Laboratorium 5 - SOA.
Tematyka: Wprowadzenie do JPA

Praca z bazami danych z JPA

Celem laboratorium jest zaznajomienie z technologią pracy z relacyjnymi bazami danych w oparciu o specyfikacje JPA.

O samej specyfikacji warto poczytać np. tutaj-> https://javaee.github.io/tutorial/persistence-intro.html#BNBPZ

Wstęp teoretyczny.

Java Persistence API (JPA):

- specyfikacja dla bibliotek mapowania obiektowo-relacyjnego umożliwiających bezpośredni dostęp do warstwy Persystencji (bazy danych),
- popularne implementacje:
  - Hibernate (implementacja w serwerze WildFly),
  - EclipseLink (implementacja referencyjna);
- nie wymaga budowania skomplikowanych obiektów DAO (ang. Data Access Object),
- obsługuje transakcje ACID
- niezależna od dostawcy bazy danych:
- sterowniki dla wszystkich popularnych baz,
- przy prostszych przypadkach możliwość uniknięcia zabawy z SQL.

Podstawowym elementem specyfikacji są tzw. klasy encyjne. Są to klasy odwzorowywane na tabele przechowywane w bazie danych i mające następujące własności:

- zwykłe klasy POJO (Plain Old Java Object),
- pola klasy nie mogą być publiczne,
- każdy obiekt encyjny ma jednoznacznie identyfikujący go klucz główny:
  - złożone klucze główne są reprezentowane przez oddzielne klasy, które muszą implementować metody hashCode() i equals() ;
- oznaczone za pomocą adnotacji.

Istotne adnotacje JPA:

Adnotacje na poziomie klasy:

@Entity – oznaczenie klasy jako encyjnej (wymagana),

@Table – własności tabeli w bazie danych, np.:

name – nazwa tabeli,

indexes – dodatkowe indeksy (domyślny indeks dla klucza głównego).

Adnotacje na poziomie pól:

@Id – oznaczenie klucza głównego,

@GeneratedValue – automatyczne generowanie wartości klucza głównego,

@Column – własności kolumny w bazie danych, np.:

name – nazwa kolumny,

nullable – czy pole jest wymagane,

unique – czy kolumna przechowuje unikalne wartości,

updatable – czy wartość w kolumnie można modyfikować po zapisaniu nowego wiersza w tabeli;

@Temporal – wymagane dla typów Date i Calendar : umożliwia wykorzystanie bazodanowych typów do przechowywania daty/czasu, jeśli baza je oferujel

@Transient – oznaczenie pól klasy, które mają zostać pominięte przy mapowaniu obiektowo-relacyjnym.

Inne Pojęcia występujące w JPA

Jednostka trwałości (Persistence Unit) grupuje klasy encyjne mapowane na tabele w tej samej bazie danych. Aplikacja może definiować kilka jednostek trwałości (używać kilku baz danych). Jej zadaniem jest określenie sposób połączenia z bazą danych: nazwa hosta, port, nazwa bazy, login, hasło, sterownik JDBC do połączenia, źródło danych (ang. DataSource) dostarczane przez środowisko wykonawcze (np. serwer aplikacji). Standardowo konfigurowana w pliku persistence.xml (w katalogu META-INF projektu),

Kontekst (ang. Persistence Context) to zbiór zarządzanych obiektów encyjnych. Zmiany w zarządzanych obiektach encyjnych są śledzone i zapisywane do bazy danych po zatwierdzeniu transakcji.

Każdy obiekt posiada tzw. tożsamość bazodanowa. Tozsamość obiektu encyjnego wyrażona identyfikatorem powiązanym z istniejącym kluczem głównym w bazie danych.

Entity Manager - udostępnia podstawowe operacje zarządzania encjami.

void persist(Object o) – zapis do bazy danych

void remove(Object o) – usunięcie encji

void refresh(Object o) – aktualizuje stan obiektu encyjnego na podstawie bazy,

<T> T find(Class<T> entityClass, Object key) – wyszukiwanie na podstawie klucza głównego,

EntityTransaction getTransaction() – zwraca obiekt transakcji:

większość frameworków umożliwia automatyczne zarządzanie transakcjami.

Entity Transaction:

begin() – rozpoczyna transakcję,

commit() – kończy transakcję, zapisuje zmiany do bazy, odłącza zarządzane

obiekty encyjne,

rollback() — wycofuje transakcję, odłącza zarządzane obiekty encyjne.

Środowisko wykonawcze (np. serwer aplikacji) dostarcza instancje klasy EntityManager .

```
EntityManagerFactory factory = Persistence.createEntityManagerFactory("booksPu");
EntityManager em = factory.createEntityManager();
@ApplicationScoped
public class BookService {
    @PersistenceContext(name = "bookPu")
    private EntityManager em;
}
```

JPA posiada własny język - Java Persistence Query Language (JPQL):

- obiektowo zorientowany język zapytań o składni nawiązującej do języka SQL,
- opiera się na zdefiniowanym w projekcie modelu klas encyjnych,
- umożliwia odpytywanie bazy danych (SELECT), aktualizację (UPDATE) i usuwanie (DELETE) encji,
- zapytania automatycznie tłumaczone na język SQL:
    - z uwzględnieniem automatycznych złączeń tabel w czasie trawersowania
    - związków między encjami.

Przykłady.

Wywołanie zapytań bez parametrów:

```
String queryString = "SELECT p FROM Product p";

Query query = em.createQuery(queryString, Product.class);

List<Product> products = query.getResultList();
```

Wywołanie zapytań z parametrami:

```
String queryString = "SELECT p FROM Product p WHERE p.name LIKE :name";

Query query = em.createQuery(queryString, Product.class);

query.setParameter("name", "Monitor");

List<Product> products = query.getResultList();
```

Zliczanie:

```
String queryString = "SELECT COUNT(p) FROM Product p WHERE p.name LIKE :name";

Query query = em.createQuery(queryString, Long.class);

query.setParameter("name", "Laptop");

Long count = query.getSingleResult();
```

Dla zapytań UPDATE oraz DELETE:

```
String queryString = "DELETE FROM Product p";
```

```
Query query = em.createQuery(queryString);

int changedRows = query.executeUpdate();
```

JPA w aplikacji Java EE:

- połączeniem z bazą danych zarządza serwer aplikacji:
  - konfiguracja danych dostępowych zależna od wybranego serwera,
  - konfiguracja nie występuje w kodzie samej aplikacji:
    - przenośność pomiędzy serwerami (+),
    - rozproszenie konfiguracji (-);
- serwer wykorzystuje pulę połączeń z bazą danych do obsługi wielu użytkowników,
- połączenie jest udostępniane aplikacji jako zasób JNDI.
- Wstrzykiwanie obiektu klasy EntityManager:

```
@PersistenceContext

EntityManager em;
```

- transakcje na serwerze Java EE
  - zarządzane przez kontener – w warstwie EJB (automatycznie),
  - zarządzane przez użytkownika (dewelopera),
  - zarządzane w beanach CDI (@Transactional);

Aby korzystać z JPA niezbędna jest konfiguracja połączenie z bazą danych. Można to zrobić na dwa sposoby:

- konfigurując serwer aplikacyjny

- lub definiując połączenie bezpośrednio w aplikacji

1. Konfiguracja serwera aplikacyjnego pod kątem obsługi baz danych

Ja na swoje potrzeby używałem Postgres. Państwo możecie użyć dowolnej innej. Wymagane jest tylko użycie dedykowanego konektora.

Do konfiguracji JPA z bazą Postgres niezbędny będzie PostgreSQL JDBC konektor. Użyłem wersji 4.2 Driver, 42.2.5. Należy go ściągnąć ze strony producenta, zapisać lokalnie a następnie zainstalować na serwerze WildFly.

Kolejnym krokiem jest konfiguracja DataSource w WildFly tak aby można było korzystać z bazy Danych.

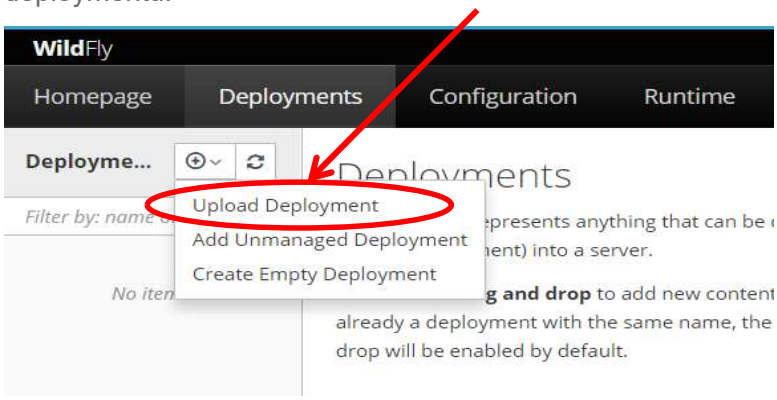
Istnieją trzy sposoby konfiguracji dostępu do bazy danych w WildFly:

1. Webowa konsola Administracyjna (localhost:9990)
2. CLI
3. Bezpośrednio edytując plik standalone.xml

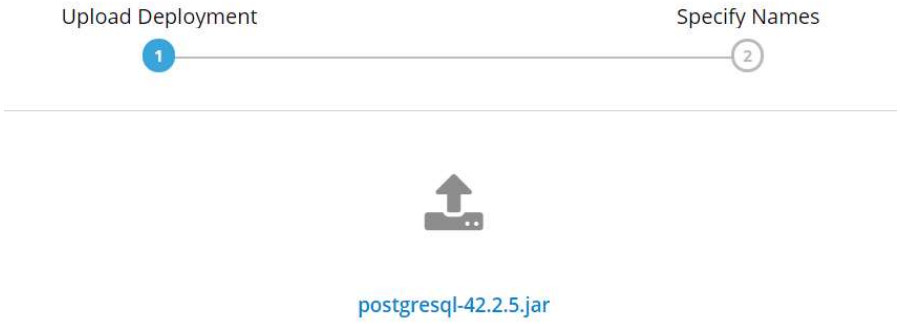
Zademonstruje Państwu każdy z nich – Państwo na lab wykonacie dowolną z nich.

1. Podejście przy użyciu Webowej Konsoli Administracyjnej

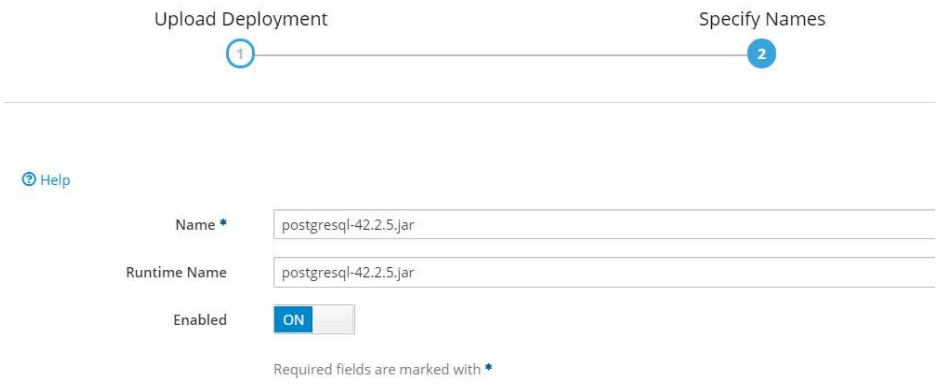
Zaczynamy od załadowania do serwera sterownika PostgresJDBC. Zrobimy to za pomocą deploymentu.



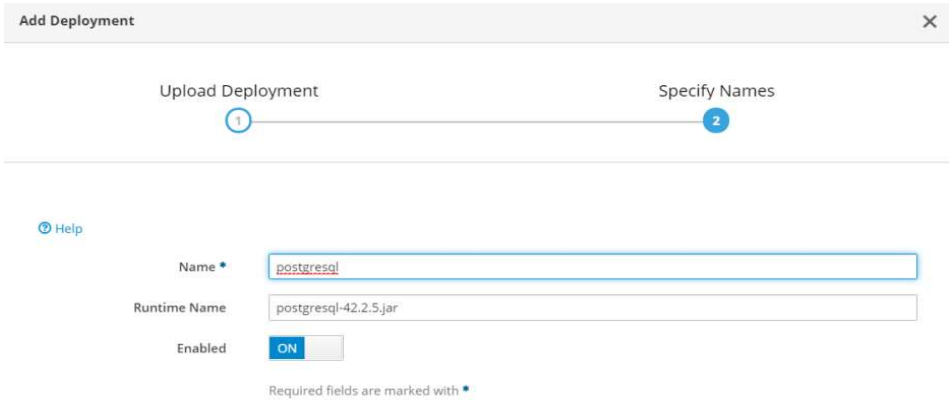
Po wybraniu zaznaczonej pozycji pojawia się kreator ładowania i konfiguracji sterownika. Wybieramy plik sterownika i przechodzimy do kolejnego kroku.



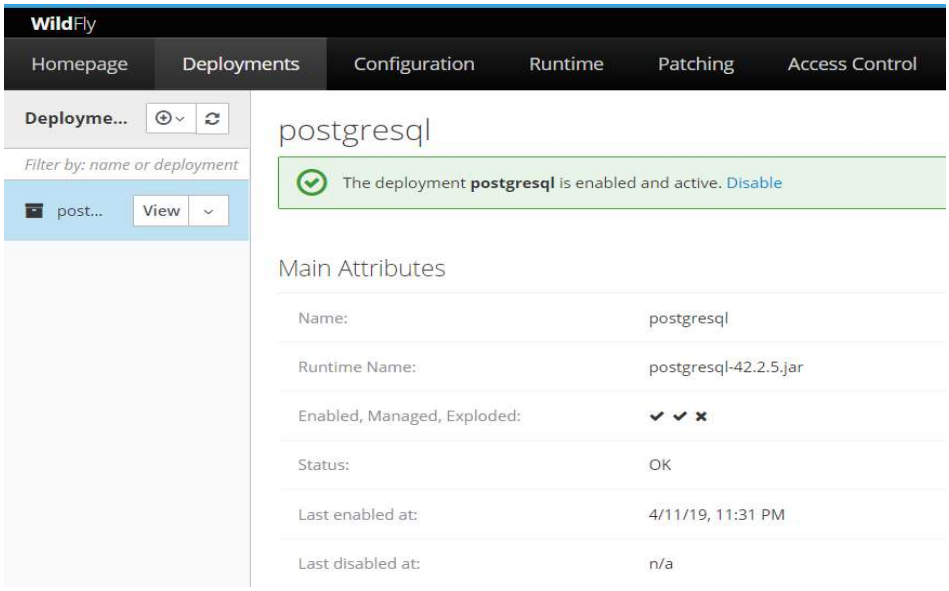
Automatycznie na podstawie nazwy pliku ustawiane są wymagane nazwy.



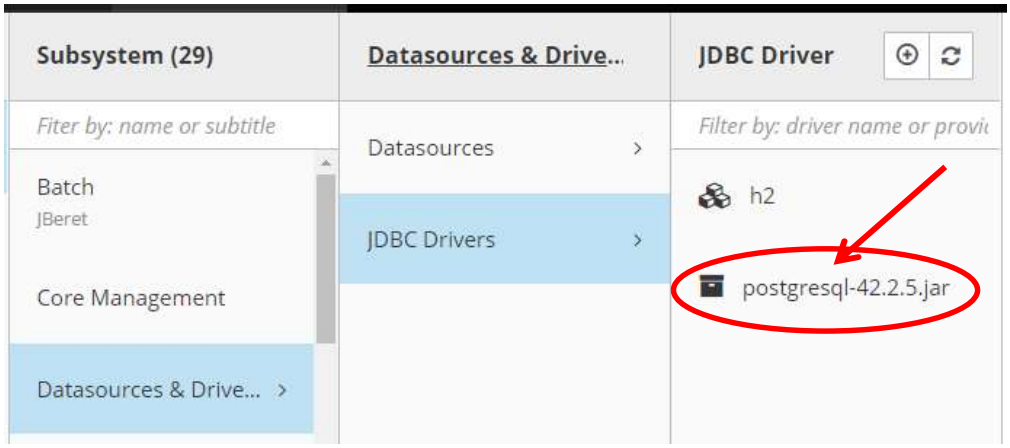
Modyfikujemy nazwę Name (nazwa dla użytkownika) na bardziej czytelną



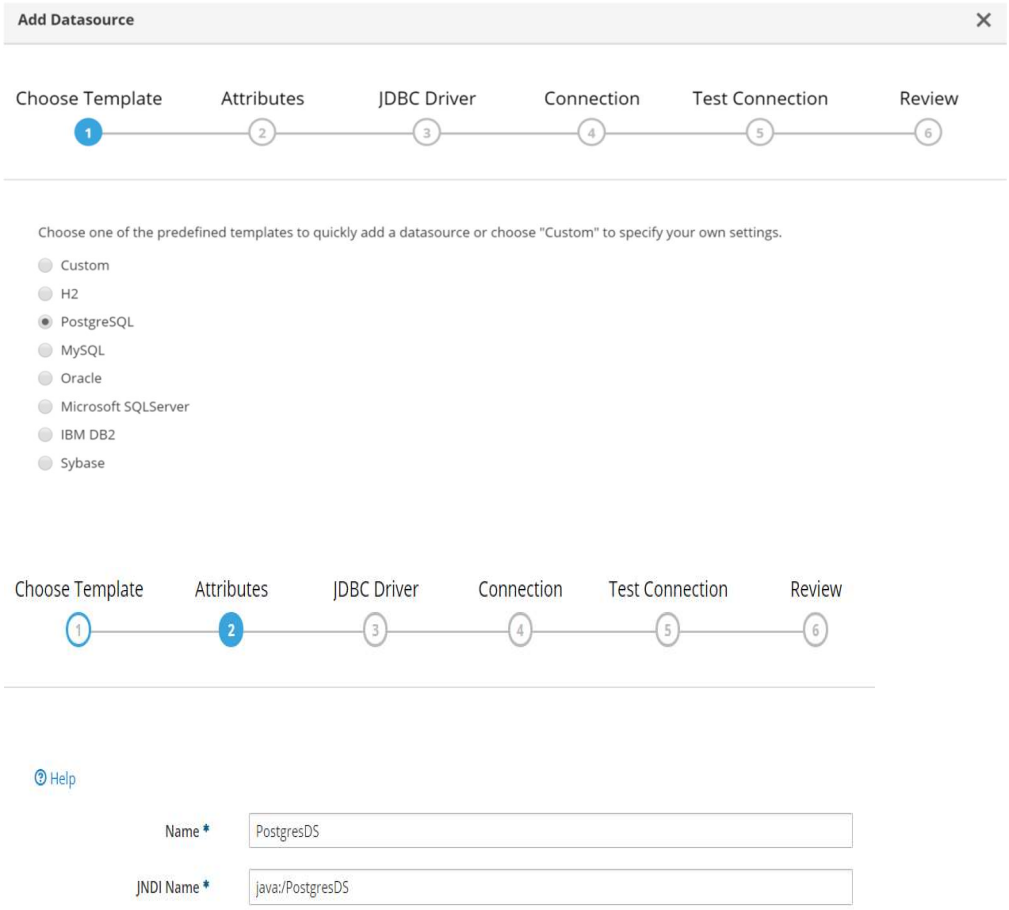
Proces Deploymentu zakończył się sukcesem, czego dowodem jest poniższy ekran.



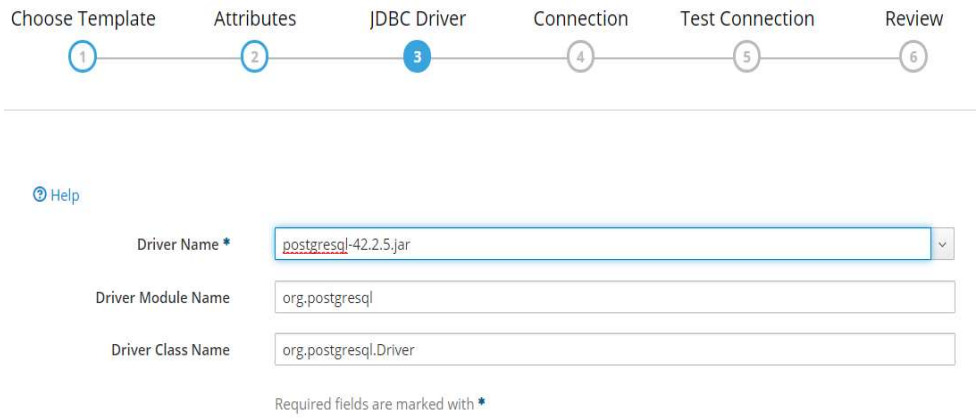
Po przejściu do sekcji Configuration -> SubSystem-> DataSource&Drivers widzimy ze pojawił się w niej nasz zainstalowany driver.



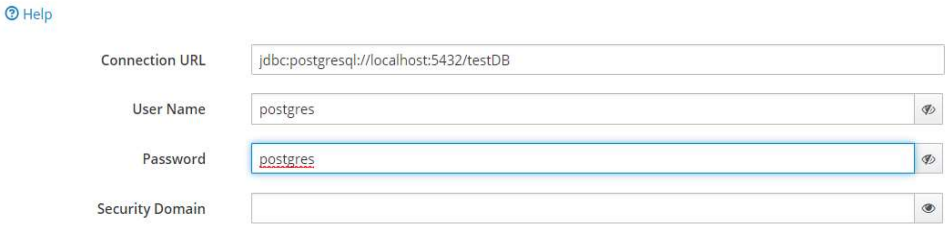
Rozpoczynamy proces konfigurowania DataSource za pomocą kreatora.



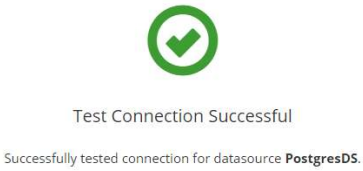
W kroku 3 nazwę Drivera wybieramy z listy rozwijalnej (są tam nazwy zarejestrowanych sterowników).



W kolejny kroku podajemy lokalizacje bazy z którą chcemy się łączyć + dane do autentykacji.



W kolejnym kroku weryfikujemy czy mamy połączenie bazą danych.



Cała nasza konfiguracja wygląda wiec tak jak na ekranie poniżej:

Z ciekawość sprawdźmy jeszcze jak wygląda odpowiedni fragment pliku standalone.xml

```
<subsystem xmlns="urn:jboss:domain:datasources:5.0">
    <datasources>
        <datasource jndi-name="java:jboss/datasources/ExampleDS" pool-name="ExampleDS" enabled="true" use-java-context="true" .
            <connection-url>jdbc:h2:mem:test;DB_CLOSE_DELAY=-1;DB_CLOSE_ON_EXIT=FALSE</connection-url>
            <driver>h2</driver>
            <security>
                <user-name>sa</user-name>
                <password>sa</password>
            </security>
        </datasource>
        <datasource jndi-name="java:/PostgresDS" pool-name="PostgresDS">
            <connection-url>jdbc:postgresql://localhost:5432/testDB</connection-url>
            <driver-class>org.postgresql.Driver</driver-class>
            <datasource-class>org.postgresql.ds.PGSimpleDataSource</datasource-class>
            <driver>postgresql-42.2.5.jar</driver>
            <security>
                <user-name>postgres</user-name>
                <password>postgres</password>
            </security>
            <validation>
                <valid-connection-checker class-name="org.jboss.jca.adapters.jdbc.extensions.postgres.PostgreSQLValidConnectio
                <background-validation>true</background-validation>
                <exception-sorter class-name="org.jboss.jca.adapters.jdbc.extensions.postgres.PostgreSQLExceptionSorter"/>
            </validation>
        </datasource>
```

W ten sposób udało się nam skonfigurować serwer do współpracy z konkretną bazą danych.

UWAGA!!!. Ja do autentykacji użyłem standardowego konta Postgresowego. Oczywiście można użyć dowolnego innego konta, które zostało stworzone w Postgresie.

Przy próbie polaczenia możecie Państwo dostawać informacje o błędzie. Błąd nie wynika z niepoprawnej konfiguracji po stronie Serwera Aplikacyjnego a z niepoprawnie skonfigurowanej polityki autoryzacji Postgresa – nie obsługującej poprawnie kodowania md5.

Należy samodzielnie aktywować odpowiednie metody autentykacji w pliku - Postgres authentication config (pg_hfa.conf).

Zakładam ze ta wiedza jest Państwu znana – skoro mieliście Państwo Bazy Danych.

.

2. Podejście przy użyciu CLI

Uruchmiamy CLI **-> jboss-cli.bat**

Łączymy się z serwerem aplikacyjnym i rozpoczynamy zabawę:

Potrzebujemy stworzyć nowy moduł do obsługi Postgres. Możemy to zrobić bezpośredni z linii komend wykonując następującą komendą

module add --name=org.postgres --resources=/tmp/postgresql-42.2.5.jar --dependencies=javax.api,javax.transaction.api

gdzie tmp/postgres….. miejsce w katalogu bin gdzie umieściłem ściągnięty konektor jdbc.

Kolejnym krokiem jest zarejestrowanie stworzonego modułu z poziomu konsoli.

/subsystem=datasources/jdbc-driver=postgres:add(driver-name=postgres,driver-module-name=org.postgres,driver-class-name=org.postgresql.Driver)

```
[standalone@localhost:9990 /] module add --name=org.postgres --resources=/tmp/po
stgresql-42.2.5.jar --dependencies=javax.api,javax.transaction.api
[standalone@localhost:9990 /] /subsystem=datasources/jdbc-driver=postgres:add(dr
iver-name="postgres",driver-module-name="org.postgres",driver-class-name=org.pos
tgresql.Driver)
{"outcome" => "success"}

[standalone@localhost:9990 /]
```

Kolejnym krokiem jest stworzenie źródła danych na naszym serwerze. Nie jest to czynność zbyt skomplikowana, aczkolwiek należy podać kilka parametrów. W końcu musimy mieć jakąś bazę danych, do której źródło możemy skonfigurować : >

```
[standalone@localhost:9990 /] data-source add --jndi-name=java:/PostgresDS --con
nection-url=jdbc:postgresql://localhost:5432/testDB --driver-name=postgres --use
r-name=postgres --password=postgres --name=PostgresPool
[standalone@localhost:9990 /]
```

Teraz wystarczy tylko za pomocą Webowej Consoli wykonać TestConnection połączenia z bazą.

3. Ręczna edycja plik standalone.xml

Proces konfiguracji DataSource można przeprowadzić bezpośrednio edytując plik standalone.xml.

Zaczynamy od stworzenia w katalogu modules podkatalogu org\postgres\main w którym umieścimy dwa pliku: Driver JDBC dla postgres oraz modul.xml, który proszę samodzielnie wyedytować tak jak poniżej:

```xml
<?xml version='1.0' encoding='UTF-8'?>

<module xmlns="urn:jboss:module:1.1" name="org.postgres">

    <resources>
        <resource-root path="postgresql-42.2.5.jar"/>
    </resources>

    <dependencies>
        <module name="javax.api"/>
        <module name="javax.transaction.api"/>
    </dependencies>
</module>
```

Następnie edytujemy plik standalone.xml i w rozszerzamy sekcje drivers o nową pozycje:

```
<drivers>
    <driver name="h2" module="com.h2database.h2">
        <xa-datasource-class>org.h2.jdbcx.JdbcDataSource</xa-datasource-class>
    </driver>
    <driver name="postgres" module="org.postgres">
        <driver-class>org.postgresql.Driver</driver-class>
    </driver>
</drivers>
```

A w sekcji datasource rejestrujemy nasze ustawienia do połączenia z bazą danych:

```
<datasources>
    <datasource jndi-name="java:jboss/datasources/ExampleDS" pool-name="ExampleDS" enabled="true" us
        <connection-url>jdbc:h2:mem:test;DB_CLOSE_DELAY=-1;DB_CLOSE_ON_EXIT=FALSE</connection-url>
        <driver>h2</driver>
        <security>
            <user-name>sa</user-name>
            <password>sa</password>
        </security>
    </datasource>
    <datasource jndi-name="java:/PostgresDS" pool-name="PostgresPool">
        <connection-url>jdbc:postgresql://localhost:5432/testDB</connection-url>
        <driver>postgres</driver>
        <security>
            <user-name>postgres</user-name>
            <password>postgres</password>
        </security>
    </datasource>
```

Teraz wystarczy tylko przejść do aplikacji Web Consola aby przeprowadziać TestConnection. Powinien się zakończyć sukcesem.

2. Tworzymy aplikacje testowa.

Tworzymy projekt typu maven i definiujemy wymagane zależności.

<!-- https://mvnrepository.com/artifact/org.clojure/java.jdbc -->

<dependency>

<groupId>org.clojure</groupId>

<artifactId>java.jdbc</artifactId>

<version>0.7.9</version>

</dependency>

Oraz

<!-- https://mvnrepository.com/artifact/org.hibernate/hibernate-core -->

<dependency>

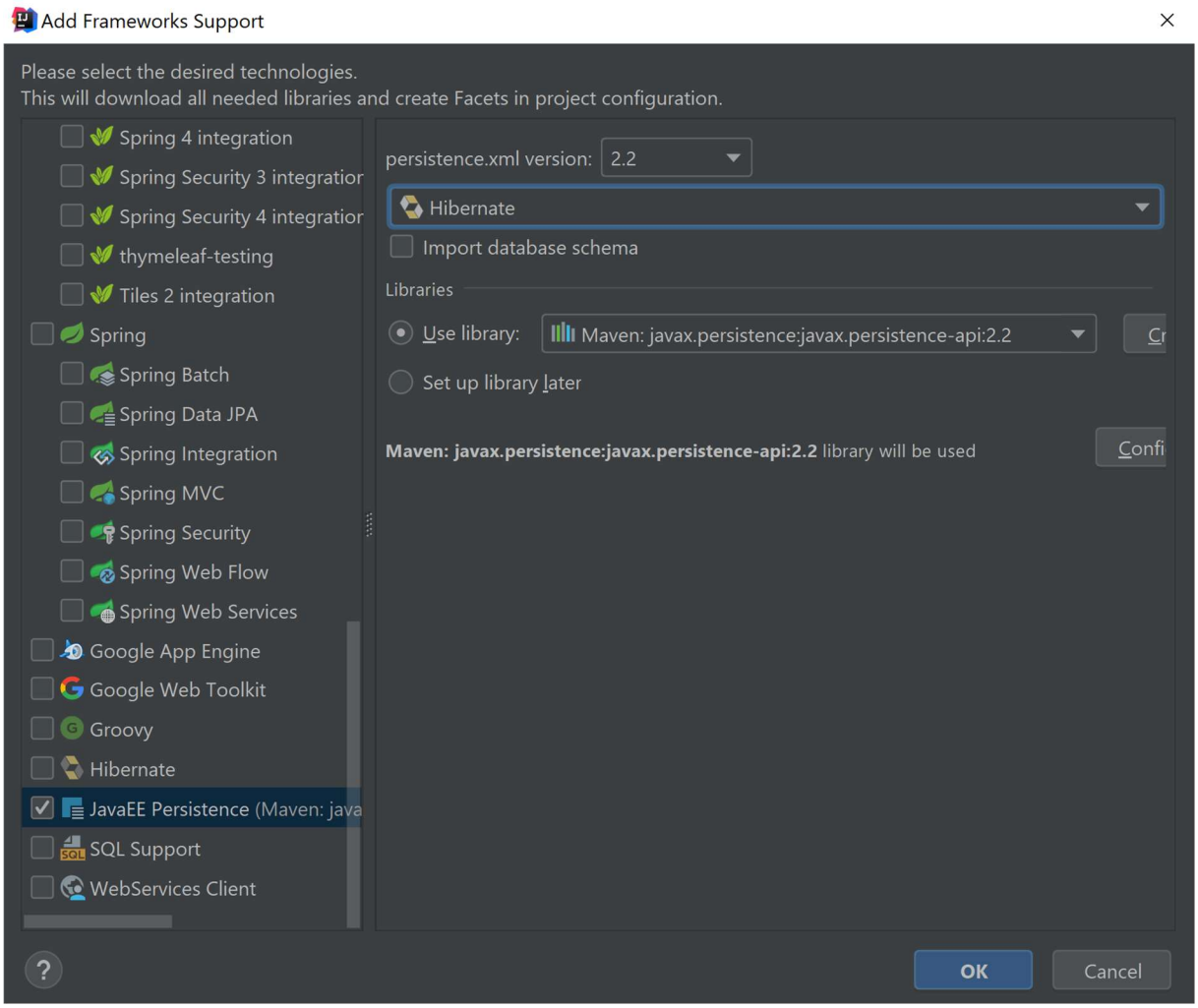
<groupId>org.hibernate</groupId>

<artifactId>hibernate-core</artifactId>

<version>5.4.2.Final</version>

</dependency>

Następnie tworzymy ręcznie lub generujemy poprzez dodanie obsługi JavaEE Persistence plik **persistence.xml**



Plik może mieć dwie postaci:

// Konfiguracja pliku persistance.xml do współpracy z dataSource z wildFly

```
<persistence>
  <persistence-unit name="JPA-Zajecia">
    <provider>org.hibernate.jpa.HibernatePersistence</provider>
    <jta-data-source>java:/PostgresDS</jta-data-source>        // nazwa naszego dataSource
    <properties>
      property name="hibernate.dialect"    value="org.hibernate.dialect.PostgreSQLDialect"/>
      <property name="hibernate.hbm2ddl.auto" value="create-drop"/>
    </properties>
  </persistence-unit>
</persistence>


//  lub wersja z ręczną konfiguracja dostępu do bazy danych
// postgres
<persistence xmlns=http://xmlns.jcp.org/xml/ns/persistence
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
          http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd"    version="2.1">


   <persistence-unit name="JPA-Zajecia" transaction-type="RESOURCE_LOCAL">
    <properties>
      <property name="javax.persistence.jdbc.driver" value="org.postgresql.Driver" />
     <property name="javax.persistence.jdbc.url" value="jdbc:postgresql://localhost:5432/testDB" />
       <property name="javax.persistence.jdbc.user" value="postgres" /> <!-- DB User -->
       <property name="javax.persistence.jdbc.password" value="12345" /> <!-- DB Password -->
       <property name="hibernate.dialect" value="org.hibernate.dialect.PostgreSQLDialect"/>
         <property name="hibernate.hbm2ddl.auto" value="update" />
         <property name="hibernate.show_sql" value="true" /> <!-- Show SQL in console -->
```

```
        <property name="hibernate.format_sql" value="true" /> <!-- Show SQL formatted -->
    </properties>
  </persistence-unit>
</persistence>
```

Następnie napisz klasę POJO np Student:

```
import java.util.Date;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.Table;
import javax.persistence.Temporal;
import javax.persistence.TemporalType;

@Entity
@Table( name = "student" )
public class Student {
    private int id;
    private String imie;
    private String nazwisko;
    private Date dodanieData;

    public Student() {
        super();
    }

    @Id
    @GeneratedValue
    @Column(name = "id", nullable=false)
    public int getId() {
        return id;
    }

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

    public String getImie() {
        return imie;
    }

    @Column(name = "imie", nullable=false)
    public void setImie(String imie) {
        this.imie = imie;
    }

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

    public void setNazwisko(String nazwisko) {
        this.nazwisko = nazwisko;
    }

    @Temporal(TemporalType.TIMESTAMP)
    @Column(name = "created_at", nullable=true)
    public Date getDodanieData() {
        return dodanieData;
    }

    public void setDodanieData(Date dodanieData) {
        this.dodanieData = dodanieData;
```

A następnie zwykłą klasę w Java do generacji studentów do bazy.

```
public class Main {

        public static void main(String[] args) {

                EntityManagerFactory factory = Persistence.createEntityManagerFactory("JPA-Zajecia");

                EntityManager em = factory.createEntityManager();


                try {

                        Student s1 = new Student("adam", "nowak", new Date(), new Date());

                        Student s2 = new Student("marek", "kowalski", new Date(), new Date());

                        Student s3 = new Student("anna", "marchewka", new Date(), new Date());
```

```
                em.getTransaction().begin();
                em.persist(s1);
                em.persist(s2);
                em.persist(s3);
                em.getTransaction().commit();
                System.out.println("Zapisano w bazie: " + s1);
                System.out.println("Zapisano w bazie: " + s2);
                System.out.println("Zapisano w bazie: " + s3);
            }
            catch(Exception e) {
                System.err.println("Blad przy dodawaniu rekordu: " + e);
            }
        }
}
```

W celu weryfikacji czy zapis się udał napiszmy jeszcze jedną klasę do odczytania zawartości bazy.

```
public class Main2 {
    public static void main(String[] args) {
        EntityManagerFactory factory = Persistence.createEntityManagerFactory("JPA-Zajecia");
        EntityManager em = factory.createEntityManager();

        try {
            Query q = em.createQuery("FROM Student", Student.class);
            List<Student> students = q.getResultList();
            for (Student s : students)
                System.out.println(s);
        }
        catch(Exception e) {
            System.err.println("Blad przy pobieraniu rekord—w: " + e);
        }
```

```
        }
}
```

Przetestuj czy aplikacja działa poprawnie.

Zadanie do oddania

Stwórz aplikacje webowa pozwalającą na zarządzanie książkami. Aplikacja umożliwia podgląd, dodawanie, usuwanie i modyfikacje pozycji katalogu. Katalog zawiera następujące pozycje: nazwisko autora, imię, tytuł, numer ISBN, rok wydania, cena.

Można wykorzystać warstwę prezentacyjna wykonaną w ramach lab 3.