

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 specyfikację JPA.

Aby korzystać z JPA niezbędna jest konfiguracja połączenia 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 bazy 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.

Potrzebujemy stworzyć moduł jboss'owy. Możemy to zrobić bezpośrednio z linii komend wykonując następującą komendę

jboss-cli.bat

potem z linii komend wpisujemy:

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

gdzie tmp/postgres..... jest miejscem w katalogu bin gdzie umieściłem ściągnięty konektor.

Kolejnym krokiem jest zarejestrowanie stworzonego modułu z poziomu konsoli. Uruchamiamy serwer jak poprzednio, następnie uruchamiamy skrypt **jboss-cli.bat**, po którego uruchomieniu będziemy odłączeni, co naprawiamy komendą **connect** : Gdy jesteśmy już podłączeni, pozostaje nam jedynie wpisać następujące polecenie:

```
1 /subsystem=datasources/jdbc-driver=postgres:add(driver-name=postgres,driver-module-name=com.postgres.driver,driver-
  class-name=com.postgres.jdbc.Driver)
```

Kolejnym krokiem jest stworzenie źródła danych na naszym serwerze. Nie jest to czynność zbyt skomplikowana, aczkolwiek potrzebujemy do niej zainstalowanego Postgres. W końcu musimy mieć jakąś bazę danych, do której źródło możemy skonfigurować : >

Najłatwiej będzie nam to osiągnąć poprzez konsolę webową, która znajduje się, jak już poznaliśmy na lab 1, pod adresem localhost:9990.

Wchodzimy w Configuration :



Configuration

Configure subsystem settings

[Create a Datasource](#)

[Start](#)

Define a datasource to be used by deployed applications. The proper JDBC driver must be deployed and registered.

1. Select the Datasources subsystem
2. Add a Non-XA or XA datasource
3. Use the 'Create Datasource' wizard to configure the datasource settings

[Create a JMS Queue](#)

[Start](#)

Jak widać sterownik jest już zainstalowany poprawnie:

Datasources & Drivers	JDBC Driver
Datasources	Filter by: driver name or provider type
JDBC Drivers	<div><div>h2</div><div>postgres Remove</div></div>

postgres

The JDBC driver is provided by module `org.postgres`.

Main Attributes

Driver Class Name:	org.postgresql.Driver
Driver Datasource Class Name:	
Driver XA Datasource Class Name:	
Driver Version:	42.2
JDBC Compliant:	false

Pozostaje skonfigurować datasource:

Przechodzimy przez ścieżkę jak pokazane poniżej:

Add Datasource



Choose Template

Attributes

JDBC Driver

Connection

Test Connection

Review

1

2

3

4

5

6

Choose one of the predefined templates to quickly add a datasource or choose "Custom" to specify your own settings.

- ☐ Custom
- ☐ H2
- ☒ PostgreSQL
- ☐ MySQL
- ☐ Oracle
- ☐ Microsoft SQLServer
- ☐ IBM DB2
- ☐ Sybase

Add Datasource



Choose Template

Attributes

JDBC Driver

Connection

Test Connection

Review

1

2

3

4

5

6

[Help](#)

Name *

PostgresDS

JNDI Name *

java:/PostgresDS

Required fields are marked with *

Add Datasource



Choose Template

Attributes

JDBC Driver

Connection

Test Connection

Review

1

2

3

4

5

6

[Help](#)

Driver Name *

postgresql

Driver Module Name

org.postgresql

Driver Class Name

org.postgresql.Driver

Required fields are marked with *

Add Datasource



Choose Template

1

Attributes

2

JDBC Driver

3

Connection

4

Test Connection

5

Review

6

[Help](#)

Driver Name *

postgresql

Invalid driver name

Driver Module Name

org.postgresql

Driver Class Name

org.postgresql.Driver

Required fields are marked with *

Add Datasource



Choose Template

1

Attributes

2

JDBC Driver

3

Connection

4

Test Connection

5

Review

6

[Help](#)

Connection URL

jdbc:postgresql://localhost:5432/postgres

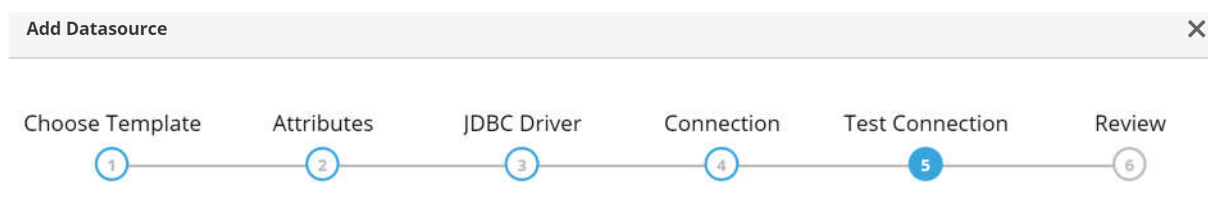
User Name

.....

Password

.....

Security Domain



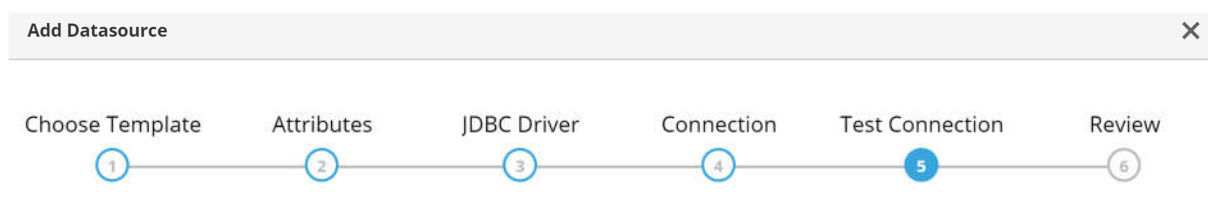
On this page you can test the connection of your datasource.

Please note that testing the connection changes the semantics of this wizard:

- If you press **Test Connection** for the **first time**, the datasource is **created in advance**.
- If you **go back** and change settings, this will **modify** the newly created datasource. Please note that you cannot change the name and JNDI bindings once the datasource has been created.
- If you **cancel** the wizard, the datasource will be **removed** again. This might require a reload of the server.

If you choose to continue without testing the connection, the datasource will be created after finishing the wizard.

Test Connection



Test Connection Successful

Successfully tested connection for datasource **PostgresDS**.

Jak widać udało się nam połączyć z baza danych. Mamy poprawnie predefiniowane dataSource.

Datasources & Drivers

Datasources

JDBC Drivers

Datasource

Filter by: name, xa, .../disabled, deployment

ExampleDS

PostgresDS

View

PostgresDS

Datasource

The datasource **PostgresDS** is enabled. [Disable](#)

Main Attributes

JNDI Name:

java:/PostgresDS

Driver Name:

postgres

Connection URL:

jdbc:postgresql://localhost:5432/postgres

Enabled:

true

Statistics Enabled:

false

2. Tworzymy aplikacje testowa.

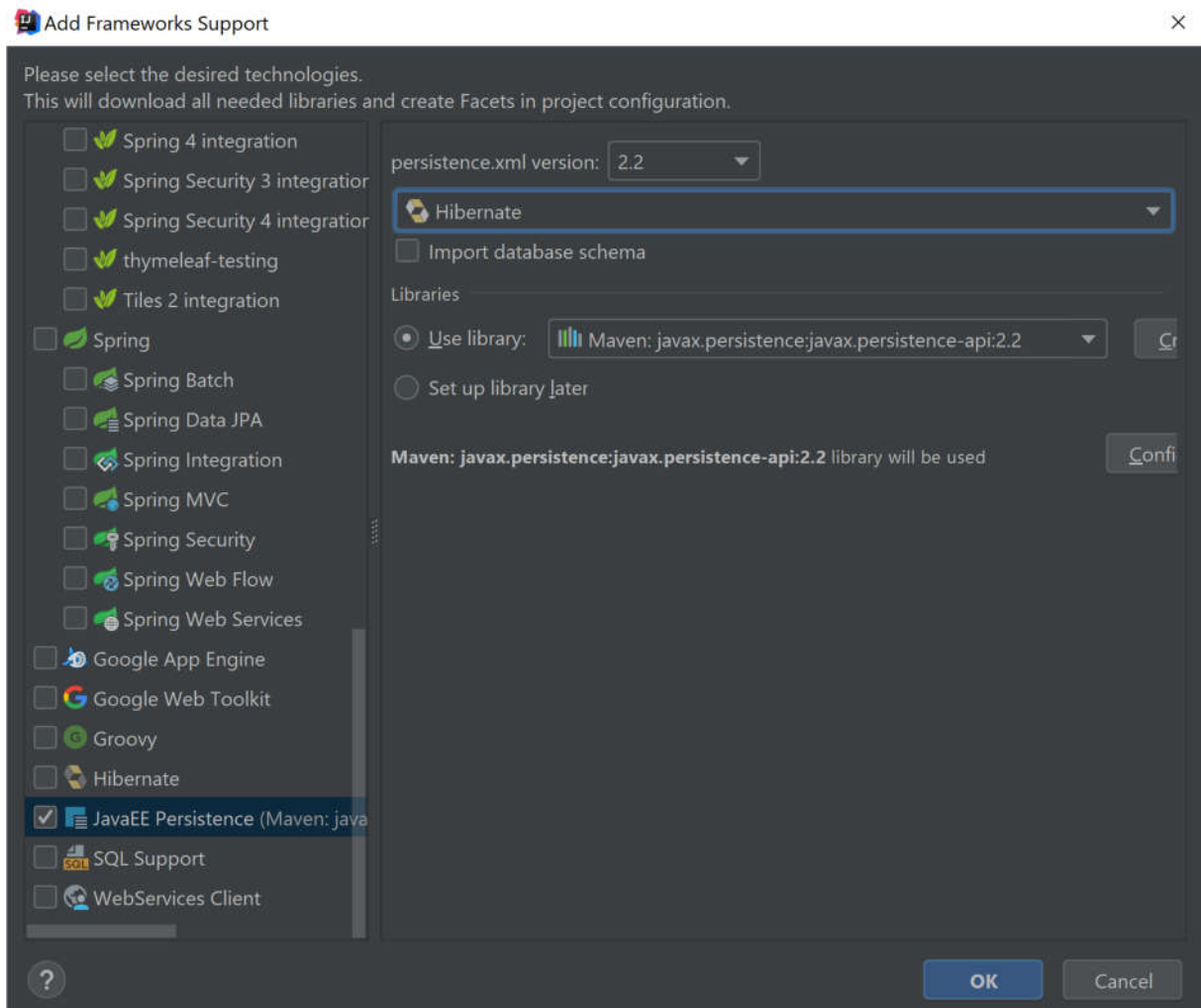
Tworzymy projekt typu maven i definiujemy wymagane zależności.

```
<!-- 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 persistence.xml do współpracy z dataSource z wildFly

```
<persistence>
```

```
  <persistence-unit name="JPA-Zajecia">
```

```
    <provider>org.hibernate.jpa.HibernatePersistence</provider>
```

```
    <jta-data-source>java:/PosgresDS</jta-data-source>    // nazwa naszego dataSource
```

```
  <properties>
```

```
    property name="hibernate.dialect" value="org.hibernate.dialect.HSQLDialect"/>
```

```
    <property name="hibernate.hbm2ddl.auto" value="create-drop"/>
```

```
  </properties>
```

```
</persistence-unit>
```

</persistence>

// lub wersja z ręczną konfiguracją 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/dbName" />
```

```
<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("Bład 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("Bład przy pobueraniu rekord—w: " + e);

        }

    }

}

```

Przetestuj czy aplikacja działa poprawnie.

Zadanie do oddania

Stwórz aplikację webową pozwalającą na zarządzanie książkami. Aplikacja umożliwi podgląd, dodawanie, usuwanie i modyfikację pozycji katalogu. Katalog zawiera następujące pozycje: nazwisko autora, imię, tytuł, numer ISBN, rok wydania, cena.

Można wykorzystać warstwę prezentacyjną wykonaną w ramach lab 3.