";                                          //14
        if (!zapytanie.contains(";"))
                return false;
        try {
                polaczenie.createStatement().execute(zapytanie);     //15
                return true;
        } catch (SQLException ex) {
                System.err.println("Usuwanie z bazy zakończyło się błędem. Kod błędu " +
ex.getErrorCode() + ", komunikat: " + ex.getMessage());
                System.out.println(zapytanie);
        }
        return false;
```

Linie 1-4 działają analogicznie do aktualizacji danych.

Piąta linia tworzy zmienną tekstową zawierającą pierwszą część zapytania do bazy danych.

W linii szóstej i siódmej tworzymy zmienne pomocnicze zawierające błędne wartości. Wartości te będą podmienione w określonym czasie – jeżeli dane podane jako parametry funkcji zostaną odpowiednio znalezione w danych z bazy SQL.

Ósma i dziewiąta linia podmieniają wyżej wspomniane zmienne w określonych okolicznościach. Należy przy tym pamiętać, że podmiana identyfikatora przeważnie nastąpi dokładnie jeden raz (jest unikatowy z założenia), natomiast nazwy będą dodane do puli podmiany (aczkolwiek nazwa również powinna być dokładnie 1).

Dziesiąta linia, analogicznie do operacji z metody zapisz, usuwa ostatni przecinek z zapisu dodając zamiast niego zamykający nawias. Należy mieć na uwadze, że linia ta wykona się tylko wówczas gdy nazwaIN będzie zawierać co najmniej 2 lub więcej znaków. W innym wypadku zostanie wyczyszczona.

Jedenasta linia dodaj klauzulę WHERE do zapytania – o ile nazwaIN lub idIN posiadają odpowiednie wartości.

Dwunasta linia dodaje klauzulę AND o ile nazwaIN oraz idIN posiadają pożądane wartości, w przeciwnym wypadku linia ta nigdy się nie wykona.

Jeżeli warunek będzie negatywny, linia trzynasta dodaje do zapytania średnik.

W przeciwnym wypadku wykonywany jest kolejny warunek. Jeżeli i on napotka błąd (nazwaIN nie będzie jednak zawierać nazwy – jej długość będzie mniejsza od zadeklarowanej) – wykona się linia czternasta.

Piętnasta linia wykonuje próbę wywołania polecenia UPDATE. Podobnie jak w pozostałych wypadkach – jeżeli zakończy się fiaskiem, zareaguje obsługa wyjątku.

7. Podsumowanie.

Jak można zauważyć, program wykorzystujący technologię Hibernate wymaga od programisty utworzenia znacznie mniejszej ilości kodu. W zasadzie większość pracy sprowadza się do utworzenia odpowiednich plików konfiguracyjnych lub odpowiedniego mapowania wartości przez adnotacje w kodzie.

Należy jednak pamiętać, że Hibernate ma pewne ograniczenia. Przede wszystkim technologia odwzorowuje tabele 1:1 (wszystkie pola). Co za tym idzie, przystosowana jest do uzupełniania wszystkich danych w danym wierszu, tak samo aktualizacji. Nie zawsze rozwiązanie tego typu będzie efektywne. Tak samo w kwestii pobierania danych z bazy Hibernate może okazać się mało efektywny, chyba że będziemy budować własne zapytania wybierające określone dane. Wyjątkiem od tego będą sytuacje, gdzie program będzie potrzebował wszystkich danych celem poprawnego działania.

Z kolei przedstawiony powyżej sposób połączenia do bazy danych (klasa GryDriver.java) z początku może wydawać się bardzo czasochłonny. Jednak mamy większą kontrolę nad tym co i w jaki sposób będzie zapisywane w bazie. Należy bowiem pamiętać, że możemy tworzyć dowolną liczbę funkcji komunikujących się do bazy lub dowolnie zmieniać zapytania w pojedynczej funkcji dostępowej do tabel.

Trzeba też pamiętać, że można napisać własną, indywidualną klasę do obsługi połączenia, a w klasach-sterownikach modyfikować treść zapytań. Tego typy działanie będzie najefektywniejsze, jednak najmniej uniwersalne.

Stąd ostateczny wybór obsługi połączeń z bazą danych to kwestia indywidualnego podejścia do tworzonego oprogramowania. Jeżeli zależy nam na jak najlepszym dopasowaniu oprogramowania do naszych rozwiązań, efektywności w przekazywaniu danych oraz pełnej kontroli nad zapytaniami SQL – najlepszym wyborem będzie tworzenie indywidualnych metod połączeniowych.

Natomiast jeżeli zależy nam na jak najprostszym, niemal bezobsługowym połączeniu do bazy, do tego w pełni standaryzowanym, możliwym do wykorzystania w jak największej ilości różnych projektów – idealnym wyborem będzie Hibernate.

Na koniec należy pamiętać o obecnym trendzie, który wskazuje na nowe rodzaje przechowywania danych – bądź to w pełni NoSQL, bądź w systemie mieszanym (dane bieżące NoSQL, dane archiwalne SQL). W takim wypadku Hibernate traci na znaczeniu, gdyż jest on w pełni przystosowany jedynie do rozwiązań relacyjnych, zaś w bazach NoSQL radzi sobie znacznie gorzej lub wcale (domyślnie w ogóle nie obsługuje baz nierelacyjnych).

Oba projekty, zarówno Maven, jak i Gradle, można znaleźć na repozytorium gitlab.com – https://gitlab.com/tebjava. Projekty zawierają jedynie szkielet połączeń do baz danych, można wykorzystać je celem zbudowania w pełni działającej aplikacji.

Odnośniki do repozytoriów:

https://gitlab.com/tebjava/java-maven-hibernate – repozytorium z narzędziem Maven

https://gitlab.com/tebjava/java-gradle-hibernate – repozytorium z narzędziem Gradle

Źródła i materiały dodatkowe:

https://stackoverflow.com/questions/6074678/setting-properties-programmatically-in-hibernate

https://docs.jboss.org/hibernate/orm/4.1/javadocs/org/hibernate/cfg/AvailableSettings.html

https://mkyong.com/hibernate/how-to-add-hibernate-xml-mapping-file-hbm-xml-programmatically/

https://stackoverflow.com/questions/6692882/hibernateerror-entity-class-not-found/6692937

https://www.tutorialspoint.com/hibernate/hibernate_mapping_files.htm

https://www.tutorialspoint.com/hibernate/hibernate_annotations.htm

https://docs.jboss.org/hibernate/orm/3.6/quickstart/en-US/html/hibernate-gsg-tutorial-basic.html

https://www.codejava.net/frameworks/hibernate/hibernate-hello-world-tutorial-for-beginners-with-eclipse-and-mysql

https://stackoverflow.com/questions/46920461/java-lang-noclassdeffounderror-javax-activation-datasource-on-wsimport-intellij

https://www.roseindia.net/hibernate/

https://www.tutorialspoint.com/hibernate/hibernate_examples.htm

https://www.javatpoint.com/steps-to-create-first-hibernate-application

https://howtodoinjava.com/hibernate/hibernate-3-introduction-and-writing-hello-world-application/

https://en.wikipedia.org/wiki/Gradle

https://pl.wikipedia.org/wiki/Apache_Maven