Hibernate

Spis treści

Hibernate - Intro

Obraz 1. Logo Hibernate

Hibernate - Powstał w 2001 i jest jedną z najpopularniejszych implementacji *JPA*. Projekt ten został rozpoczęty przez Gavina Kinga. Jest tak duży i rozbudowany, że dostał miano framework'u. Swoimi możliwościami wychodzi poza standard *JPA*. Powodem jego powstania było stworzenie alternatywy do używanych wtedy EJB2-style entity beans (nie będziemy o nich rozmawiać). Jest to narzędzie **ORM**, które jest open source. Oficjalną stronę znajdziesz tutaj.

Definicja z Wikipedii wygląda tak:

> (...) an object–relational mapping tool for the Java programming language (...)

Hibernate to narzędzie **ORM**, które będzie przez nas wykorzystywane do zapewnienia mapowania encji

w postaci klas Javy na relacyjną bazę danych i odwrotnie. Oprócz tego będziemy wykorzystywali to narzędzie do wykonywania zapytań na bazie danych i pobierania danych w sposób bardziej zautomatyzowany niż ręczne pisanie zapytań **SQL**. Konfiguracja **Hibernate** może być zdefiniowana w plikach **XML** lub przy wykorzystaniu adnotacji Java.

Co nam daje stosowanie Hibernate w projekcie?

Zalety stosowania Hibernate:

- Zwiększa produktywność, dając na tacy coś, co trzeba by było przygotowywać od zera, przez co pozwala się skupić na aspektach biznesowych, a nie technicznych. Podobnie jak omawiany wcześniej *Spring Framework*.
- Ułatwia utrzymanie aplikacji, zmniejszając ilość kodu, który musimy napisać, co poprawia jego czytelność i upraszcza wprowadzanie ewentualnych zmian, czy refaktoryzacji.
- Poprawia wydajność, dzięki wbudowanym mechanizmom optymalizacji, jak przykładowo mechanizm *cache'owania*, który omówimy później.
- Pozwala na pewną elastyczność i niezależność od stosowanego **RDBMS** (*Relational Database Management System*), ponieważ umożliwia jego zmianę na inny wspierany system.

Hibernate vs JDBC

Gdybyśmy chcieli zestawić te dwa terminy ze sobą, to moglibyśmy powiedzieć, że **Hibernate** jest odpowiedzialny za mapowanie **ORM**, podczas gdy **JDBC** jest po prostu interfejsem, który zapewnia łączność z bazą danych.

	Hibernate	JDBC
Cel	ORM *(Object-Relational Mapping)*	Połączenie do bazy danych
Język zapytań	HQL *(Hibernate Query Language)* lub JPQL *(Jakarta Persistence Query Language)*	SQL *(Structured Query Language)*
Skąd to wziąć	Zależność w postaci pliku `*.jar`	Zawarte w Java SE
Cacheowanie	Wspiera poprzez 2 poziomy cache	Nie wspiera
Kto zarządza	JBoss by Red Hat	Oracle

O cacheowaniu wspominaliśmy przy okazji omawiania wzorców projektowych. Wrócimy do tematu w późniejszych materiałach dotyczących **Hibernate**.

Hibernate vs JPA

JPA jest specyfikacją dla dostawców warstwy persystencji do implementacji (*specification for persistence providers to implement* - niektóre rzeczy lepiej brzmią po angielsku ☺). Hibernate jest jedną z takich implementacji.

Różnica między JPA i Hibernate polega na tym, że możemy oznaczać klasy w kodzie przy wykorzystaniu adnotacji JPA ile chcemy, ale bez implementacji tej specyfikacji, nic się nie wydarzy. Dlatego JPA możemy rozumieć jak interfejsy, podczas gdy implementacja JPA, czyli Hibernate to kod, który służy do

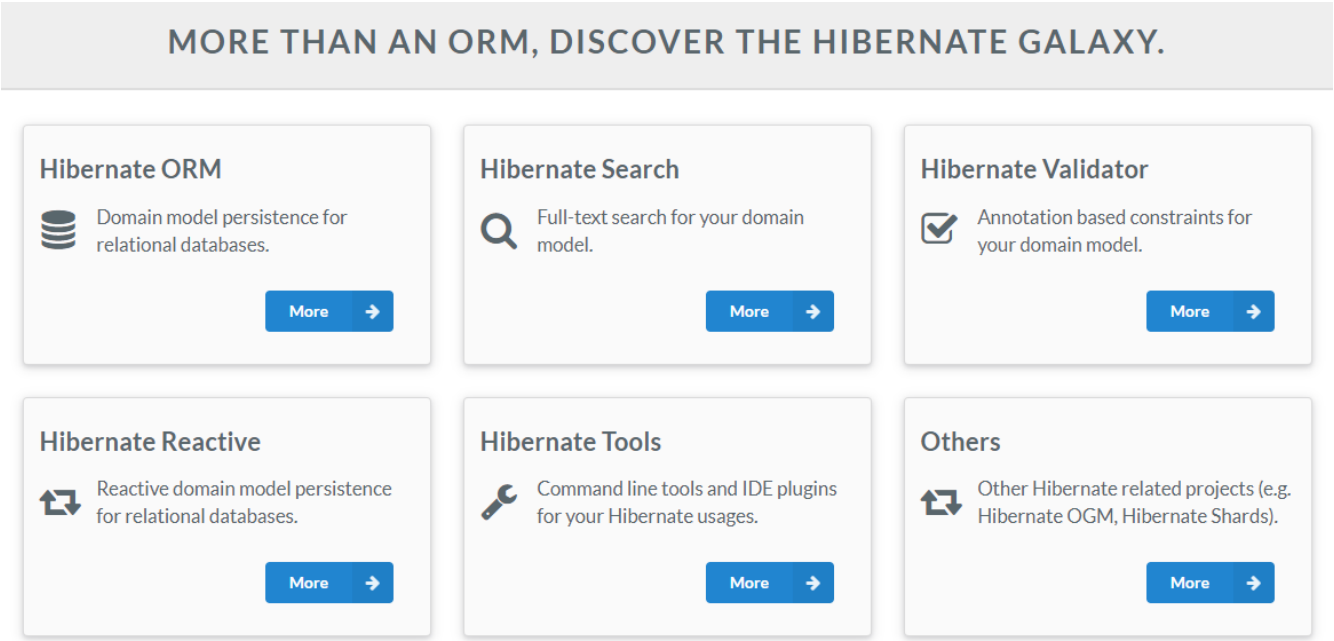
implementowania tych interfejsów. Hibernate implementuje pod maską funkcjonalności oferowane przez JPA.

Gdy stosujemy Hibernate, to w rzeczywistości używamy JPA z implementacją w postaci Hibernate. Jeżeli w kodzie będziemy opierali się o kod JPA, czyli o specyfikację, to teoretycznie w każdej chwili powinniśmy być w stanie wymienić implementację tej specyfikacji. Analogicznie to samo możemy zrobić z implementacją interfejsu. Jeżeli kilka klas implementuje ten sam interfejs, to teoretycznie bez problemu powinniśmy być w stanie wymienić klasę implementującą dany interfejs.

Z drugiej strony możemy u nas w kodzie stosować "czysty" Hibernate, wtedy wymiana dostawcy JPA nie będzie już taka prosta, bo w tym przypadku opieramy się bezpośrednio o Hibernate, a nie o specyfikację. Stanie się to jaśniejsze, gdy przejdziemy do praktyki.

Z jakich projektów składa się Hibernate?

Spójrzmy na oficjalną stronę hibernate.org.



Obraz 2. Projekty Hibernate

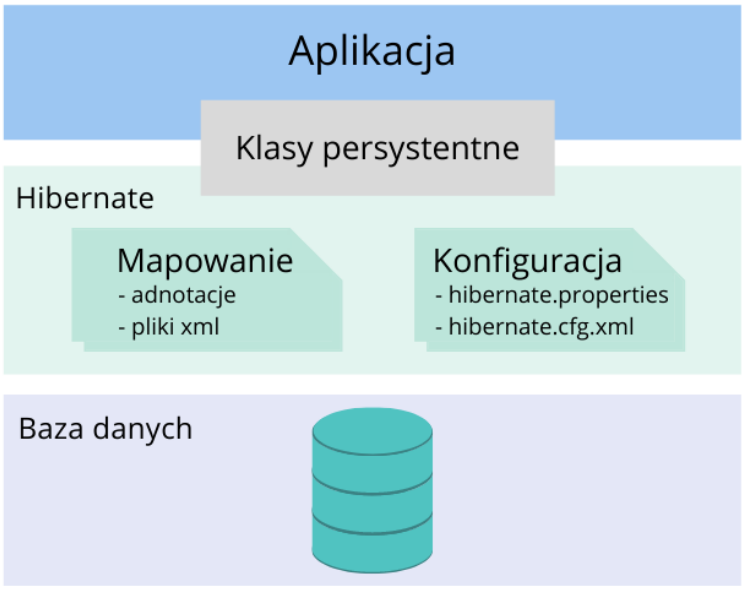
Przykładowe projekty składające się na Hibernate:

- Hibernate **ORM** - Wcześniej znane jako *Hibernate Core*. Fundament i najstarsza część *Hibernate* odpowiadająca za mapowanie obiektowo-relacyjne. Działająca niezależnie od reszty komponentów Hibernate.
- Hibernate **Envers** - Moduł, który służący do audytów i wersjonowania klas encji. Można go porównać do systemów kontroli wersji jak *Git*, czy *Subversion*, ale dla klas encji.
- Hibernate **Search** - Dostarcza pomoc do wyszukiwań na postawie zadanego tekstu.
- Hibernate **Validator** - Niezależny projekt implementujący specyfikację *Bean Validation*. Opisując to w ludzki sposób, projekt wprowadza adnotacje, które pozwalają na definiowanie ograniczeń oraz sprawdzanie czy obiekty się tych ograniczeń trzymają.

Zależność do Hibernate ORM znajduje się pod nazwą `hibernate-core`*:*

```
// https://mvnrepository.com/artifact/org.hibernate.orm/hibernate-core
implementation "org.hibernate.orm:hibernate-core:$hibernateVersion"
```

Jak bardzo ogólnie będzie wyglądała praca z Hibernate? Będziemy definiować mapowania w klasach encji (klasy persystentne - czyli takie, których będziemy używali do pracy z bazą danych), żeby framework "wiedział", jak chcemy połączyć ze sobą obiekty w Javie z tabelami w bazie danych. Możemy to nazwać konfiguracją mapowania. Do tego będziemy stosować pliki określające konfigurację, takie jak np. *hibernate.properties* lub *hibernate.cfg.xml*.



Obraz 3. Widok wysokopoziomowej architektury Hibernate

Konfiguracja

Do konfiguracji Hibernate można użyć:

- pliku `hibernate.properties` - stare podeście
- pliku `hibernate.cfg.xml`.

Pliki te należy umieścić w folderze *src/main/resources* albo *src/test/resources*, w zależności od kontekstu użycia.

Plik konfiguracyjny zawiera informacje dotyczące bazy danych, takie jak namiary na bazę lub parametry konfiguracyjne. Przykładowe właściwości konfiguracji:

- *hibernate.connection.driver_class* - Klasa sterownika JDBC,
- *hibernate.connection.url* - Adres URL połączenia JDBC,
- *hibernate.connection.username* - Nazwa użytkownika bazy danych,
- *hibernate.connection.password* - Hasło do bazy danych,
- *hibernate.dialect* - Dialekt stosowany przez używaną bazę danych. Dialekt dostarcza informacji na temat sposobu konwersji zapytań Hibernate (HQL) do natywnych zapytań SQL,

- *hibernate.hbm2ddl.auto* - Jeżeli ustawimy ten parametr to Hibernate będzie starał się automatycznie tworzyć encje w bazie danych na podstawie naszej konfiguracji encji. Na tym etapie dla uproszczenia powiedzmy, że najlepiej jest zostawić tę wartość ustawioną na **none**,
- *hibernate.show_sql* - Czy wykonane zapytania SQL mają byc drukowane w konsoli. Gdy ustawimy ten parametr, to Hibernate będzie drukował zapytania na ekranie, ale parametry wykorzystane w tych zapytaniach będą zastąpione znaczkiem ?. Wrócimy do tego później,
- *hibernate.format_sql* - Czy zapytania SQL mają być sformatowane przed wydrukowaniem w konsoli,
- *hibernate.use_sql_comments* - Hibernate będzie generował komentarze, dzięki którym będzie starał się nam przekazać co właśnie robi.

Stwierdzenie natywny oznacza, że coś lub ktoś jest takie jak w warunkach naturalnych, właściwe danemu środowisku informatycznemu. Czyli w uproszczeniu, natywne zapytanie SQL to będzie takie, które jest naturalne dla danej bazy danych. Słowo to będzie się jeszcze powtarzało i będzie je łatwiej zrozumieć w konkretnym kontekście.

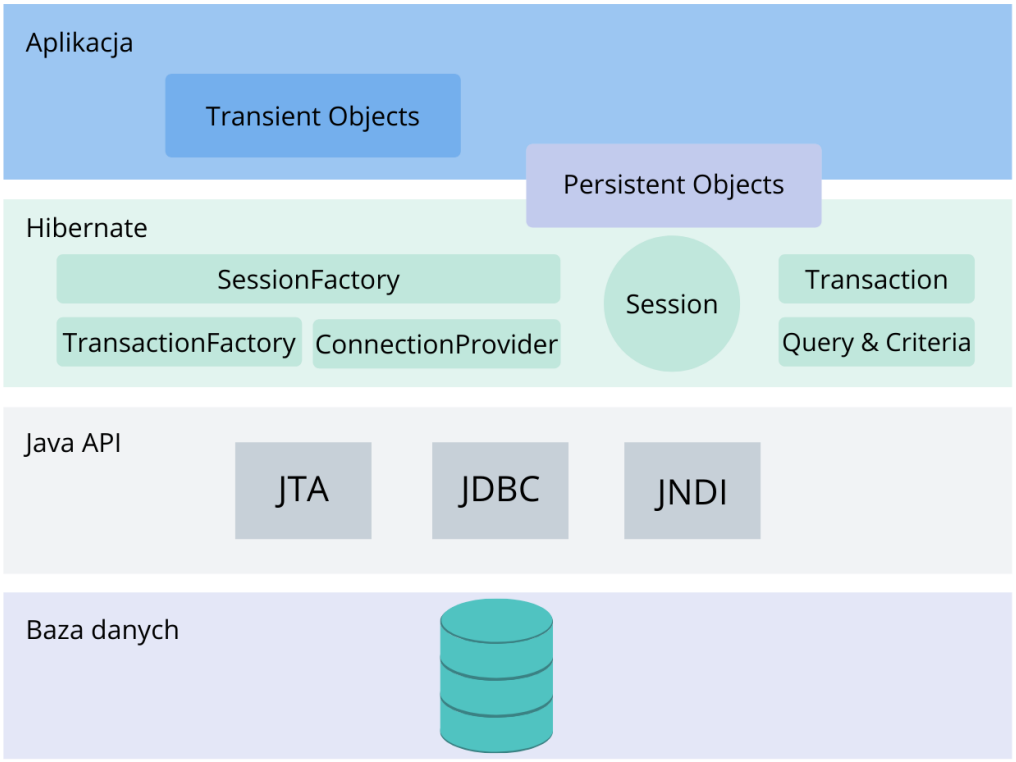
Przykładowa konfiguracja w pliku `hibernate.cfg.xml`*:*

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration PUBLIC
        "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
        "http://hibernate.org/dtd/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
    <session-factory>
        <property name="hibernate.connection.driver_class">org.postgresql.Driver</property>
        <property name="hibernate.connection.url">jdbc:postgresql://localhost:5432/zajavka</property>
        <property name="hibernate.connection.username">postgres</property>
        <property name="hibernate.connection.password">postgres</property>

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

        <property name="hibernate.show_sql">true</property>
        <property name="hibernate.format_sql">true</property>
        <property name="hibernate.use_sql_comments">true</property>

        <mapping class="pl.zajavka.Person"/>
    </session-factory>
</hibernate-configuration>
```

Konfiguracja Hibernate jest również możliwa przy wykorzystaniu klasy *org.hibernate.cfg.**Configuration***. Przejdziemy do tego zaraz.

Z czego składa się Hibernate?

Gdybyśmy mieli rozrysować ogólny schemat klas i interfejsów, z których będziemy korzystać przy wykorzystaniu **Hibernate**, mógłby on wyglądać w ten sposób:



Obraz 4. Widok niskopoziomowej architektury Hibernate

Zwróć uwagę, że mamy tutaj dostępne inne klasy, niż widzieliśmy na etapie omawiania JPA. Wyjaśnijmy bardzo ogólnie znaczenie poszczególnych komponentów:

- **Transient Objects** - obiekty używane w aplikacji, które nie są zapisywane w bazie danych,
- **Persistent Objects** - obiekty używane w aplikacji, które są zapisywane w bazie danych,
- *org.hibernate.**SessionFactory*** - interfejs *org.hibernate.**SessionFactory*** jest odpowiednikiem interfejsu *jakarta.persistence.**EntityManagerFactory*** pochodzącego z JPA. Odpowiada za przygotowanie obiektu *org.hibernate.**Session***.
- *org.hibernate.**Session*** - interfejs *org.hibernate.**Session*** jest najważniejszym elementem frameworku *Hibernate*. Jest odpowiednikiem interfejsu *jakarta.persistence.**EntityManager*** pochodzącego z JPA. To właśnie obiekt ***Session*** odpowiada za fizyczne połączenie do bazy danych w celu wykonania operacji *CRUD* na instancjach zmapowanych klas encji.
- **Transaction** - interfejs, który reprezentuje transakcję, która jest pod kontrolą Hibernate,
- **TransactionFactory** - interfejs do generowania transakcji Hibernate,
- **ConnectionProvider** - interfejs do pozyskiwania połączeń JDBC,
- **Query & Criteria** - interfejsy używane do pozyskiwania danych z bazy przy wykorzystaniu Hibernate,
- **JTA** - zestaw klas i interfejsów określany jako *Jakarta Transaction API*,
- **JDBC** - zestaw klas i interfejsów określany jako *Java Database Connectivity*,
- **JNDI** - zestaw klas i interfejsów określany jako *Java Naming and Directory Interface*.

Przechodząc przez kolejne zagadnienia, będziemy odnosić się do klas i interfejsów wymienionych na powyższym schemacie.

SessionFactory i Session

Zacznijmy od tego, jak można napisać podstawowy program wykorzystujący interfejsy `SessionFactory` oraz `Session`:

Klasa Person

```
package pl.zajavka;

import jakarta.persistence.*;

@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
}
```

```
package pl.zajavka;

import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;

public class PersonExample {

    public static void main(String[] args) {
        try (
            SessionFactory sessionFactory = createSessionFactory(); ①
            Session session = sessionFactory.openSession() ②
        ) {
            session.beginTransaction(); ③
            Person newPerson = new Person(); ④
            newPerson.setName("Robert");
            newPerson.setAge(38);
            session.persist(newPerson); ⑤
            session.getTransaction().commit(); ⑥

            Person person = session.find(Person.class, 1); ⑦
            System.out.println(person); ⑧
        }
    }

    private static SessionFactory createSessionFactory() {
        Configuration configuration = new Configuration();
        configuration.configure("/META-INF/hibernate.cfg.xml");
        return configuration.buildSessionFactory();
```

```
    }
}
```

① Tworzymy `SessionFactory`. Stosujemy `try-with-resources`, więc otwarte połączenia do bazy danych zostaną zamknięte po zakończeniu tego bloku. W całej aplikacji powinno być otwarte tylko jedno `SessionFactory`. Inaczej mówiąc `SessionFactory` powinno działać jak Singleton,

② Tworzymy `Session`. `Session` jest `Closeable`, zatem po zakończeniu bloku `try-with-resources`, zostanie automatycznie wywołana metoda `close()`. Dokumentacja metody `close()` mówi, że: *End the session by releasing the JDBC connection and cleaning up*. Czyli wykonanie tej metody zamyka otwarte połączenia do bazy danych i zwalnia zasoby,

③ Rozpoczynamy transakcję,

④ Tworzymy encję `Person`,

⑤ Zapisujemy `Person` do bazy danych,

⑥ Commitujemy transakcję,

⑦ Wyszukujemy obiekt `Person` w bazie danych po kluczu głównym równym 1,

⑧ Wyświetlamy osobę na ekranie.

Czyli czym są te `SessionFactory` oraz `Session`. Przeczytajmy o tym w dokumentacji. Przypominam, że jeżeli chcesz dowiedzieć się więcej na temat każdej z klas czy interfejsów, możesz to zrobić w dokumentacji, czyli np. klikając w Intellij `ctrl + q`. Przykładowo:

SessionFactory:

> A SessionFactory represents an "instance" of Hibernate: it maintains the runtime metamodel representing persistent entities, their attributes, their associations, and their mappings to relational database tables, along with configuration that affects the runtime behavior of Hibernate, and instances of services that Hibernate needs to perform its duties.
>
> Crucially, this is where a program comes to obtain sessions. Typically, a program has a single SessionFactory instance, and must obtain a new Session instance from the factory each time it services a client request. Depending on how Hibernate is configured, the SessionFactory itself might be responsible for the lifecycle of pooled JDBC connections and transactions, or it may simply act as a client for a connection pool or transaction manager provided by a container environment.
>
> (…)
>
> Every SessionFactory is a JPA EntityManagerFactory.
>
> (…)

Session:

> The main runtime interface between a Java application and Hibernate. Represents the notion of a persistence context, a set of managed entity instances associated with a logical

> transaction.
>
> The lifecycle of a Session is bounded by the beginning and end of the logical transaction. (But a long logical transaction might span several database transactions.)
>
> The primary purpose of the Session is to offer create, read, and delete operations for instances of mapped entity classes. An instance may be in one of three states with respect to a given session:
>
> - transient: never persistent, not associated with any Session,
> - persistent: associated with a unique Session,
> - detached: previously persistent, not associated with any Session
>
> Any instance returned by get(Class, Object) or by a query is persistent.
>
> A transient instance may be made persistent by calling persist(Object). A persistent instance may be made detached by calling detach(Object). A persistent instance may be marked for removal, and eventually made transient, by calling remove(Object). Persistent instances are held in a managed state by the persistence context. Any change to the state of a persistent instance is automatically detected and eventually flushed to the database.
>
> (...)
>
> Every Session is a JPA EntityManager.
>
> (...)

Chciałbym teraz nawiązać do stwierdzeń *SessionFactory vs EntityManagerFactory* oraz *Session vs EntityManager*. Których powinniśmy używać? Tak naprawdę, to możemy używać klas z paczki `org.hibernate` rozumiejąc to jako jeden sposób pracy z Hibernate, lub możemy stosować klasy z paczki `jakarta.persistence` i również będzie to prawidłowe podejście. Decydując się na któreś z nich, trzeba natomiast pamiętać o konsekwencjach.

Używanie klas z paczki `jakarta.persistence` jest jak stosowanie interfejsów w Javie zamiast konkretnych klas. Jeżeli stosujemy interfejs, to możemy łatwo wymieniać jego implementację. Stosowanie klas z paczki `org.hibernate` jest jak używanie klas bezpośrednio, bez interfejsów. W przypadku gdy wystąpi potrzeba dokonania zmiany takiej klasy, sprawy zaczynają się komplikować. Pamiętając, że Hibernate jest implementacją JPA, to można to rozumieć analogicznie do klasy implementującej jakiś interfejs. Jeżeli będziemy stosowali Hibernate bezpośrednio, czyli klasy z paczki `org.hibernate`, możemy mieć problem jeżeli będziemy chcieli zmienić dostawcę JPA.

Ale szczerze?

- Ja nigdy nie spotkałem się z takim przypadkiem w praktyce - to jest raz,
- Później jak połączymy to wszystko ze Springiem, to podejście do wykorzystania tych klas będzie jeszcze inne - to jest dwa.

Jeżeli natomiast chcielibyśmy używać klas i interfejsów JPA, natomiast Hibernate nadal miałby być

naszą implementacją, to pisalibyśmy kod wykorzystując klasy, interfejsy i adnotacje z paczki `jakarta.persistence`.

Mapowanie

Spora część czasu, podczas którego będziemy faktycznie korzystać z Hibernate, będzie poświęcana na tworzenie mapowań pomiędzy tabelami w bazie danych a obiektami w kodzie. Do mapowania encji na tabele i odwrotnie w Hibernate można użyć:

- `adnotacji` *(opisanych wcześniej w sekcji JPA)*
- plików `*.hbm.xml`

Przykładowa konfiguracja w pliku `person.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 = "Person" table = "PERSON">
      <meta attribute = "class-description"> This class contains the person detail. </meta>
      <id name = "personId" type = "int" column = "person_id">
         <generator class="native"/>
      </id>
      <property name = "name" column = "name" type = "string"/>
      <property name = "age" column = "age" type = "int"/>
   </class>
</hibernate-mapping>
```

W tej części notatek nie zgłębiamy się dalej w temat, gdyż mapowania będą nam towarzyszyły przez cały czas, gdy będziemy korzystać z Hibernate. Przykład z plikiem `*.hbm.xml` został pokazany jednorazowo. W kolejnych przykładach będziemy cały czas stosować adnotacje z paczki `jakarta.persistence`.

flush()

Hibernate pozwala nam na konfigurację, która będzie odkładała moment wykonania zapytania na bazie danych i zmiany w encjach będą przetrzymywane w pamięci operacyjnej.

Flushowanie to proces synchronizowania stanu `Session`, jaki znajduje się w pamięci operacyjnej z bazą danych. Hibernate będzie robił taki `flush()` automatycznie, natomiast mamy też możliwość wymuszenia takiej synchronizacji ręcznie, poprzez metodę `Session#flush()`.

Co to oznacza? Skoro zmiany są przetrzymywane w pamięci, to możemy doprowadzić do takiej sytuacji, że aplikacja będzie widziała zmiany wprowadzone w jakiś encjach, ale nie zostaną wykonane jeszcze żadne zapytania SQL na bazie danych. Żeby wtedy to wymusić ręcznie, musielibyśmy napisać `flush()`.

Samo wykonanie `flush()`, powoduje wykonanie SQL, ale nie oznacza to wykonania *commit*. Inaczej można zrozumieć to tak, że to Hibernate decyduje, kiedy wykona zapytania na bazie danych. Mechanizm ten może służyć do optymalizacji, w takim kontekście, że możemy ograniczyć ilość zapytań

do bazy danych i pracować w pamięci operacyjnej, ale to już jest wyższa szkoła jazdy i nie będziemy się w to zagłębiać.

Strategie zarządzania bazą danych

Omówimy sobie teraz pokrótce parametr `hibernate.hbm2ddl.auto`, jakie są konsekwencje jego stosowania i do czego on służy.

Podczas pracy z JPA i Hibernate mamy dwie możliwości zarządzania schematem bazy danych (czyli aktualizowania tego jak wyglądają np. kolumny w tabelach):

- Istnieją narzędzia, które pozwalają na zapisywanie kolejnych DDL i zapisywanie, które DDL zostały już na takiej bazie danych wykonane, a które mają jeszcze zostać uruchomione. Na razie nie będziemy wchodzić w to głębiej, ale przykładowe takie narzędzie to np. Flyway - omówimy go później. Można to rozumieć jak taki Git dla bazy danych.
- Przy wykorzystaniu ustawienia `hibernate.hbm2ddl.auto` można wygenerować lub aktualizować schemat bazy danych w oparciu o mapowania encji, które mamy zdefiniowane.

Osobiście o wiele częściej stosowałem to pierwsze podejście i będzie ono przez nas preferowane, ale warto zaznaczyć, co wnosi nam korzystanie z parametru `hibernate.hbm2ddl.auto`, i czemu ustawiliśmy tę wartość na `none`.

Ustawienie wartości `hibernate.hbm2ddl.auto` powoduje, że Hibernate będzie generował i wykonywał DDL za nas automatycznie. Parametr ten może przyjąć następujące wartości:

- **none** – ustawiając ten parametr mówimy Hibernate'owi, że ma nic nie robić i zostawić generowanie DDL w spokoju,
- **create-only** – ta wartość mówi, że Hibernate na utworzyć schemat bazy danych na podstawie modelu encji,
- **drop** – ta wartość mówi, że Hibernate ma usunąć schemat bazy danych na podstawie modelu encji,
- **create** – ta opcja może być rozumiana jak usuń i stwórz na nowo na podstawie modelu encji,
- **create-drop** – ponownie, ta opcja może być rozumiana jak usuń i stwórz na nowo na podstawie modelu encji, różnica jest taka, że gdy zostanie zamknięty `EntityManagerFactory` lub `SessionFactory`, schemat bazy danych zostanie usunięty ponownie,
- **validate** – ta opcja mówi, że Hibernate ma zwalidować (sprawdzić), czy schemat bazy danych jest zgodny z modelem mapowań encji.
- **update** – ta opcja nakazuje aktualizację schematu bazy danych poprzez porównanie istniejącego schematu z mapowaniami encji i wygenerowanie odpowiednich skryptów migracji schematu

Model mapowań encji oznacza, naszą konfigurację mapowań w aplikacji, np. przy wykorzystaniu adnotacji `@Entity` itp. Czy te klasy, które służą nam do określenia jak mają być mapowane na tabele w bazie danych.

Pojawia się zatem pytanie, kiedy byłoby nam to potrzebne i jakie podejście będziemy stosować w przykładach?

Chciałbym, żebyśmy na potrzeby przykładów stosowali ustawienie **none**. Dzięki temu będziemy

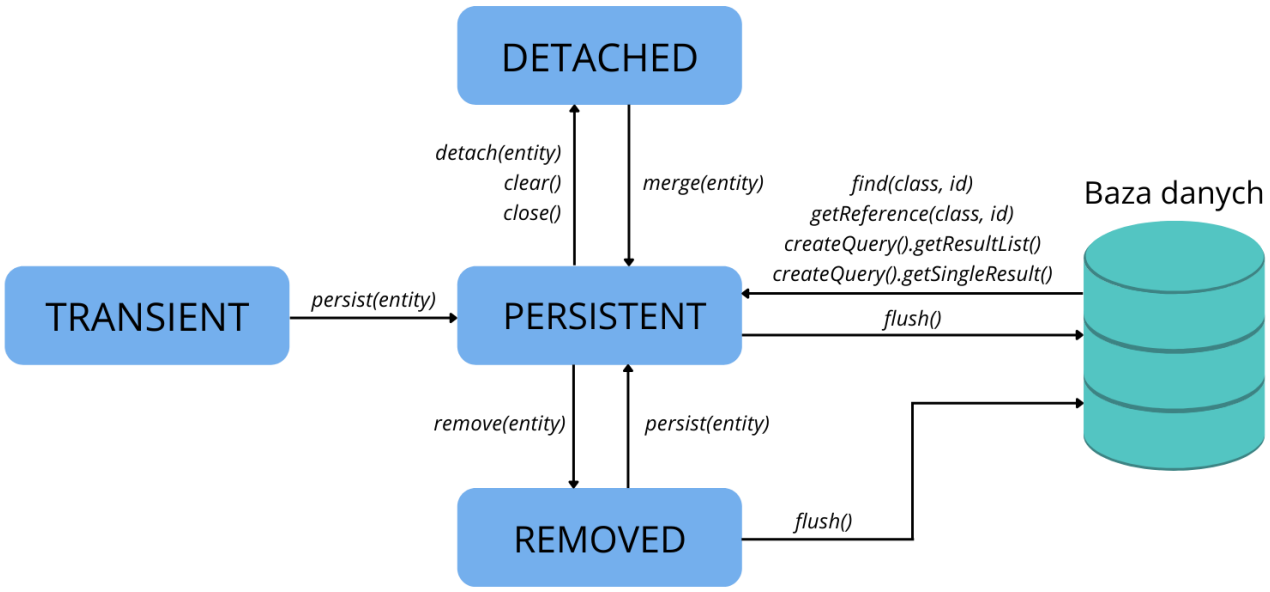
rozumieć, jakie DDL są faktycznie wykonywane na bazie danych i czemu. Drugi aspekt jest taki, że podejście z pisaniem DDL samodzielnie przybliża nas do stosowania narzędzi takich jak np. wspomniany wcześniej Flyway.

Kiedy natomiast mogą nam się przydać inne ustawienia niż **none**? Jeżeli przykładowo wykonywalibyśmy testy integracyjne na bazie danych i chcieli, żeby za każdym razem przed uruchomieniem testów, baza danych była "stawiana" od nowa, a na koniec cały schemat był usuwany - możemy wtedy wykorzystać `hibernate.hbm2ddl.auto`. Stosowanie tego ustawienia w aplikacji produkcyjnej doprowadziłoby do chaosu w bazie danych. Druga kwestia jest taka, że jeżeli w trakcie wykonywania takiej aktualizacji schematu wystąpi błąd, to schemat może zostać zaktualizowany tylko częściowo - ponownie, byłby to dosyć ciężki błąd do odkręcenia.

Cykl życia encji

Encje w rozumieniu Hibernate też mają swoje stany: *Transient*, *Persistent*, *Detached*, *Removed*. Za przejścia pomiędzy stanami odpowiadają poszczególne metody z *Session*.

Pamiętasz stany omawiane na etapie JPA? Teraz wrócimy do tego tematu na chwilę.

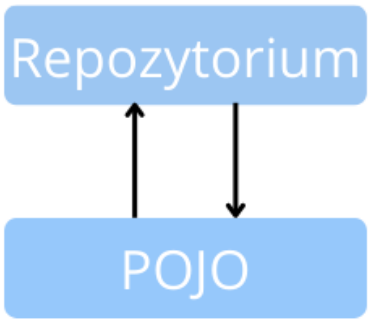


Obraz 5. Cykl życia encji z metodami z Session

Stany cyklu życia encji, jakie wyróżniamy to:

Transient

Encja, która nie ma swojej reprezentacji w *persistence context* i nie jest zarządzana w żadnej sesji. Obiekty te istnieją na stercie jako normalne obiekty Java.



Obraz 6. Entity state Trsansient

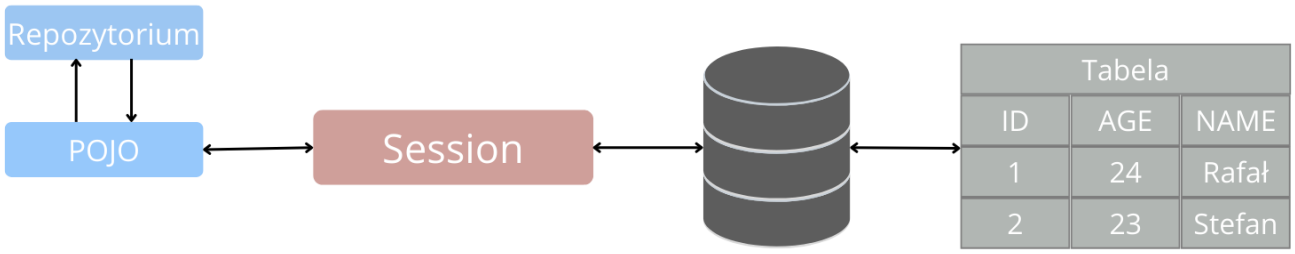
Persistent (*Managed*)

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 Session, 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.

Encja taka jest mapowana do konkretnego wiersza bazy danych, identyfikowanego przez klucz. Bieżąca sesja Hibernate jest odpowiedzialna za śledzenie wszystkich zmian dokonanych w zarządzanej encji i propagację tych zmian do bazy danych.

Encję w takim stanie możemy pozyskać na dwa sposoby:

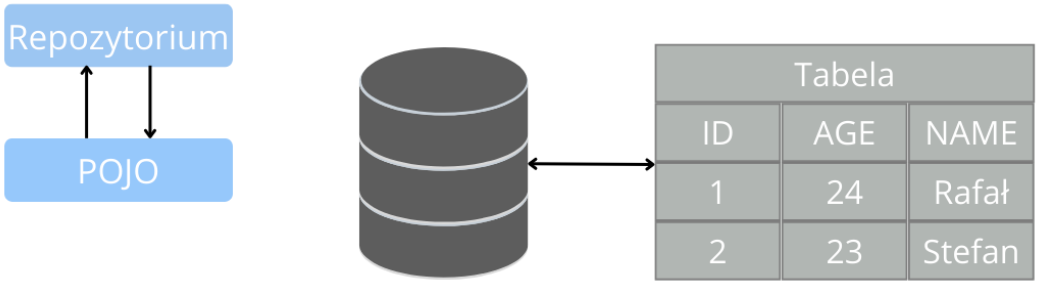
- Załadowanie encji z bazy danych, np. za pomocą metody `get()`,
- Encję w stanie Transient lub Detached możemy zapisać do bazy danych przy wykorzystaniu np. metody `persist()`.



Obraz 7. Entity state Persistent

Detached

Stan oznaczający, że encja ma reprezentację w bazie danych, ale nie jest już zarządzana przez Session i nie jest śledzona przez *persistence context*. Wszelkie zmiany w tej encji nie zostaną odzwierciedlone w bazie danych i na odwrót. Encję taką można utworzyć, zamykając sesję, z którą była powiązana, lub usuwając ją z sesji za pomocą wywołania metody `detach()`.



Obraz 8. Entity state Detached

Removed

Użycie metody *remove()* z Session ustawia encję w stan *Removed*, co oznacza, że po zakończeniu transakcji, odpowiedni wiersz w bazie danych zostanie usunięty.

Hibernate vs DAO vs Repository

Korzystając z okazji, chciałbym dodać tutaj wpis na temat *DAO vs Repository*. Przypomnij sobie całą filozoficzną dyskusję na ten temat i staraj się pamiętać o niej na etapie przyswajania materiałów o Hibernate. Wracaj do tego fragmentu, gdy przyjdzie taka potrzeba.

Teoria vs Praktyka

Przykład **DAO** vs **Repository** jest bardzo fajnym przykładem z filozoficznego punktu widzenia. Zacznijmy może od pewnego posta na Stackoverflow:

Cytując:

> DAO and Repository pattern are ways of implementing Data Access Layer (DAL). So, let's start with DAL, first.
>
> Object-oriented applications that access a database, must have some logic to handle database access. In order to keep the code clean and modular, it is recommended that database access logic should be isolated into a separate module. In layered architecture, this module is DAL.
>
> So far, we haven't talked about any particular implementation: only a general principle that putting database access logic in a separate module.

Komentarz Karola: Święte słowa, jeżeli separujemy kod dotyczący wywołań do baz danych, dążymy do rozdziału pojęć w kodzie. Dzięki temu mamy porządek i potem łatwiej się to wszystko ogarnia. Lecimy dalej z cytatem:

> Now, how we can implement this principle? Well, one know way of implementing this, in particular with frameworks like Hibernate, is the DAO pattern.
>
> DAO pattern is a way of generating DAL, where typically, each domain entity has its own DAO. For example, User and UserDao, Appointment and AppointmentDao, etc. An

> example of DAO with Hibernate: http://gochev.blogspot.ca/2009/08/hibernate-generic-dao.html.
>
> Then what is Repository pattern? Like DAO, Repository pattern is also a way achieving DAL. The main point in Repository pattern is that, from the client/user perspective, it should look or behave as a collection. What is meant by behaving like a collection is not that it has to be instantiated like Collection collection = new SomeCollection(). Instead, it means that it should support operations such as add, remove, contains, etc. This is the essence of Repository pattern.
>
> In practice, for example in the case of using Hibernate, Repository pattern is realized with DAO. That is an instance of DAL can be both at the same an instance of DAO pattern and Repository pattern.
>
> Repository pattern is not necessarily something that one builds on top of DAO (as some may suggest). If DAOs are designed with an interface that supports the above-mentioned operations, then it is an instance of Repository pattern. Think about it, If DAOs already provide a collection-like set of operations, then what is the need for an extra layer on top of it?

Komentarz Karola: Dwa ostatnie akapity są tutaj kluczowe. Wzorce te są do siebie tak podobne, że w praktyce **DAO** i **Repository** przeplatają się ze sobą w takim stopniu, że nie ma wyraźnej granicy pomiędzy **DAO** i **Repository**. Moim zdaniem wynika to ze zdrowego rozsądku. Gdy przejdziemy do omówienia *Spring Data JPA*, to zobaczysz, że wiele metod typu `read()`, `save()`, czy `delete()` będzie Ci dostarczone przez Spring i nie będziesz musiał/musiała ich pisać ręcznie. Z tego powodu w praktyce "merguje" się ze sobą **DAO** i **Repository** i często są one traktowane jak jedno i to samo. Zrozumiesz, co mam na myśli, dopiero po omówieniu warsztatu *Spring Data JPA*. Dlatego właśnie w przykładach i projektach Zajavkowych, będziemy je ze sobą mieszać, tak samo, jak jest to wymieszane w praktyce.

Czemu tak? Pamiętaj, że w Zajavce chcemy Ci przybliżać, jak wygląda praca w praktyce. A w praktyce to normalne, że wzorce teoretyczne sobie istnieją, a zespół implementuje je po swojemu ☺. Oczywiście do pewnego stopnia, ale prawda jest taka, że nie zawsze w realnym kodzie znajdziesz w 100% odwzorowane teoretyczne wzorce projektowe. A bo tak było wygodniej albo z jeszcze innego powodu.

Konwencje

Dlatego właśnie w tym przypadku podjęliśmy decyzję, żeby pokazać Ci, czego możesz spodziewać się w praktyce, pomimo że teoria tych wzorców mówi trochę co innego, niż to, jak przedstawimy to w przykładowych rozwiązaniach.

W przykładach Zajavkowych będziemy stosować kilka konwencji, które będziemy ze sobą przeplatać. Wszystko po to, żeby przyzwyczaić Cię do tego, jak to będzie wyglądało w praktyce (czyli nie trzymamy się sztywno wzorców **DAO** i **Repository**, bo chyba nie ma projektów, gdzie wszystko jest zrobione zgodnie ze sztuką ☺).

Opiszę te konwencje słownie. Po przeczytaniu każdej spróbuj porównać to z tym, co już wiesz, z resztą ten sam fragment zostanie celowo skopiowany do kolejnych dwóch warsztatów, żeby sprowokować Cię do zastanowienia się nad tym po zapoznaniu się z nowymi mechanizmami.

1. Konwencja polegająca na tym, że nie ma **DAO**, są same **Repository**. W tym przypadku nie trzymamy się sztywno tego, że **Repository** ma być *collection-like*. Repository służy do szeroko rozumianego dostępu do danych. Wyróżniamy wtedy interfejsy **Repository** i klasy implementujące te interfejsy, gdzie nazwa klasy określa, że dane są przechowywane w bazie danych, a nie np. w pliku. Takie podejście będzie stosowane projekcie w warsztacie *Spring Data Access*, czyli o w tym właśnie.
2. Konwencja polegająca na tym, że interfejsy z dopiskiem **DAO** w nazwie będą rozumiane jako ogólny sposób dostępu do danych. Nie ma to znaczenia, czy te dane są przechowywane w bazie danych, czy w pliku, **DAO** ma tylko umożliwić szeroko rozumiany dostęp do danych, gdzie interfejsy **DAO** są wykorzystywane przez serwisy. Nie będziemy się również sztywno trzymać konwencji CRUD, czyli w interfejsach **DAO** będą dodane dowolne metody. Do tego będziemy tworzyć klasy z **Repository** w nazwie, które implementują interfejsy z **DAO** w nazwie. Klasy **Repository** będą służyły do faktycznej realizacji komunikacji z bazą danych. Równie dobrze takie klasy mogłyby implementować operacje na danych w plikach i wtedy moglibyśmy zaimplementować taki interfejs dwiema klasami **DatabaseRepository** oraz **FileRepository**. W tym przypadku pamiętajmy, że **DAO** oznacza generalny sposób na dostęp do danych, które to dane są zapisane "gdzieś", a klasy z **Repository** w nazwie, zawierają "bebechy" do komunikacji z bazą danych.
3. Kolejna konwencja (często stosowałem ją w praktyce), w której są 2 poziomy **Repository**, poziom bliżej danych i poziom bliżej serwisów. Jeden poziom **Repository** jest bliżej związany z mechanizmami Springa (to ten bliżej danych) i zapewnia bezpośrednio dostęp do danych w bazie, nazwijmy go poziomem niższym. Drugi poziom **Repository** grupuje pod spodem wywołania do niższych poziomów **Repository** i zapewnia bardziej biznesowy sposób na pracę z pobranymi danymi. Nie pojawiała się tutaj nazwa **DAO**, tylko samo **Repository**. Czy może już Ci coś świta? Podpowiem, omawialiśmy to wyżej w części teoretycznej, ale klasy/interfejsy zawierały tam **DAO** w nazwie.

Przedstawione konwencje są tylko przykładami, w praktyce możesz spotkać wiele innych. To zespół na etapie realizacji projektu podejmuje decyzje, w jaki sposób chce pracować z danymi i jakie nazewnictwo będzie stosował w kodzie. Co więcej, wielu programistów uważa, że dodawanie **DAO** jest zbędne, bo tylko zaciemnia obraz i woli pracować na samych repozytoriach (na jednym poziomie, a nie dwóch). Inni z kolei wolą dodawać kilka warstw repozytoriów (przypadek trzeci z powyższych), ale nie użyją nazwy **DAO**, tylko **Repository** (nawet w interfejsach), pomimo że teoretycznie pracują z **DAO**, to nazwą je **Repository**.

W takim przypadku pojawia się pytanie, czy sposób trzeci implementuje wzorzec projektowy **Repository**, skoro układ klas jest zgodny ze wzorcem, ale nazewnictwo klas już nie? Moim zdaniem tak, ale to już jest kwestia do filozoficznej dyskusji ☺.

Ja z kolei w przykładach implementacji projektów specjalnie wydzielę interfejsy z nazwą **DAO**, ~~żeby Ci namieszać w głowie~~ żeby dać Ci do myślenia i sprowokować lepsze zapamiętanie tego przypadku, nawet jeżeli będzie to trochę na przekór przedstawionej teorii. Zapis **DAO** będzie przeze mnie rozumiany jako sposób na dostęp do danych. Nie ma znaczenia, gdzie te dane będą, gdzieś będą. Interfejsy **DAO** będą następnie implementowane przez **Repozytoria**, żeby pokazać, że sposób na pracę z danymi jest zrealizowany poprzez bazę danych, a nie np. plik na dysku.

Co więcej, "kontrowersja", jaką wprowadzę, będzie skupiała się wokół nazewnictwa. W przyjętej konwencji mógłbym równie dobrze nie używać nazw **DAO**, tylko napisać **Repository**, a jednocześnie stosować opisany przeze mnie wzorzec **Repository** z agregowaniem wywołań do repozytoriów pod spodem.

Wspomniane wyżej konwencje będą stawały się bardziej jasne, jak będziesz iść dalej z materiałem i realizować kolejne etapy projektów. Pamiętaj, że to, co jest tutaj przedstawiane to tylko przykłady. W Twoich własnych projektach możesz napisać to inaczej, czyli np. nie używać **DAO**, a same repozytoria. Ważne jest to, żeby zachować tą warstwę abstrakcji w dostępie do danych i żeby Twoje założenia były zrozumiałe dla innych.

Postarałem się tutaj zebrać dużo wyjaśnień, ale niestety, cała ta wiedza przyjdzie Ci z praktyką, wyrobisz sobie też wtedy konkretne zdanie na różne podejścia.