JPA

Spis treści

JPA *(Jakarta Persistence API)*

JPA *(Jakarta Persistence API)* - To pojęcie dotyczy już konkretnie świata Javy. Jest to **standard** określający, w jaki sposób program Java ma się komunikować z relacyjną bazą danych. Jak to zazwyczaj w przypadku Javy bywa, do określenia specyfikacji, czyli co i jak ma działać, stosowane są interfejsy. Tak samo jest w tym przypadku i standard *JPA* to zbór interfejsów, które określają jakie metody muszą zostać zaimplementowane.

Przypomnij sobie, że analogiczna sytuacja występowała przy komunikacji z bazami danych przy wykorzystaniu JDBC. Korzystaliśmy z interfejsów, ale konkretna ich implementacja była dostarczana przez sterownik bazy danych.

Definicja z Wikipedii wygląda tak:

> (...) a specification that describes the management of relational data in enterprise Java applications (...)

W wersji 3.0 nazwa **JPA** została zmieniona z ***Java Persistence API*** na ***Jakarta Persistence API***. Niby nic wielkiego, ale zmiana dotyczyła także nazw pakietów i

właściwości używanych do konfiguracji z ***javax**.persistence* na ***jakarta**.persistence*.

Zależność do Jakarta Persistence API:

```
// https://mvnrepository.com/artifact/jakarta.persistence/jakarta.persistence-api
implementation "jakarta.persistence:jakarta.persistence-api:$jakartaPersistenceApiVersion"
```

Zależność do Javax Persistence API:

```
// https://mvnrepository.com/artifact/javax.persistence/javax.persistence-api
implementation "javax.persistence:javax.persistence-api:$javaxPersistenceApiVersion"
```

Specyfikacja JPA definiuje:

- W jaki sposób mapować klasy (*encje*) do tabel bazodanowych (*adnotacje, pliki xml*),
- Interfejsy do operacji ***CRUD*** (***c**reate*, ***r**ead*, ***u**pdate*, ***d**elete*)
- Język (*JPQL - **J**akarta **P**ersistence **Q**uery **L**anguage*) i API do tworzenia zapytań SQL *Criteria API*,
- Strategie pracy z transakcjami, pobierania zależności między encjami i strategie optymalizacji

Przypomnijmy sobie czym było **CRUD**:

Obraz 1. Operacje CRUD

JPA vs JDBC

Jak możemy poukładać sobie w głowie czym jest **JPA**? Spróbujmy to odnieść do **JDBC**.

Jeszcze raz, jak możemy rozumieć **JDBC**, cytując IBM:

> Java™ database connectivity (JDBC) is the JavaSoft specification of a standard application programming interface (API) that allows Java programs to access database management systems. The JDBC API consists of a set of interfaces and classes written in the Java programming language.

Czyli **JDBC** jest standardem, który pozwala Javie rozmawiać z bazami danych. Zestawiając to z informacjami odnośnie **JPA**, można zestawić te dwa (JDBC i JPA) ze sobą w ten sposób:

JDBC to standard bezpośredniego łączenia się z bazą danych i uruchamiania na niej zapytań SQL, np. `SELECT * FROM ORDERS`. W wyniku wykonywania zapytań, możemy zwracać zestawy danych, które następnie możemy ręcznie obsługiwać w naszej aplikacji. Możemy również wykonywać zapytania **INSERT**, **DELETE** oraz **UPDATE**. Jest to jedna z technologii leżących u podstaw większości dostępu do baz danych Java (w tym dostawców JPA).

Jednym z problemów związanych z wykorzystaniem "tylko" JDBC jest to, że często będziemy musieli napisać dużo powtarzalnego kodu, który będzie określał mapowania pomiędzy encjami z bazy danych a obiektami napisanymi w Java. Przez to ulegamy takiej pokusie, żeby logika programu zaczynała się mieszać i przeplatać z zapytaniami do bazy danych.

Dla porównania, **JPA** to standard *mapowania relacyjnego obiektów*, czyli **ORM**. Technologie JPA pozwalają na mapowanie pomiędzy encjami bazodanowymi, a kodem Java w "prostszy" sposób. Stąd we wcześniejszych materiałach o bazach danych zostało wspomniane, że po omówieniu JDBC przejdziemy do omówienia "kolejnej warstwy abstrakcji". Narzędzia JPA dają taką możliwość, że deweloper nie musi pisać SQL "z palca", dużo rzeczy może zostać wykonanych automatycznie / magicznie / automagicznie. Praca z narzędziami JPA będzie się skupiała na określeniu w klasach Java jak ma wyglądać analogiczna struktura encji bazodanowej - do tego przejdziemy.

Najbardziej znanym dostawcą JPA jest Hibernate. Pod maską Hibernate i większość innych dostawców JPA tworzy za nas SQL i używa JDBC do komunikacji z bazami danych.

JPA vs ORM

ORM jest podejściem polegającym na mapowaniu danych w postaci obiektów do relacyjnych baz danych, czyli tabel w **RDBMS**. Można powiedzieć, że **JPA** jest ustandaryzowaną specyfikacją dla **ORM**.

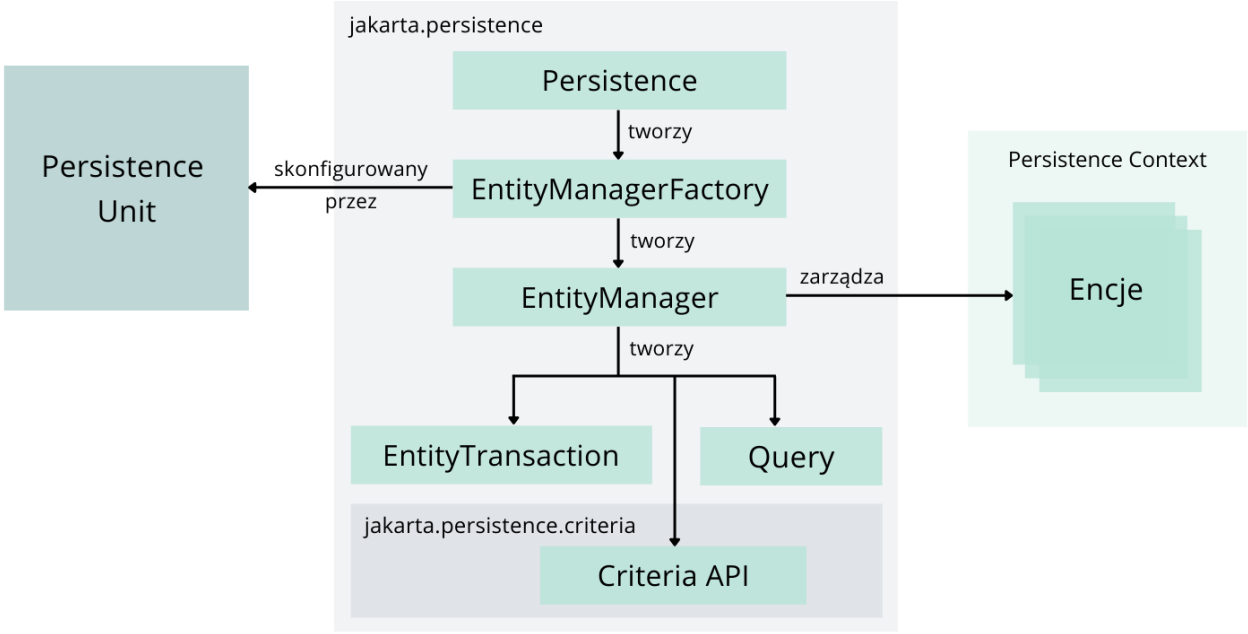
Dostawca JPA

JPA jest specyfikacją, oznacza to, że żeby móc z niego korzystać, musimy mieć dostawcę, który będzie taką specyfikację implementował. W naszym przypadku skupimy się na **Hibernate**. Oznacza to, że w kolejnych przykładach nazwa ta będzie się przeplatała, do momentu, gdzie przejdziemy do konkretnego omówienia **Hibernate**.

Z czego składa się JPA?

Podstawowe składniki JPA zostały wypisane na poniższej grafice. W kolejnych przykładach będziemy przechodzili przez kolejne elementy z przedstawionej grafiki.

Wiem, że wypisane poniżej elementy na tym etapie za wiele Ci nie powiedzą. W pierwszej kolejności omówimy je teoretycznie, a potem będziemy wykorzystywać je w kodzie.



Obraz 2. Uproszczona architektura JPA

Persistence Unit

Definiuje zestaw wszystkich klas encji, które mają być zarządzane przez **EntityManager**. Określa się go w pliku konfiguracyjnym `persistence.xml` (później wrócimy do tego co robi się z tym plikiem). W pliku tym dodajemy również opis połączenia do bazy danych. **Persistence Unit** posiada nazwę, do której należy się odwołać podczas tworzenia obiektu **EntityManagerFactory**.

Przykład pliku `persistence.xml`*:*

```
<persistence
        xmlns="https://jakarta.ee/xml/ns/persistence"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        version="3.0"
        xsi:schemaLocation= "https://jakarta.ee/xml/ns/persistence
            https://jakarta.ee/xml/ns/persistence/persistence_3_0.xsd"> ①
    <persistence-unit name="zajavkaPersistenceUnitExample"> ②
        <class>pl.zajavka.Person</class> ③

        <properties> ④
            <property name="jakarta.persistence.jdbc.driver" value="org.postgresql.Driver"/>
            <property
                name="jakarta.persistence.jdbc.url" value="jdbc:postgresql://localhost:5432/zajavka"/>
            <property name="jakarta.persistence.jdbc.user" value="postgres"/>
            <property name="jakarta.persistence.jdbc.password" value="password"/>

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

① Definicja użytej wersji *JPA*.

② Nazwa *Persistence Unit*.

③ Klasa encji, którą ma zarządzać **EntityManager** utworzony na podstawie *persistence.xml*.

④ Sekcja zawierająca wartości zdefiniowanych własności. Widzimy tutaj różne ustawienia, powinniśmy już kojarzyć parametry połączenia do bazy danych.

Encja (*ang. entity*)

Jest to reprezentacja obiektu świata rzeczywistego, zawierająca informację o tych obiektach, które są przechowywane w bazie danych. Obiektami encji w praktyce są proste obiekty **POJO**, które odzwierciedlają tabele w relacyjnej bazie danych, gdzie pola obiektu odzwierciedlają kolumny z tabeli. Mapowanie encji do tabeli definiuje się za pomocą adnotacji lub plików XML.

Definicja z Wikipedii wygląda tak:

> (...) an object that has an identity, which is independent of the changes of its attributes. It represents long-lived information relevant for the users and is usually stored in a database (...)

jakarta.persistence.Persistence

Klasa, która zawiera statyczne metody pomagające uzyskać **EntityManagerFactory** w sposób niezależny od używanego dostawcy *JPA*.

JPA Providers (*dostawcy JPA*) - tak nazywane są narzędzia implementujące standard *JPA*, takie jak *EclipseLink*, *TopLink* i ten, który będziemy omawiać szczegółowo Hibernate.

jakarta.persistence.EntityManagerFactory

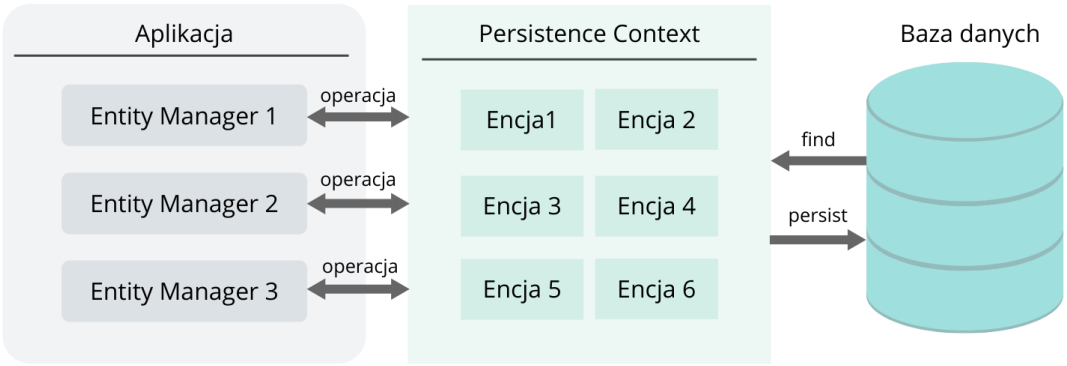
Interfejs, który służy do utworzenia instancji **EntityManager**, gdzie każda z instancji zarządza połączeniem do tej samej bazy danych. **EntityManagerFactory** może być wiele, ale najczęściej definiuje się jeden dedykowany dla każdego źródła danych w aplikacji. Jego odpowiednikiem jest SessionFactory z Hibernate, o czym będzie później.

Przykład tworzenia EntityManagerFactory:

```
EntityManagerFactory entityManagerFactory =
        Persistence.createEntityManagerFactory("zajavkaPersistenceUnitExample");
```

jakarta.persistence.EntityManager

Interfejsu tego używa się do wykonania jakiegoś zadania na bazie danych. W grę wchodzą operacje *CRUD*. Każda instancja **EntityManager** jest powiązana z *Persistence Context*. Jego odpowiednikiem jest Session z Hibernate, o czym będzie później. Używanie **EntityManager** może być rozumiane jak interakcja z *Persistence Context*.



Obraz 3. Persistence Context

Persistence Context jest warstwą abstrakcji pomiędzy aplikacją a bazą danych, za pomocą której **EntityManager** zarządza encjami, dbając o ich synchronizacje i prawidłowy cykl życia.

Przykład tworzenia EntityManager:

```
EntityManager entityManager = entityManagerFactory.createEntityManager();
```

jakarta.persistence.EntityTransaction

Ten interfejs umożliwia grupowanie wielu operacji na danych w jednej transakcji. Innymi słowy, zbiera wszystkie zmiany na powiązanych encjach i wprowadza je na bazie danych jako pojedyncza transakcja, zgodnie z zasadą: wszystko albo nic. Wiemy już całe szczęście czym jest transakcja.

Przykład tworzenia EntityTransaction:

```
EntityTransaction transaction = entityManager.getTransaction();
```

jakarta.persistence.Query

Interfejs używany do wykonywania zapytań na bazie danych. Za dostarczenie **Query** odpowiada **EntityManager**.

Przykład użycia Query:

```
String selectQuery = "SELECT p FROM Person p";
Query query = entityManager.createQuery(selectQuery);
List<Person> people = query.getResultList();
```

Criteria API

Jest to zbiór klas wspierających budowanie zapytań *SQL* za pomocą obiektów Javowych, które pozwalają na dynamiczne tworzenie zapytań.

Przykład użycia Criteria API:

```
CriteriaBuilder criteriaBuilder = entityManager.getCriteriaBuilder();
CriteriaQuery<Person> criteriaQuery = criteriaBuilder.createQuery(Person.class);
Root<Person> root = criteriaQuery.from(Person.class);
criteriaQuery.select(root);
TypedQuery<Person> query = entityManager.createQuery(criteriaQuery);
List<Person> people = query.getResultList();
```

Adnotacje

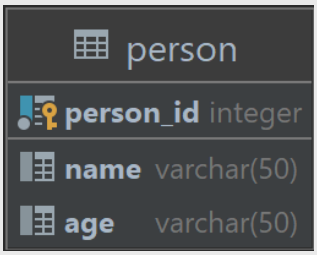
Podczas pracy z **JPA** będziemy korzystali z adnotacji. Adnotacje takie będziemy dodawać w klasie Java, która będzie nam określała encję w bazie danych. Wszystko stanie się jasne jak przejdziemy do przykładów, ale zanim to nastąpi, omówmy bardzo krótko jakie mamy kluczowe adnotacje potrzebne do pracy z JPA (też dostępne w pakiecie *jakarta.persistence*).

- Adnotacje definiujące klasę encji:
  - **@Entity** - Identyfikuje klasę jako encję,
  - **@Table** - Określa do jakiej tabeli odnosi klasa encji,
- Adnotacje określające mapowanie kolumn:
  - **@Column** - Określa, na jaką kolumnę z tabeli mapowane jest pole klasy. Dodatkowo umożliwia dostosowanie jej atrybutów jak np. *updatable*, czy *insertable*, określające czy kolumna jest uwzględniania w generowanej instrukcji *UPDATE* lub *INSERT*,
  - **@Id** - Służy do oznaczenia, które pole klasy jest kluczem głównym w tabeli,
  - **@GeneratedValue** - Stosowana razem z adnotacją *@Id* do oznaczenia strategii generowania wartości klucza głównego,
  - **@Enumerated** - Pozwala wybrać czy mapowanie enum'a działa na podstawie nazwy enum'a (*EnumType.STRING*), czy według jego kolejność (*EnumType.ORDINAL*)
  - **@Lob** - Oznacza, że pole ma być traktowane jako duży obiekt w typie wspieranym przez bazę danych,
- Mapowanie asocjacji (może być jedno lub dwukierunkowe):
  - **@ManyToMany** - dla asocjacji wiele-do-wielu,
  - **@ManyToOne** - dla asocjacji wiele-do-jednego,
  - **@OneToMany** - dla asocjacji jeden-do-wielu,
  - **@OneToOne** - dla asocjacji jeden-do-jednego,
  - **@JoinColumn** - określa kolumnę, na podstawie której tworzona jest relacja między encjami.

Poniżej możesz zobaczyć przykład tabeli `person` i jej odzwierciedlenia w postaci encji *pl.zajavka.Person*. Konfiguracja encji przy wykorzystaniu klas Java i adnotacji będzie wyglądała analogicznie do sytuacji przedstawionej poniżej.

Encja pl.zajavka.Person

```
package pl.zajavka;

import jakarta.persistence.Column;
import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
import jakarta.persistence.Table;

@Entity
@Table(name = "person")
public class Person {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "person_id")
    private Integer personId;

    @Column(name = "name")
    private String name;

    @Column(name = "age")
    private Integer age;

    // getters and setters
}
```



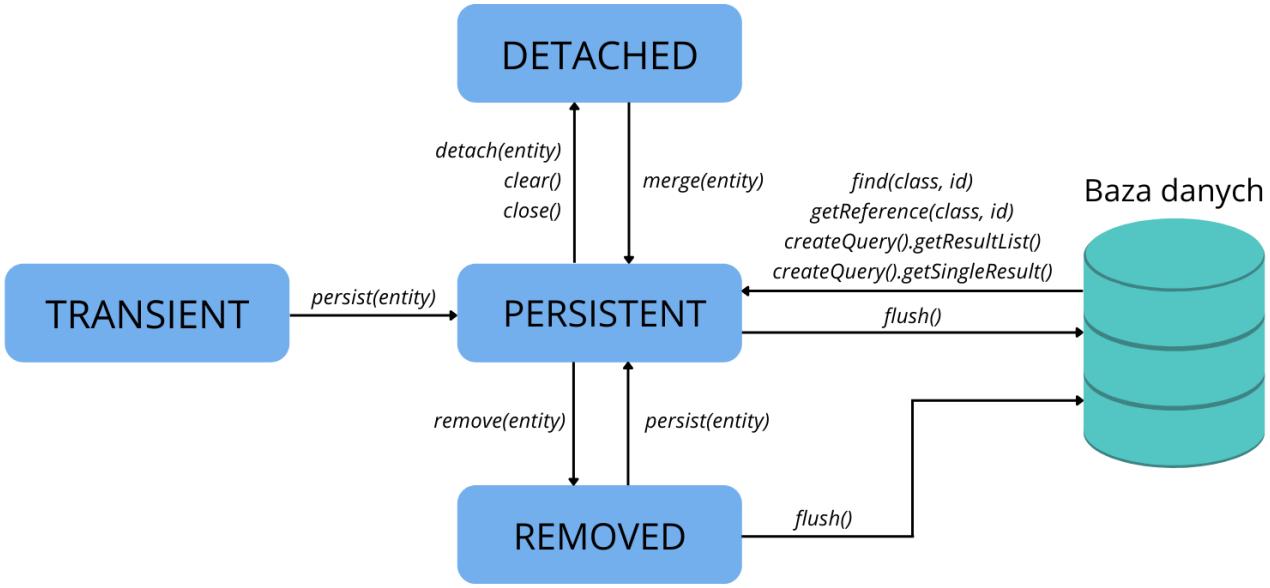
Obraz 4. Tabela Person

Cykl życia encji

W życiu każdego człowieka ważny jest *work-life balance*, czyli równowaga między życiem prywatnym a zawodowym. Encje w rozumieniu JPA też mają swoje życie, ale mniej barwne i w dodatku okrojone do 4 stanów: *Transient*, *Persistent*, *Detached*, *Removed*. Za przejścia pomiędzy stanami odpowiadają poszczególne metody *EntityManager*.

Stany, które zostaną poniżej wypisane związane są z dodaniem kolejnego poziomu abstrakcji nad "czystą" komunikację z bazą danych. Stany te zostały wprowadzone, żeby *Persistence Context* wiedział co się dzieje z encją, na której w danym momencie pracuje.

Można to rozumieć trochę analogicznie do stanów w jakich mogły znajdować się pliki w Git. Mechanizm taki został tam wprowadzony, żeby Git wiedział jak ma odnosić się do każdego z plików. Analogicznie jest tutaj.



Obraz 5. Cykl życia encji z metodami z EntityManager

Stany cyklu życia encji jakie wyróżniamy to:

- **Transient** *(New)* - Stan nowo utworzonej instancji obiektu encji, która nie jest jeszcze powiązana z EntityManager'em i nie jest zapisana w bazie.
- **Persistent** *(Managed)* - Encja jest już we władaniu *persistent context'u*, co oznacza, że *persistent provider* (np. Hibernate) pilnuje zmian w encji. Do bazy danych zmiany wprowadzane są przez metodę *persist()* wywołaną przez EntityManager'a, co ważne, musi być to wykonane w obrębie aktywnej transakcji. Jeżeli encja jest w stanie *persistent*, zmiany w encji są automatycznie synchronizowane na bazie z momentem zatwierdzenia transakcji.
- **Detached** - Stan oznaczający, że encja nie jest już zarządzana przez EntityManager'a i nie ma gwarancji na synchronizację encji z bazą danych.
- **Removed** *(Deleted)* - Użycie metody *remove()* z EntityManager ustawia encję w stan *Removed*, co oznacza, że po zakończeniu transakcji, odpowiedni wiersz w bazie danych zostanie usunięty.

Hibernate i konstruktor bezargumentowy

Encje, które są używane przez Hibernate, potrzebują mieć zdefiniowany kostruktor bezargumentowy. Pamiętasz, że jak go nie napiszesz to i tak jest dostępny, ale jak zaczniesz dodawać swoje własne konstruktory, to już zanika? To tutaj ta wiedza nam się już przyda.

Cytując Stackoverflow:

> Hibernate, and code in general that creates objects via reflection use 'Class<T>.newInstance()' to create a new instance of your classes. This method requires a public no-arg constructor to be able to instantiate the object. For most use cases, providing a no-arg constructor is not a problem.

Wiemy na tej podstawie, że musimy w encjach zapewnić konstruktor bezargumentowy i warto o tym pamiętać. Warto też pamiętać jakimi zasadami rządzi się Java, jeżeli chodzi o domyślne generowanie takich konstruktorów.

To jeszcze jedna refleksja. Czym w Java jest refleksja, co to i po co to? Wspominam o tym tylko na potrzeby wyjaśnienia powyższego mechanizmu. Nie poświęcam oddzielnej części całej ścieżki na wyjaśnianie, czym jest refleksja, gdyż jest to mechanizm, którego w naszej ocenie na tym etapie nauki nie musisz umieć wykorzystywać. Natomiat zachęcamy Cię do własnych poszukiwań.

Wróćmy do tego, o co w tym chodzi. Refleksja to mechanizm, który umożliwia analizę i manipulację strukturą programu w czasie wykonania. Refleksja umożliwia programowi "przeglądanie" swojego własnego kodu, poznawanie informacji o klasach, interfejsach, polach, metodach i konstruktorach, a także wywoływanie tych metod i konstruktorów dynamicznie.

Główne klasy i interfejsy związane z refleksją znajdują się w pakiecie `java.lang.reflect`. Przykłady takich klas to `Class`, `Method`, `Field` i `Constructor`.

Za pomocą refleksji można wykonywać różne czynności, takie jak:

- Pobieranie informacji o klasach: Można uzyskać informacje o dostępnych klasach, interfejsach, polach, metodach i konstruktorach w czasie wykonania.
- Tworzenie nowych obiektów: Można tworzyć nowe instancje klas dynamicznie, nawet jeśli nie znamy ich nazw w czasie kompilacji.
- Wywoływanie metod: Można wywoływać metody na obiektach, nawet jeśli nie znamy ich nazw w czasie kompilacji.
- Manipulacja polami: Można odczytywać i zmieniać wartości pól w czasie wykonania.

Refleksja jest szczególnie przydatna w przypadkach, gdy struktura programu jest dynamiczna i nieznana w czasie kompilacji, lub gdy chcemy zaimplementować mechanizmy takie jak wstrzykiwanie zależności (dependency injection) lub automatyczne mapowanie obiektowo-relacyjne (ORM). Mam nadzieję, że widzisz już, że frameworki takie jak Spring lub Hibernate korzystają z refleksji, gdyż z ich punktu widzenia, nie znają kodu, który Ty napiszesz.

Przykład?

Klasa ReflectionExample

```
import java.lang.reflect.Method;

public class ReflectionExample {
    public static void main(String[] args) throws Exception {
        Class<?> clazz = ExampleClass.class; ①
        Object obj = clazz.getDeclaredConstructor().newInstance(); ②

        Method method = clazz.getMethod("exampleMethod", int.class); ③
        method.invoke(obj, 123); ④
    }
}
```

① Pobieranie klasy `ExampleClass`.

② Tworzenie instancji klasy `ExampleClass`.

③ Pobieranie metody `exampleMethod()` z parametrem `int`.

④ Wywoływanie metody na instancji obiektu.

Klasa ExampleClass

```
class ExampleClass {
    public void exampleMethod(int number) {
        System.out.println("Called exampleMethod() with argument: " + number);
    }
}
```

W tym przykładzie mamy klasę `ExampleClass`, która posiada metodę `exampleMethod()` przyjmującą jeden argument typu `int`. W głównej metodzie `ReflectionExample` używamy refleksji do pobrania klasy `ExampleClass`, utworzenia jej instancji za pomocą `newInstance()`, a następnie pobrania metody `exampleMethod()` za pomocą `getMethod()`. Następnie możemy użyć metody `invoke()` do wywołania metody na instancji obiektu, przekazując odpowiednie argumenty.

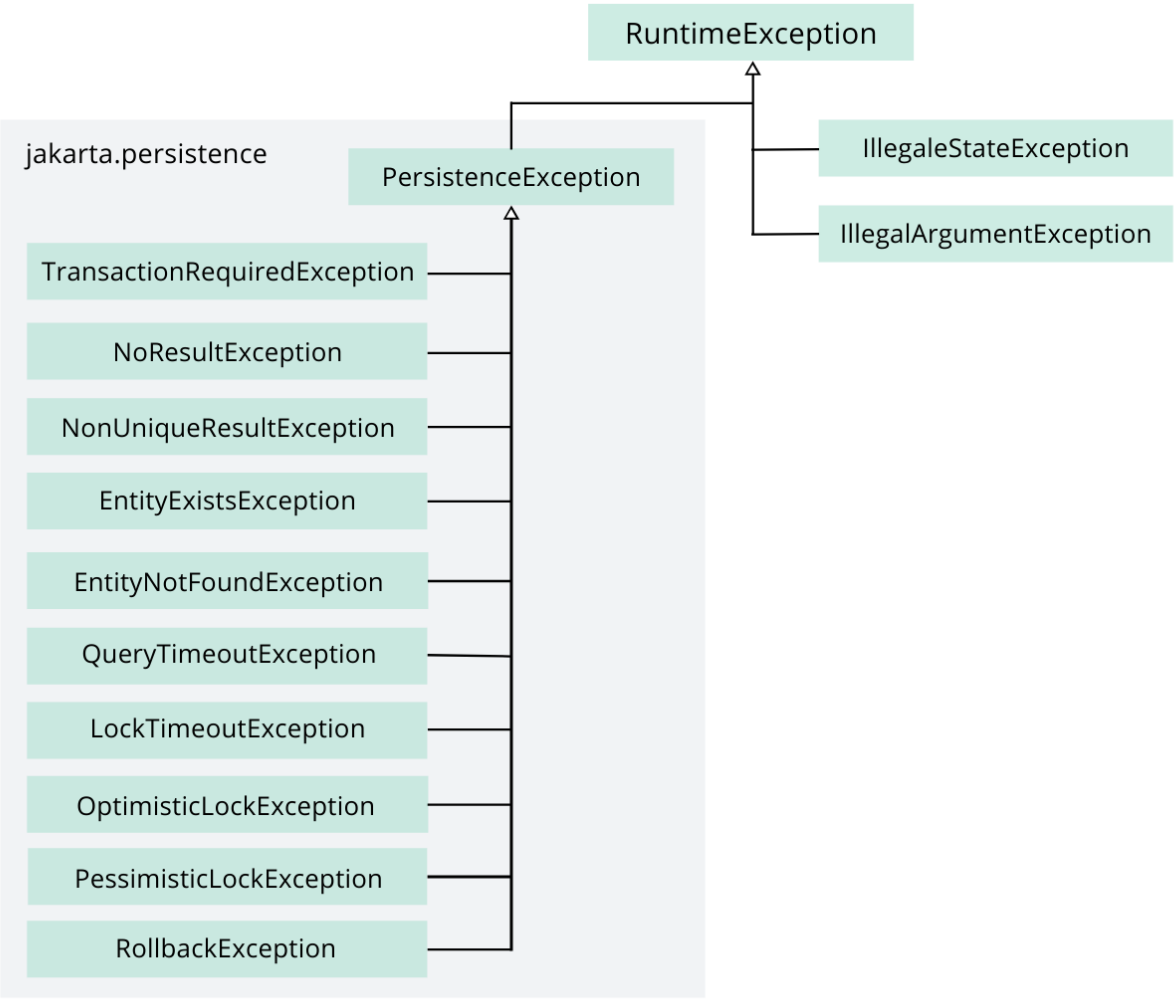
Dzięki refleksji możemy dynamicznie wywołać tę metodę, nawet jeśli nie znamy jej nazwy w czasie kompilacji (nazwa metody jest podana w Stringu). W wyniku działania tego kodu powinno zostać wypisane na konsoli: *"Called exampleMethod() with argument: 123"*.

Korzystanie z refleksji może wiązać się z różnymi dziwnymi wyjątkami, takimi jak np. `IllegalAccessException`, `NoSuchMethodException` czy `InvocationTargetException`, które w praktyce są dosyć uciążliwe do znalezienia przyczyny i rozwiązania.

Refleksja jest potężnym narzędziem, ale może wpływać na wydajność i bezpieczeństwo programu. Dlatego należy stosować ją ostrożnie i z umiarem.

Wyjątki

Wyjątki używane przez **JPA** dziedziczą po *RuntimeException* i są unchecked:



Obraz 6. Wyjątki JPA

- *jakarta.persistence.**PersistenceException***
  Jest to główna klasa wyjątków **JPA**, która reprezentuje podstawowe wyjątki. Z tej klasy dziedziczą poszczególne, bardziej szczegółowe wyjątki. Kilka z nich wyjaśniamy poniżej:
  - *jakarta.persistence.**TransactionRequiredException***
    Pojawia się, kiedy operacja jest wykonywana bez wymaganej aktywnej transakcji.
  - *jakarta.persistence.**RollbackException***
    Rzucany w przypadku błędu podczas zatwierdzania transakcji (`commit()`), czyli podczas zapisu zmian w bazie danych.
  - *jakarta.persistence.**EntityExistsException***
    Rzucany w przypadku wprowadzania do bazy encji (`persist()`), która już istnieje.
  - *jakarta.persistence.**EntityNotFoundException***
    Rzucany w przypadku braku szukanej encji.
  - *jakarta.persistence.**QueryTimeoutException***
    Rzucany w przypadku zbyt długiego wykonywania zapytania.

Przykładowe użycie EntityManager

Zanim przejdziemy do konkretów i do wykorzystania **JPA** razem z **Hibernate**, chciałbym przedstawić mały przykład, obrazujący jak praca z **JPA** mogłaby wyglądać. W kolejnych materiałach przejdziemy już do konkretnej pracy z **Hibernate**. Poniższy test zakłada, że konfiguracja pliku `build.gradle` wygląda następująco:

```
dependencies {
    implementation "ch.qos.logback:logback-classic:$logbackVersion"

    compileOnly "org.projectlombok:lombok:$lombokVersion"
    annotationProcessor "org.projectlombok:lombok:$lombokVersion"

    implementation "org.hibernate.orm:hibernate-core:$hibernateVersion"
    implementation "org.postgresql:postgresql:$postgresqlDriverVersion"

    testCompileOnly "org.projectlombok:lombok:$lombokVersion"
    testAnnotationProcessor "org.projectlombok:lombok:$lombokVersion"

    testImplementation "org.junit.jupiter:junit-jupiter-api:$junitVersion"
    testRuntimeOnly "org.junit.jupiter:junit-jupiter-engine:$junitVersion"
}

test {
    useJUnitPlatform()
    testLogging {
        events "passed", "skipped", "failed"
    }
}

compileJava.options.encoding = 'UTF-8'
```

Analogiczna konfiguracja dla Maven powinna być już na tym etapie prosta i intuicyjna. Pojawiała się już w notatkach kilka razy.

Warto jest tutaj jednak zaznaczyć, że ustawienie `compileJava.options.encoding` dla Maven mogłoby wyglądać tak:

```
<project>
  <!-- ... -->
  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  </properties>
  <!-- ... -->
</project>
```

Warto w tym miejscu zaznaczyć, że na tym etapie, gdy omawiamy **ORM**, **JPA** i **Hibernate**, nie będziemy używali Springa. Do połączenia Springa i Hibernate przejdziemy później.

Materiały są skonstruowane w ten sposób, żeby dać Ci poczucie, że pracujemy z niezależnymi narzędziami, które następnie są w świadomy sposób łączone, a nie z

> potężnym wielkim tworem, który przedstawimy Ci od razu jako całość.

Definicja tabeli `person`*:*

```
CREATE TABLE person
(
    person_id SERIAL NOT NULL,
    age       INT,
    name      VARCHAR(255),
    PRIMARY KEY (person_id)
);
```

Klasa Person

```
package pl.zajavka;

import jakarta.persistence.Column;
import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
import jakarta.persistence.Table;

@Entity
@Table(name = "person")
public class Person {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY) ①
    @Column(name = "person_id")
    private Integer personId;

    @Column(name = "name")
    private String name;

    @Column(name = "age")
    private Integer age;

    // getters and setters
}
```

① Adnotacja ta określa strategię generowania wartości dla kluczy głównych. W tabeli `person` zastosowaliśmy dla klucza głównego typ `SERIAL`, czyli klucze główne są generowane automatycznie. Przy wykorzystaniu tej konfiguracji możemy powiedzieć JPA, w jaki sposób są generowane klucze główne. Istnieje także możliwość, żeby to JPA automatycznie generowało te wartości. Nie będziemy głębiej schodzić w temat, na nasze potrzeby wystarczy tutaj użyć `GenerationType.IDENTITY`.

Do tego, w katalogu `src/main/resources/META-INF` mamy umieszczony plik `persistence.xml`, który ma następującą zawartość:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<persistence xmlns="https://jakarta.ee/xml/ns/persistence"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             version="3.0"
             xsi:schemaLocation="https://jakarta.ee/xml/ns/persistence
https://jakarta.ee/xml/ns/persistence/persistence_3_0.xsd">
```

```
    <persistence-unit name="zajavkaPersistenceUnitExample">
        <class>pl.zajavka.hibernateintro.Person</class>

        <properties>
            <property name="jakarta.persistence.jdbc.driver" value="org.postgresql.Driver"/>
            <property
                    name="jakarta.persistence.jdbc.url" value=
"jdbc:postgresql://localhost:5432/zajavka"/>
            <property name="jakarta.persistence.jdbc.user" value="postgres"/>
            <property name="jakarta.persistence.jdbc.password" value="password"/>

            <property name="hibernate.dialect" value="org.hibernate.dialect.PostgreSQLDialect"/>
            <property name="hibernate.hbm2ddl.auto" value="none"/>

            <property name="hibernate.show_sql" value="true"/>
            <property name="hibernate.format_sql" value="true"/>
        </properties>
    </persistence-unit>
</persistence>
```

Znaczenie parametrów, przy których jest napisane **hibernate** zostanie wyjaśnione później.

Mając już dodane wspomniane pliki, możemy przejść do przykładowego testu jednostkowego przedstawiającego wykorzystanie **JPA**:

```
package pl.zajavka;

import java.util.List;

import jakarta.persistence.EntityManager;
import jakarta.persistence.EntityManagerFactory;
import jakarta.persistence.Persistence;
import jakarta.persistence.Query;

import org.junit.jupiter.api.Assertions;
import org.junit.jupiter.api.Test;

class JpaExampleTest {

    @Test
    public void createAndUpdatePersonTest() {

        EntityManagerFactory entityManagerFactory =
                Persistence.createEntityManagerFactory("zajavkaPersistenceUnitExample"); ①

        try {
            EntityManager entityManager = entityManagerFactory.createEntityManager(); ②

            String personName = "Karol";
            int personAge = 69; ③

            createPerson(entityManager, personName, personAge); ④

            List<Person> allPeople = getAllPeople(entityManager); ⑤

            int newPersonAge = 30;
            Person karol = allPeople.get(0);
            updatePersonAge(entityManager, karol, newPersonAge); ⑥
```

```
            Assertions.assertEquals(1, allPeople.size());
            Assertions.assertEquals(30, allPeople.get(0).getAge()); ⑦

            entityManager.close();
        } finally {
            entityManagerFactory.close(); ⑧
        }
    }

    // pozostałe metody
}
```

① Przygotowanie obiektu *EntityManagerFactory*.

② Przygotowanie obiektu *EntityManager*.

③ Definicja danych do utworzenia nowego obiektu *Person*.

④ Metoda tworząca obiekt *Person* i zapisująca go do bazy danych.

⑤ Metoda odczytująca wszystkie wiersze z tabeli *Person*.

⑥ Metoda aktualizująca pierwszy (i jedyny) obiekt odczytany z bazy danych.

⑦ Weryfikacja operacji wykonanych na bazie danych.

⑧ Zamknięcie *EntityManager* i *EntityManagerFactory*.

Metoda zapisująca do bazy obiekt Person:

```
private void createPerson(EntityManager entityManager, String newPersonName, int newPersonAge) {
    entityManager.getTransaction().begin(); ①

    Person person = new Person();
    person.setName(newPersonName);
    person.setAge(newPersonAge); ②

    entityManager.persist(person); ③

    entityManager.getTransaction().commit(); ④
}
```

① Otwarcie transakcji.

② Przygotowanie do zapisu nowej encji *Person*.

③ Dodanie encji do persistent context'u za pomocą metody *jakarta.persistence.EntityManager.* ***persist(****Object entity****)***.

④ Zatwierdzenie transakcji i zapis encji do bazy.

Metoda odczytująca z bazy wszystkie wiersze z tabeli Person:

```
private List<Person> getAllPeople(EntityManager entityManager) {
    entityManager.getTransaction().begin(); ①

    String selectQuery = "SELECT p FROM Person p";
    Query query = entityManager.createQuery(selectQuery); ②
    List<Person> people = query.getResultList(); ③

    entityManager.getTransaction().commit(); ④
```

```
    return people;
}
```

① Otwarcie transakcji.

② Przygotowanie zapytania z użyciem *JPA Query String*.

③ Wykonanie zapytania za pomocą użycia metody *jakarta.persistence.Query.**getResultList***.

④ Zatwierdzenie transakcji.

Metoda aktualizująca wiersz w tabeli Person:

```
private void updatePersonAge(EntityManager entityManager, Person personToUpdate, int newPersonAge) {
    entityManager.getTransaction().begin(); ①

    personToUpdate.setAge(newPersonAge); ②

    entityManager.getTransaction().commit(); ③
}
```

① Otwarcie transakcji.

② Zmiana wartości pola Age w encji Person.

③ Zatwierdzenie transakcji i zapis encji do bazy.

Podsumowanie

Wiemy już na tym etapie "jak mniej więcej będzie się pracowało z JPA" i "na czym mniej więcej to polega". Omówiliśmy standard **JPA**, możemy teraz przejść do przykładów konkretnej pracy z **Hibernate**.