



Chapter 6

Adding a Database with JPA

Wykonali:

- Filip Glinkowski
- Artur Gurak
- Jakub Jarecki
- Bartosz Knapik
- Tomasz Kot
- Emilia Kwolek



Wstęp

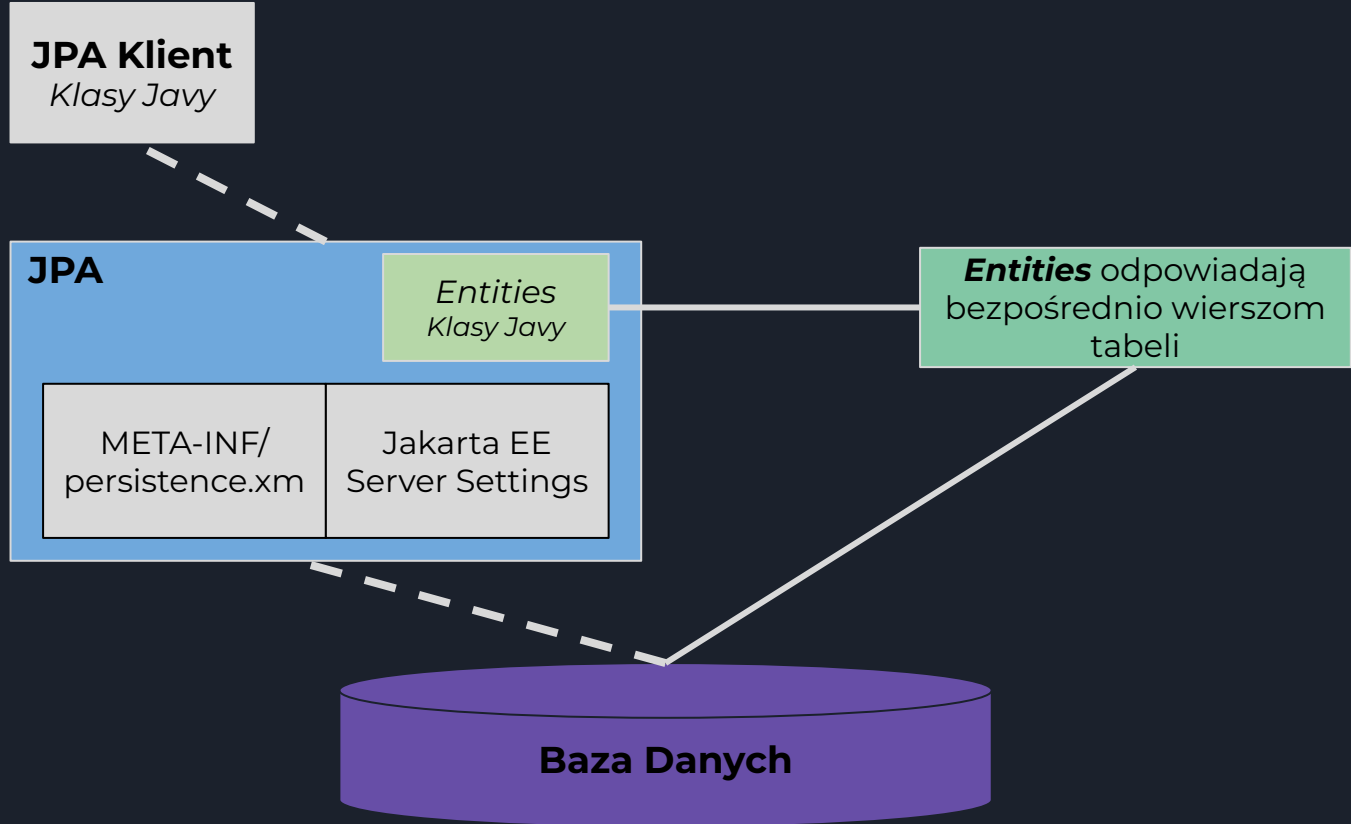
- JPA (Java Persistence API) - dedykowana technologia dostępu do relacyjnych baz danych dla Jakarta EE
- Interfejs pomiędzy tabelami SQL a obiektami Javy
- Zabezpieczanie konwersji danych przechowywanych w tabelach, z wyszczególnieniem:
 - wartości *null*
 - typów numerycznych
 - wartości logicznych
 - wartości daty i czasu



Abstracting Away Database Access with JPA

- Zapewnienie abstrakcyjnego dostępu do bazy danych pozwalającego na:
 - Mapowanie odpowiedzi zapytań bazodanowych na obiekty Javy
 - Mapowanie obiektów Javy do postaci zapytań bazodanowych
 - Ukrycie informacji o połączeniu (właściwości połączenia, użytkowników, hasła)
- Główną klasą JPA zapewniającą ten dostęp jest *EntityManager*, jej konfiguracja zawarta jest z pliku *persistence.xml* oraz ustawieniach serwera Jakarta EE
- Klasy odzwierciedlające wiersze tabel oznaczane są jako klasy *entities*

Abstracting Away Database Access with JPA






Setting Up a SQL Database

Serwer Jakarta EE 8 Glassfish tworzy bazę danych Derby (lub JavaDB) w celach deweloperskich. Aby ją uruchomić należy wykonać następujące polecenia:

```
cd [GLASSFISH_INST]
cd bin
./asadmin start-database
```




Kontynuując przykład aplikacji Calipso, po uruchomieniu bazy danych poleceniem [GLASSFISH_INST]/javadb/bin/ij wchodzimy do konsoli i tworzymy bazę danych oraz ustalamy hasło dla użytkownika:

```
connect
'jdbc:derby://localhost:1527/calypso;create=true;user=user0';

call SYSCS_UTIL.SYSCS_CREATE_USER('user0', 'pw715');
```


Po czym uruchamiamy bazę danych ponownie poleceniami:

```
cd [GLASSFISH_INST]
cd bin
./asadmin stop-database && ./asadmin start-database
```



Po inicjalizacji bazy możemy zalogować się do niej z poziomu konsoli (przy pomocy pliku `ij`):

```
connect  
'jdbc:derby://localhost:1527/calypso;create=true;user=user0;password=pw715';
```




W celu prostego połączenia z bazą danych tworzymy dwa zasoby - zbiór połączeń *connection pool* oraz zasób JDBC:

```
cd [GLASSFISH_INST]
cd bin

./asadmin create-jdbc-connection-pool \
--datasourceclassname org.apache.derby.jdbc.ClientDataSource \
--restype javax.sql.DataSource \
--property \
portNumber=1527:password=pw715:user=user0:
serverName=localhost:databaseName=calypso:
securityMechanism=3 \
Calypso

./asadmin create-jdbc-resource \
--connectionpoolid Calypso jdbc/Calypso
```

Po inicjalizacji bazy możemy tworzyć zapytania przy pomocy dowolnego klienta, na przykład na potrzeby przykładowej aplikacji Calypso:

```
CREATE TABLE MEMBER (  
    ID INT NOT NULL,  
    LAST_NAME VARCHAR(128) NOT NULL,  
    FIRST_NAME VARCHAR(128) NOT NULL,  
    BIRTHDAY CHAR(10) NOT NULL,  
    PRIMARY KEY (ID));  
  
INSERT INTO MEMBER (ID, LAST_NAME, FIRST_NAME, BIRTHDAY)  
VALUES (-3, 'Smith', 'John', '1997-11-05'),  
       (-2, 'Tender', 'Linda', '1997-11-05'),  
       (-1, 'Quast', 'Pat', '2003-04-13');  
  
CREATE SEQUENCE MEMBER_SEQ start with 1 increment by 50;
```



Adding EclipseLink as ORM

Jednym z wymogów działania JPA jest dodanie do projektu biblioteki ORM (Object Relational Mapping).

W książce wybrano **EclipseLink**, ponieważ jest referencyjną implementacją JPA 2.2.



Aby dodać EclipseLink należy:

1. Pobrać instalator EclipseLink (np. ZIP);
2. Rozpakować archiwum na komputerze;
3. Utworzyć nowy projekt “calypso-jpa”;
4. Dodajemy do Maven’a:

```
Group-Id: book.jakarta8  
Artifact-Id: calypso-jpa  
Version: 0.0.1-SNAPSHOT
```

5. Zmień nazwę pakietu na “book.jakarta8.calypsojpa”;
6. Kopiujemy wszystkie pliki do nowego projektu Calypso i sprawdzamy czy działa.



Adres do przeglądarki:

<http://localhost:8080/calypsojpa/static/main.html>

7. Tworzymy folder "src/main/webapp/WEB-INF/lib" i kopiujemy pliki bibliotek w następujący sposób:

`eclipselink.jar`

`jakarta.persistence_x.y.z.jar`

`org.eclipse.persistence.jpa.modelgen_*.jar`

`org.eclipse.persistence.jpars_*.jar`

z dystrybucji EclipseLink do folderu "WEB-INF/lib". Dodajemy te JAR-y do zakładki "Libraries" we właściwościach projektu (sekcja: "Java Build Path").

8. Tworzymy plik src/main/resources/META-INF/persistence.xml:

```
<persistence
  xmlns=
    "http://java.sun.com/xml/ns/persistence"
  xmlns:xsi=
    "http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation=
    "http://java.sun.com/xml/ns/persistence
    persistence_1_0.xsd"
  version="1.0">
  <persistence-unit name="default"
    transaction-type="JTA">
    <jta-data-source>jdbc/Calypto</jta-data-source>
    <exclude-unlisted-classes>
      false
    </exclude-unlisted-classes>
    <properties />
  </persistence-unit>
</persistence>
```

Utworzony plik to jest centralnym plikiem konfiguracyjnym dla JPA.
W taki sposób aplikacja zostaje połączona z bazą danych.



Adding Data Access Objects

Data Access Object (DAO) - klasa Javy, która hermetyzuje operacje CRUD bazy. Klient DAO nie musi wiedzieć jak ono działa, musi jedynie zadbać o jego funkcjonalność.

Kontynuując na przykładzie klasy Calypso - klasa DAO zostaje wstrzyknięta anotacją @EJB (EJB - obiekt kontrolowany przez serwer). Klasa DAO została nazwana MemberDAO.

```
@EJB private MemberDAO members;
```

Dodatkowo została dodana klasa entity Member. Jest to klasa reprezentująca członka, o własnościach: imię, nazwisko, data urodzin i ID. To obiekt zwracany przez JPA po wyszukanie konkretnego rekordu z bazy, bądź wysyłany do bazy w celu dodania bądź zmiany.

Member DAO

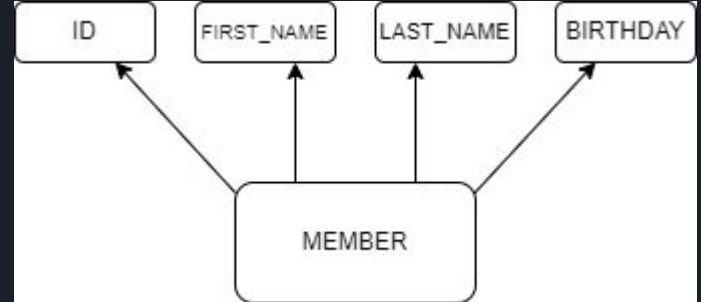
```
@Singleton
public class MemberDAO {
    @PersistenceContext
    private EntityManager em;
    public List<Member> allMembers() {
        TypedQuery<Member> q = em.createQuery(
            "SELECT m FROM Member m", Member.class);
        List<Member> l = q.getResultList();
        return l;
    }
    public Member getMember(int id) {
        return em.find(Member.class, id);
    }
    public int newMember(String lastName,
        String firstName, String birthday) {
        Member m = new Member();
        m.setFirstName(firstName);
        m.setLastName(lastName);
        m.setBirthday(birthday);
        em.persist(m);
        em.flush(); // needed to get the ID
        return m.getId();
    }
    public void updateMember(String lastName,
        String firstName, String birthday, int id)
    {
        Member m = em.find(Member.class, id);
        m.setLastName(lastName);
        m.setFirstName(firstName);
        m.setBirthday(birthday);
        em.persist(m);
    }
    public void deleteMember(int id) {
        Member m = em.find(Member.class, id);
        em.remove(m);
    }
}
```

Adding Entities

Encja to reprezentacja wiersza tabeli jako obiektu.

Jeśli pomyślimy o tabeli MEMBER, przykładową encją będzie coś, co ma jedno imię, jedno nazwisko, jedną datę urodzenia i jedno ID.

Oczywiście odpowiada to klasie Java z polami firstName, lastName, birthday oraz id.



```
public class Member {  
    private int id; // + getter/setter  
    private String lastName; // + getter/setter  
    private String firstName; // + getter/setter  
    private String birthday; // + getter/setter  
}
```




Adding Entities

W celu dokonania połączenia z bazą danych, niezbędne jest dodanie meta-informacji. Są to dane informujące, że dana klasa to klasa encja, jaka jest nazwa tworzonej tabeli, nazwy kolumn, określenie kolumny z kluczem głównym, specyfikację generatora unikalnych identyfikatorów oraz ograniczenia wartości pól w tabeli. Jak to zwykle bywa w Javie, do takich metainformacji używamy adnotacji.

Adding Entities | Example

● @Entity

Zaznacza encję, aby JPA wiedziało, że jest to klasa encji.

● @Table

Używany do określenia nazwy tabeli. Jeśli nie określimy nazwy, będzie nią nazwa klasy.

● @SequenceGenerator

Używany do określenia generatora sekwencji dla unikalnych ID.

● @Id

Określa, że odpowiednie pole będzie unikalnym ID encji.

● @GeneratedValue

Mówi, że nowe encje będą automatycznie generować wartości dla tego pola.

● @Column

Używamy do określenia nazwy kolumny odpowiadającej temu polu. Jeśli nie zostanie określona, nazwa pola zostanie użyta jako nazwa kolumny.

● @NotNull

Warunek mówiący o tym, że dane pole nie może być puste.

● @Pattern

Warunek dla pól tekstowych, określająca, że pole musi odpowiadać określonemu wzorcowi.

```
package book.jakarta8.calypsojpa.jpaa;

import javax.persistence.*;
import javax.validation.constraints.*;

@Entity
@Table(name="MEMBER")
@SequenceGenerator(name="MEMBER_SEQ",
                   initialValue=1, allocationSize = 50)
public class Member implements Comparable<Member> {
    @Id
    @GeneratedValue(strategy = GenerationType.SEQUENCE,
                   generator="MEMBER_SEQ")
    @Column(name = "ID")
    private int id;

    @NotNull
    @Column(name = "LAST_NAME")
    private String lastName;

    @NotNull
    @Column(name = "FIRST_NAME")
    private String firstName;

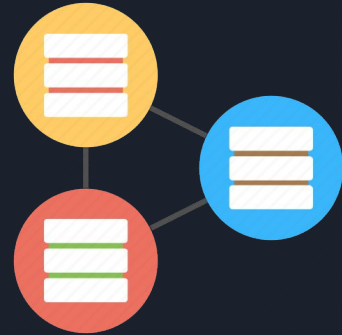
    @NotNull
    @Column(name = "BIRTHDAY", length = 10)
    @Pattern(regexp = "\\d{4}-\\d{2}-\\d{2}",
            message="Birthday format: yyyy-MM-dd.")
    private String birthday;

    @Override
    public int compareTo(Member o) {
        if(o.birthday.compareTo(birthday) != 0)
            return o.birthday.compareTo(birthday);
        // + getters and setters for all properties
    }
}
```

Adding Relations

Relacyjne bazy danych opierają się na relacyjnym modelu danych, którego podstawę stanowią relacje różnego typu, pozwalające w jednoznaczny sposób powiązać odpowiednie tabele.

JPA udostępnia rozwiązanie umożliwiające definiowanie wszystkich typów relacji w bardzo łatwy sposób . W tym celu wykorzystuje specjalne adnotacje, które dodawane są do klas encji.





Adding Relations | Example

Przykład: Zastosowanie adnotacji w celu utworzenia odpowiedniej relacji pomiędzy encjami.

Kontynuując przykład aplikacji Calypso: *Dodajemy tabelę "STATUS", która przechowuje status członkostwa (np. "Gold", "Platinum" itp.). Każdy członek może mieć od 0 do N wpisów statusu, czyli relacja jeden-do-wielu.*

Kolejne kroki:

1. Utworzenie tabeli "STATUS" oraz sekwencji generującej wartość ID statusu.
2. Utworzenie klasy encji "Status" odwzorowującej pola utworzonej powyżej tabeli
3. W klasie encji "Member" dodanie pola reprezentującego relację pomiędzy członkiem a statusem:

Adding Relations | Example - cont.

```
...
@JoinColumn(name = "MEMBER_ID")
@OneToMany(cascade =
CascadeType.ALL, orphanRemoval =
true)
private Set<Status> status;
...
```

@JoinColumn odnosi się do pola w encji "Status":

```
...
@NotNull
@Column(name = "MEMBER_ID") private
int memberId;
...
```

Na tym etapie mamy wszystko gotowe. Ponieważ encje zostały powiązane przy użyciu @OneToMany (CascadeType.ALL) - wszelkie operacje menadżera encji będą wykonane kaskadowo.