

JDBC i Architektura Oprogramowania

Spis treści

Przypomnienie czym jest JDBC	1
Rodzaje zapytań SQL	2
Architektura oprogramowania i architektura trójwarstwowa	3
DAO	5
Teoria	6
Interfejs DAO	7
Implementacja DAO	7
Repository	9
DAO vs Repository	9
Komplikujemy przypadek	10
Wnioski	12
Teoria vs Praktyka	12
Ścieżka zajavka	16

Przypomnienie czym jest JDBC



Pamiętasz JDBC? Jeżeli nie pamiętasz, to zaczniemy od przypomnienia, bo wracamy do tej tematyki, tylko, że tym razem będzie nam pomagał Spring. Podczas omawiania zagadnień dotyczących JDBC nie poruszaliśmy też tematyki transakcji bazodanowych - zostanie to poruszone w obrębie tego warsztatu.

Java Database Connectivity, JDBC to API, które zapewnia nam podstawowe interfejsy, pozwalające na podłączenie się do bazy danych przy wykorzystaniu Javy. Do poprawnego działania, z konkretną wersją bazy, potrzebna jest implementacja tych interfejsów, czyli odpowiedni sterownik bazy danych w postaci pliku *.jar*. Sterownik należy pobrać z internetu, najlepiej z oficjalnego źródła, dodać do projektu jako biblioteka i gotowe, możemy łączyć się do bazy. Tylko do której? Na to pytanie odpowiada **JDBC URL**, czyli adres bazy, do której chcemy się połączyć. Struktura takiego adresu wygląda w ten sposób:

```
<protokół>:<rodzaj_bazy_danych>://<adres_bazy_danych>/<nazwa_bazy_danych>
```

A z rzeczywistymi wartościami adres mógłby wyglądać w taki sposób:

```
jdbc:postgresql://localhost:5432/zajavka
```

- **jdbc** — nazwa protokołu, jaki wykorzystujemy do połączenia

- **postgres** — nazwa bazy danych, do jakiej chcemy się połączyć
- **localhost** — nazwa hosta (inaczej strony internetowej)
- **5432** — port, na którym baza danych jest dostępna
- **zajavka** — nazwa bazy danych

I poniżej szybkie przypomnienie użycia *JDBC*:

```
String url = "jdbc:postgresql://localhost:5432/zajavka";
String user = "postgres";
String password = "password";

Connection conn = DriverManager.getConnection(url, user, password); ①
Statement statement = connection.createStatement();

Integer numberOfModifiedRows = statement.executeUpdate(query); ②
ResultSet resultSet = statement.executeQuery(query); ②
boolean resultSetExists = statement.execute(query); ②
```

① Utworzenie połączenia do bazy.

② Różne query, w zależności od rodzaju SQL, jaką chcemy wykonać.

Do jakich rodzajów zapytań *query* można stosować określone metody:

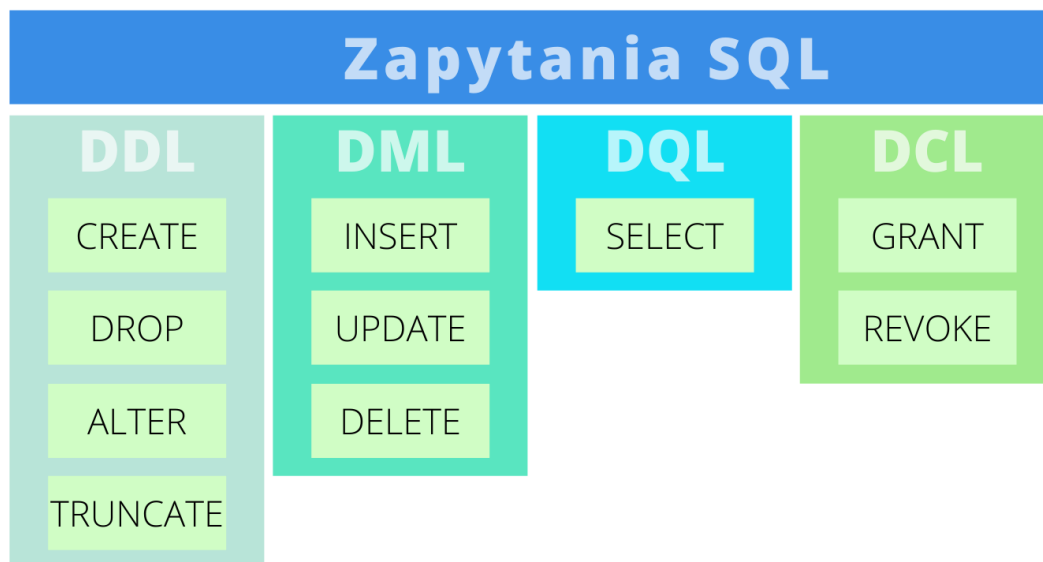
Metoda	DDL	DML	DQL	DCL
<i>execute()</i>	TAK	TAK	TAK	TAK
<i>executeUpdate()</i>	TAK	TAK	NIE	NIE
<i>executeQuery()</i>	NIE	NIE	TAK	NIE

Rodzaje zapytań SQL

Chciałbym, żebyśmy do wiedzy, którą posiadamy w tym momencie dodali też podział samych zapytań SQL.

Zapytania SQL (statements) dzieli się na cztery podzbiory:

- **DDL** (ang. Data Definition Language)
- **DML** (ang. Data Manipulation Language)
- **DQL** (ang. Data Query Language)
- **DCL** (ang. Data Control Language)



Obraz 1. Zapytania SQL

Poniżej znajdziesz informacje dotyczące tego, co robi każde ze wspomnianych zapytań. Nie wszystkie z nich wykorzystywaliśmy dotychczas w praktyce, ale warto wiedzieć, że oprócz zapytań, które były wcześniej poruszane, dostępnych jest dużo więcej możliwości:

- **CREATE** - ten rodzaj zapytania służy do tworzenia nowej tabeli w bazie danych,
- **DROP** - ten rodzaj zapytania służy do usuwania tabeli z bazy danych,
- **ALTER** - ten rodzaj zapytania służy do modyfikacji tabeli w bazie danych. Możemy go używać do dodawania, modyfikowania oraz usuwania kolumn w tabeli. Możemy w ten sposób również dodawać lub usuwać constrainty z bazy danych,
- **TRUNCATE** - ten rodzaj zapytania służy do usuwania danych z tabeli z bazy danych bez usuwania tabeli samej w sobie,
- **INSERT** - ten rodzaj zapytania służy do nowych rekordów do tabeli w bazie danych,
- **UPDATE** - ten rodzaj zapytania służy do aktualizowania rekordów w tabeli w bazie danych,
- **DELETE** - ten rodzaj zapytania służy do usuwania rekordów z tabeli w bazie danych,
- **SELECT** - ten rodzaj zapytania służy do pobierania rekordów z tabeli w bazie danych,
- **GRANT** - ten rodzaj zapytania służy do nadania określonych uprawnień konkretnemu użytkownikowi w bazie danych,
- **REVOKE** - ten rodzaj zapytania służy do usunięcia określonych uprawnień konkretnemu użytkownikowi w bazie danych,

Architektura oprogramowania i architektura trójwarstwowa

Skoro przypomnieliśmy już sobie na czym polegało JDBC, chciałbym żebyśmy na chwilę przeszli do tematu **architektury oprogramowania**.



Powołując się na [Wikipedię](#), czym jest **Architektura Oprogramowania**? Jest to

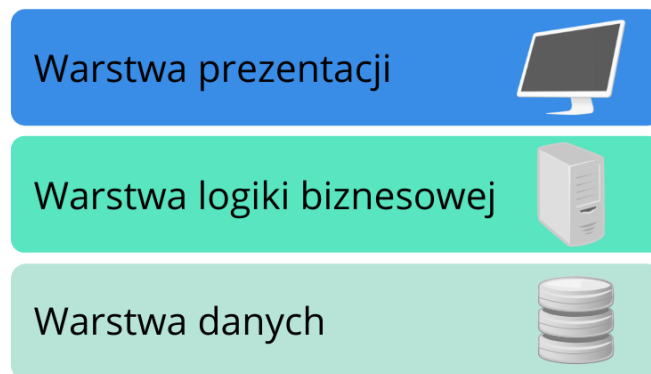
podstawowa organizacja systemu wraz z jego komponentami, wzajemnymi powiązaniami, środowiskiem pracy i regułami ustanawiającymi sposób jej budowy i rozwoju.

Czyli w uproszczeniu, konkretna **architektura aplikacji** (**architektura oprogramowania**) odnosi się do tego w jaki sposób nasza aplikacja jest zorganizowana, które komponenty aplikacji mają do siebie dostęp. Można to zrozumieć troszkę jak wzorce projektowe, gdzie konkretny wzorzec oznaczał pewien schemat relacji pomiędzy obiektami. Tutaj możemy stwierdzić, że architektura aplikacji oznacza pewien rodzaj relacji pomiędzy częściami systemu. W świecie oprogramowania można wyróżnić wiele rodzajów architektury, natomiast chciałbym, żebyśmy skupili się teraz na **Architekturze Trójwarstwowej**.

Rodzajów architektury oprogramowania jest wiele. Wprowadźmy zatem jeden z podstawowych rodzajów architektury aplikacji.

Architektura trójwarstwowa (ang. *three-tier architecture* / *three-layer architecture*) to koncepcja prostego podziału aplikacji na trzy warstwy. Każda z warstw może działać niezależnie od innych na własnej infrastrukturze, dlatego mogą być rozwijane i modyfikowane oddzielnie np. przez różne zespoły deweloperskie. Przez wiele lat ten model był dominujący dla aplikacji typu klient-serwer. Na razie nie zagłębialiśmy się w wyjaśnienie architektury klient-serwer, wrócimy do tego później.

Jeżeli chodzi o architekturę warstwową, to możemy wyróżnić więcej warstw niż te wspomniane, dlatego od tego momentu będziemy mówić po prostu o architekturze warstwowej. W architekturze trójwarstwowej, możemy wyróżnić następujące warstwy.



Obraz 2. Architektura trójwarstwowa

- Warstwa **prezentacji** — dostarcza interfejs dla użytkownika i jest miejscem, w którym użytkownik może wejść w interakcję z systemem,
- Warstwa **logiki biznesowej** — zawiera w sobie wszystkie reguły biznesowe, przetwarza dane wejściowe użytkownika i wysyła zapytania do bazy danych,
- Warstwa **danych** — to miejsce przechowywania informacji przetwarzanych przez aplikację, czyli baza danych.

Zalety architektury trójwarstwowej:

- Szybkość tworzenia aplikacji — podział pozwala na równoczesny rozwój każdej z warstw jednocześnie

- Skalowalność — każda warstwa może być **skalowalna** niezależnie od innych.
- Niezawodność — awaria jednej z warstw nie dotyka innych warstw.
- Bezpieczeństwo — tak jak widać na rysunku, warstwa prezentacji nie ma bezpośredniego dostępu do warstwy danych, dzięki temu warstwa środkowa działa jak dodatkowa zaporę na różne ataki.

Do tematu architektury będziemy jeszcze wracać, wyjaśnimy natomiast, czym jest **DAO**.

DAO

W praktyce spotkasz się z określeniem **DAO** (*Data Access Object*). Bardzo dobrą definicję tego określenia możemy znaleźć na [Wikipedii](#):

In software, a data access object (DAO) is a pattern that provides an abstract interface to some type of database or other persistence mechanism. By mapping application calls to the persistence layer, the DAO provides some specific data operations without exposing details of the database. This isolation supports the single responsibility principle. It separates what data access the application needs, in terms of domain-specific objects and data types (the public interface of the DAO), from how these needs can be satisfied with a specific DBMS, database schema, etc. (the implementation of the DAO).



Tutaj przyznam się szczerze, że omawiając ten temat po raz pierwszy, przyjąłem bardzo mocne uogólnienie wynikające z tego, że teoria teorią, a praktyka praktyką. Kierując się doświadczeniami z praktyki, uogólniłem kilka kwestii. Natomiast w związku z tym tematem, wywiązała się spora dyskusja na naszym Discordzie, która zachęciła mnie do głębszego omówienia tego zagadnienia. Do tematu **DAO** (i **Repository**) będziemy jeszcze wracać w notatkach z kolejnych warsztatów, dlatego zaznaczam, że ta notatka nie wyczerpuje tematu.

DAO, jest wzorcem projektowym, natomiast jest to o tyle specyficzne zagadnienie, że jak już trafisz w praktyce na jakiś projekt, to **DAO** może być zaimplementowane i rozumiane przez zespół kompletnie inaczej (w poprzek i do góry nogami), niż mówi o tym teoria tego wzorca. Początkowo nie chciałem wchodzić w ten temat jakoś mocno głęboko, natomiast tak jak wspomniałem, zachęciła mnie do tego dyskusja na Discord.

Zacznę zatem od bardzo ogólnego podejścia do tematu (trochę wbrew temu, co można znaleźć w internecie szukając wyjaśnienia *DAO vs Repository*), a potem przejdziemy do szczegółów.

Przyjmując duży poziom uogólnienia i uproszczenia można powiedzieć, że naszym **DAO** są klasy takie jak np. **OwnerRepository** lub **EmployeeRepository**, bo służą one do zapewnienia dostępu do danych. Poniższe podejście będzie częściej wykorzystywane, gdy przejdziemy do omówienia projektu w warsztacie o *Spring Data JPA*, natomiast moglibyśmy podejść do tworzenia obiektów służących do otrzymywania danych w ten sposób:

Interfejs EmployeeDAO

```
public interface EmployeeDAO {  
    List<Employee> getAllEmployees();  
}
```

```
}
```

Klasa *EmployeeDatabaseRepository*

```
public class EmployeeDatabaseRepository implements EmployeeDAO {  
    public List<Employee> getAllEmployees() {  
        //... implementacja z bazy danych  
    }  
}
```

Klasa *EmployeeFileRepository*

```
public class EmployeeFileRepository implements EmployeeDAO { ①  
    public List<Employee> getAllEmployees() {  
        //... implementacja z pliku  
    }  
}
```

① Zwróć uwagę na zapis **Repository implements DAO**, bo niżej sam się do siebie przyczepię o ten zapis.

Zatem przyjmując duży poziom uogólnienia i uproszczenia można powiedzieć, że **DAO** jest obiektem, który zapewnia dostęp do źródła danych. Nie ma to znaczenia, czy źródłem jest plik, czy baza danych. Jeżeli będziemy odwoływali się do takiego obiektu **DAO** z poziomu interfejsu, to nie ma to znaczenia z naszej perspektywy, czy implementacja jest w pliku, czy w bazie danych. Nas interesują dane, na których pracujemy, nie ma znaczenia, skąd one będą.



Jeżeli w tym momencie zaczniesz szukać w Internecie wyjaśnień na temat *DAO* vs *Repository*, to wszystko zacznie się płatać, bo teoria, którą znajdziesz w Internecie, kłóci się z tym, co zostało przedstawione wyżej. Kłóci się, bo to, co zostało pokazane wyżej to bardzo mocne uogólnienie. Przejdźmy zatem do dokładnego wyjaśnienia, co i jak.

Teoria

Wzorzec **DAO** jest wzorcem strukturalnym (*structural pattern*), który pozwala nam odizolować warstwę biznesową aplikacji od warstwy dostępu do danych za pomocą jakiegoś poziomu abstrakcji. Wykorzystanie tego wzorca pozwala na ukrycie przed warstwą biznesową całą złożoność wykonywania operacji CRUD (Create Read Update Delete) zawartą w mechanizmie realizującym dostęp do danych aplikacji. Jak każdy poziom abstrakcji, tak i ten pomaga w tym, żeby wspomniane warstwy aplikacji (warstwa biznesowa i warstwa dostępu do danych) mogły być rozwijane oddzielnie i nie musiały wiedzieć o sobie nawzajem.

Żeby móc lepiej zrozumieć teorię, pokażmy podstawowy przykład aplikacji, która zarządza użytkownikami. Jednocześnie chcemy (dla porządku i przyszłego łatwiejszego rozwoju aplikacji) zapewnić, żeby klasy domenowe (model domenowy) aplikacji były kompletnie niezależne od bazy danych. W tym celu stworzymy prostą klasę **DAO**, która zadba o to, żeby ten rozdział był zaimplementowany.

Po pierwsze, stworzymy klasę **User**, która będzie elementem modelu domenowego aplikacji.

```
@Data
public class User {

    private String name;
    private String email;

}
```

Klasa `User` reprezentuje użytkownika naszej aplikacji i nie robi nic więcej. Jest *kontenerem* na dane użytkownika w programie.

I tutaj zaczyna się cała zabawa, czyli jak oddzielić od siebie warstwę biznesową aplikacji, która pracuje z klasą `User` od mechanizmu dostępu do danych, który gdzieś w tej aplikacji jest zaimplementowany. Do tego służy właśnie wzorzec **DAO**.

Interfejs DAO

Jak już wiesz na tym etapie, żeby oddzielić w jakiś sposób implementację i pracować na abstrakcji (kłania się **DIP**), możemy wprowadzić interfejs. Wprowadźmy zatem interfejs **DAO**.

```
public interface Dao<T> {

    Optional<T> read(long id);

    void create(T t);

    void update(T t, String[] params);

    void delete(T t);

}
```

Z tego przykładu widać, że **DAO** definiuje abstrakcyjne metody dostępu do danych, które pozwalają nam na przeprowadzenie operacji typu CRUD na obiekcie `T`. Jak już możesz się domyślić, możemy w tym momencie zaimplementować ten interfejs, określając konkretne zachowania pod implementowanymi metodami.

Implementacja DAO

W tym celu wprowadzimy klasę `UserDAO`. Dla prostoty, w naszej implementacji zamiast mechanizmów bazodanowych wykorzystamy Listę. Tworzone obiekty będą przechowywane w pamięci programu.

```
public class UserDao implements Dao<User> {

    private List<User> users = new ArrayList<>();

    public UserDao() {
        users.add(new User("Zajavka User 1", "user1@zajavka.pl"));
        users.add(new User("Zajavka User 2", "user2@zajavka.pl"));
    }

    @Override
    public Optional<User> read(long id) {
```

```

        return Optional.ofNullable(users.get((int) id));
    }

    @Override
    public void create(User user) {
        users.add(user);
    }

    @Override
    public void update(User user, String[] params) {
        user.setName(Objects.requireNonNull(params[0], "Name cannot be null"));
        user.setEmail(Objects.requireNonNull(params[1], "Email cannot be null"));
        users.add(user);
    }

    @Override
    public void delete(User user) {
        users.remove(user);
    }
}

```

W przedstawionym podejściu, klasa `UserDAO` implementuje wszystkie metody wymagane do pracy na danych w kontekście klasy `User`. Tak jak wspomniałem wcześniej, dla uproszczenia, zamiast bazy danych, użytkownicy są zapisywani w pamięci programu.

Jak w takim razie można wykorzystać klasę `UserDAO` w naszym programie?

```

@AllArgsConstructor
public class UserService {

    private Dao<User> userDao;

    public void manageUser() {
        User user0 = userDao.read(0)
            .orElseGet(() -> new User("non-existing user", "email-missing"));
        userDao.update(user0, new String[]{"Zajavka User Updated", "userUpdated@zajavka.pl"});

        User user1 = userDao.read(1)
            .orElseGet(() -> new User("non-existing user", "email-missing"));
        userDao.delete(user1);

        userDao.create(new User("Zajavka User New", "userNew@zajavka.pl"));

        userDao.getAll()
            .forEach(user -> System.out.println(user.getName()));
    }
}

```

Oczywiście ten przykład jest bardzo uproszczony, ale pokazuje w skrócie cel wzorca **DAO**. Klasa z warstwy biznesowej `UserService` używa (przez interfejs) wzorca **DAO** do wykonywania prostych CRUDów na obiektach `User`.

Najistotniejsze z tego przykładu jest to, że klasa serwisu (czyli warstwa biznesowa) ma ukrytą przed sobą implementację (szczegóły) dostępu do mechanizmu przechowywania danych, bo nie jest jej to do niczego potrzebne. Serwis chce tylko zapisać gdzieś dane, na których pracuje, nie interesuje go czy będą one zapisane w bazie danych, czy w pliku. Mają być zapisane "gdzieś".

Repository

Wcześniej, przy przykładzie o **DAO** padło również hasło **Repository**. Znamy już ten termin, chociażby po omówieniu Gita, ale spójrzmy na niego w perspektywie bazy danych. Cytując [źródło](#):

As per Eric Evans' book Domain-Driven Design, the "repository is a mechanism for encapsulating storage, retrieval, and search behavior, which emulates a collection of objects."

Likewise, according to Patterns of Enterprise Application Architecture, it "mediates between the domain and data mapping layers using a collection-like interface for accessing domain objects."

In other words, a repository also deals with data and hides queries similar to DAO. However, it sits at a higher level, closer to the business logic of an app.

Consequently, a repository can use a DAO to fetch data from the database and populate a domain object. Or, it can prepare the data from a domain object and send it to a storage system using a DAO for persistence.

Okej, czyli w sumie podobne, ale na wyższym poziomie abstrakcji. *To w sumie, jaka jest różnica? I czemu Karol wyżej napisał, że Repository implements DAO.*

DAO vs Repository

Przejdziemy teraz do porównania tych dwóch. Oczywiście zrobimy to na teoretycznym poziomie, bo ten temat jest o tyle specyficzny, że w praktyce chyba nie widziałem żadnego projektu, w którym te wzorce byłyby zaimplementowane zgodnie z teorią. Zawsze była to jakaś własna twórczość zespołu, ale do tego jeszcze przejdziemy.

Wyżej został przedstawiony już przykład implementacji wzorca **DAO**, więc napiszmy tylko jeszcze raz, że **DAO** jest poziomem abstrakcji, który umożliwia prosty sposób na pracę z danymi, ukrywając jednocześnie "bebechy" przed logiką biznesową.

Jeżeli natomiast chcielibyśmy zaimplementować wzorzec **Repository** w kontekście tego samego przykładu **User**, zrobilibyśmy to np. w ten sposób (klasa **User** pozostaje taka sama jak w przykładzie z **DAO**):

```
public interface UserRepository {  
    User get(Long id);  
    void add(User user);  
    void update(User user);  
    void remove(User user);  
}
```

Zwróć uwagę, że nie mamy tutaj typowych metod **CRUD**, tylko metody podobne jak w kolekcji `get()`, `add()`, `update()`, `remove()` (w cytacie powyżej pojawił się taki fragment: *data mapping layers using a collection-like interface for accessing domain objects*).

Jeżeli teraz dołożymy implementację tego interfejsu:

```
public class UserRepositoryImpl implements UserRepository {

    private Dao<User> userDao;

    @Override
    public User get(Long id) {
        return userDao.read(id);
    }

    @Override
    public void add(User user) {
        userDao.create(user);
    }

    // i tak dalej ...
}
```

Przy implementacji `UserRepositoryImpl` wykorzystaliśmy `Dao<User>`, żeby zapewnić dostęp do danych z bazy danych.

Na tym etapie repozytorium wygląda jak kolejna warstwa abstrakcji w dostępie do danych, opakowująca warstwę **DAO**. Skomplikujmy zatem nieco ten przykład.

Komplikujemy przypadek

Oprócz klasy `User` wprowadzimy też klasę `Product`:

```
@Data
public class Product {

    private String description;
    private BigDecimal price;
}
```

Skoro wprowadziliśmy klasę `Product`, to musimy również wprowadzić do niej **DAO**.

```
public class ProductDao implements Dao<Product> {

    private List<Product> products = new ArrayList<>();

    public ProductDao() {
        products.add(new Product("some product 1", new BigDecimal("120.30")));
        products.add(new Product("some product 2", new BigDecimal("91.20")));
        products.add(new Product("some product 3", new BigDecimal("75.00")));
    }

    @Override
    public Optional<Product> read(long id) {
        return Optional.ofNullable(products.get((int) id));
    }

    @Override
```

```

public void create(Product product) {
    products.add(product);
}

@Override
public void update(Product product, String[] params) {
    product.setDescription(Objects.requireNonNull(params[0], "Description cannot be null"));
    product.setPrice(Objects.requireNonNull(new BigDecimal(params[1]), "Price cannot be null"));
    products.add(product);
}

@Override
public void delete(Product product) {
    products.remove(product);
}
}

```

Jak widać, **DAO** służy nam ponownie do manipulacji na obiektach typu **Product**, gdzie mówiąc manipulacja mamy na myśli zarządzanie tymi obiektami np. w bazie danych. Ponownie, dla uproszczenia przykładu, zamiast kodu odpowiedzialnego za komunikację z bazą danych, wykorzystaliśmy listę do przechowywania obiektów w pamięci programu.

Na potrzeby tego przykładu dodajmy jeszcze klasę **UserWithProducts**

```

@Data
public class UserWithProducts extends User {
    private List<Product> products;
}

```

Klasa ta reprezentuje kolejny obiekt w naszym programie, który oprócz danych użytkownika będzie zawierał również informację o zakupionych produktach. Możemy teraz stworzyć implementację **UserRepository** uwzględniającą te zmiany.

```

public class UserRepositoryImpl implements UserRepository {

    private Dao<User> userDao;
    private Dao<Product> productDao;

    @Override
    public User get(Long id, Long productId) { ①
        UserWithProducts user = (UserWithProducts) userDao.read(id);

        List<Product> products = productDao.read(productId);
        user.setProducts(products);

        return user;
    }
}

```

① Rozszerzamy listę parametrów tej metody. W domyśle, w interfejsie zrobiliśmy to samo, ale nie dodawałem już tego w przykładzie, żeby ilość kodu nie "puchła".

W przedstawionym przypadku **Repository** jest poziomem abstrakcji, który przykrywa (Facade) wywołania do konkretnych **DAO** i grupuje "logikę" pobierania danych w jednym miejscu. Zestawy

poszczególnych informacji są agregowane i repozytorium zwraca obiekt `UserWithProducts`, który następnie może być wykorzystany przez `UserService`. Repozytorium opiera się na **DAO** w celu pozyskania danych z różnych źródeł (np. różnych tabel w bazie danych).

Wnioski

Jakie wnioski można wyciągnąć z tych przykładów?

- **DAO** jest poziomem abstrakcji, który służy do zapewnienia dostępu do danych, **Repozytorium** natomiast jest poziomem abstrakcji, który zachowaniem przypomina kolekcję obiektów.
- **DAO** jest koncepcją na niższym poziomie abstrakcji, podczas gdy **Repository** na wyższym. **Repository** jest bliżej obiektów domenowych.
- **DAO** służy do ukrycia w kodzie brzydkich zapytań, które są potrzebne do gadania z bazą danych, podczas gdy **Repository** może być takim pośrednikiem pomiędzy **DAO** a warstwą logiki biznesowej (`UserService`).
- **DAO** nie jest implementowane przy wykorzystaniu **Repository**, natomiast **Repository** może pod spodem wykorzystywać **DAO**, żeby zapewnić dostęp do danych.
- ~~Karol coś narąbał w tłumaczeniu i pomieszał koncepcje, bo napisał `Repository implements DAO`~~ - Nie i właśnie do tego chcę teraz przejść.

Teoria vs Praktyka

Przykład **DAO** vs **Repository** jest bardzo fajnym przykładem z filozoficznego punktu widzenia. Zaczniemy 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 (pojawi się tutaj stwierdzenie Hibernate, będziemy o nim rozmawiać w kolejnych warsztatach):

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 przybliżyć, 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 w 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 wydzieliłem 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 stawiały 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 tę 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.

Czytamy dalej

Źródła, z którymi warto się zapoznać, jeżeli chodzi o ten temat (DAO vs Repository):

- [Stackoverflow](#),
- [Baeldung](#),
- [DevCezz](#)

A co ma w tej kwestii do powiedzenia ChatGPT?

Zapytałem również ChataGpt, co uważa na ten temat. Zaznaczam tylko, że podana przez niego odpowiedź będzie w pełni zrozumiała po warsztacie *Spring Data JPA*.

Karol:

Can I write a Spring Repository that implements an interface which is DAO?

ChatGpt:

Yes, you can write a Spring Repository that implements an interface which is a DAO (Data Access Object). In fact, this is a common approach used in Spring-based applications.

The DAO pattern is used to separate the data access code from the rest of the application logic. It typically involves defining an interface that specifies the operations that can be performed on a data source (such as a database), and then implementing that interface in one or more classes that handle the actual data access.

Spring provides a number of features that make it easy to work with DAOs, including the Repository pattern. A Spring Repository is an interface that extends the JpaRepository interface, which provides a set of CRUD (Create, Read, Update, Delete) operations for working with persistent entities.

To implement a Spring Repository that serves as a DAO, you can simply define your interface to extend JpaRepository (or one of its sub-interfaces) and add any additional methods specific to your data access needs. Then, you can create a class that implements that interface and provides the actual data access code.

For example, suppose you have a User entity and you want to create a DAO to handle CRUD operations on that entity. You could define a UserRepository interface as follows:

```
public interface UserRepository extends JpaRepository<User, Long> { ①  
  
    User findByUsername(String username);
```



```
}
```

① O co chodzi z tym `JpaRepository`, dowiesz się w warsztacie o Spring Data JPA.

This interface extends `JpaRepository` and adds a custom method to find a user by their username. You could then create a class that implements this interface and provides the data access code:

```
@Repository
public class UserRepositoryImpl implements UserRepository {

    private final EntityManager entityManager;

    @Autowired
    public UserRepositoryImpl(EntityManager entityManager) {
        this.entityManager = entityManager;
    }

    @Override
    public User findByUsername(String username) {
        // Implement data access code here
        // ...
    }

    // Implement other methods from JpaRepository here
    // ...
}
```

In this class, we use the `@Repository` annotation to indicate that it should be treated as a Spring Repository. We also use `@Autowired` to inject an `EntityManager` instance, which we can use to interact with the database.

Overall, using Spring Repositories to implement DAOs is a powerful and flexible approach that can simplify your data access code and make it easier to maintain over time.

Komentarz Karola: ChatGpt pokazał jeszcze inne podejście w tym samym temacie, w stosunku do tych, które opisałem wyżej. Tak jak zaznaczyłem, pełne zrozumienie podanego przez niego przykładu będziesz mieć po warsztacie o *Spring Data JPA*.

Ścieżka zajavka

Jak wspomniane informacje o architekturze mają się do ścieżki Zajavka?

Dawno o tym nie wspominałem, przypomnijmy zatem, że aplikacje webowe można podzielić na warstwy frontend i backend. My ucząc się Javy, cały czas rozmawiamy o niej w kontekście backendu. Dopiero w tym momencie zaczynamy dochodzić do tematów, przy których będziemy w stanie konkretnie wyjaśniać, czym jest ten backend. Dlatego właśnie przywołana została **Architektura Trójwarstwowa**.

Zacznijmy od dołu wspomnianej grafiki. Zwróć uwagę, że w ramach ścieżki zajavkowej poruszyliśmy już kwestię warstwy danych (na bardzo "prymitywnym" poziomie) podczas omawiania JDBC. Do komunikacji z bazą danych z poziomu Javy będziemy wracać jeszcze wielokrotnie. Wchodząc w świat Springa, zaczęliśmy rozgrzebywać temat warstwy logiki biznesowej, rozmawialiśmy tam o serwisach, które realizują właśnie jakąś logikę biznesową, czyli można powiedzieć, że temat ten również został

przez nas poruszony.

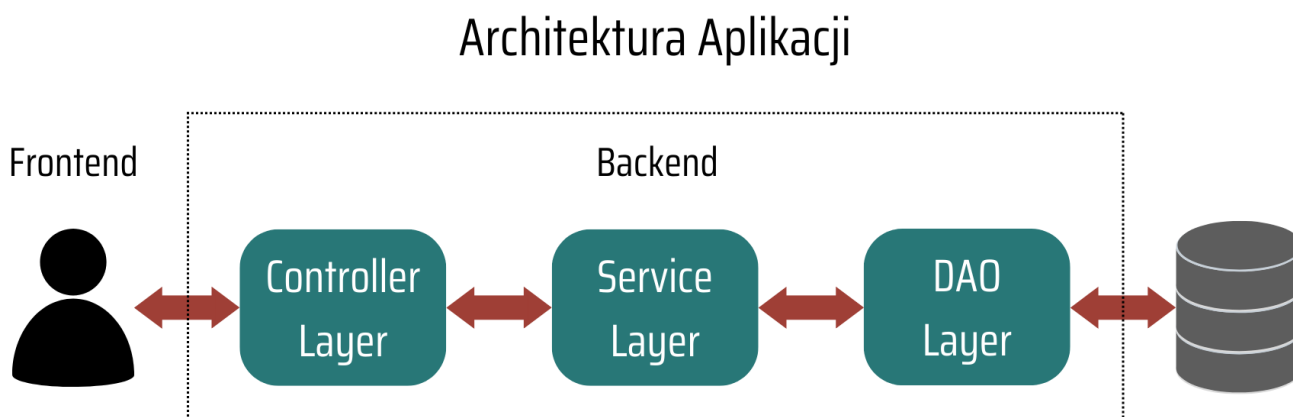
Nie rozmawialiśmy o warstwie prezentacji, czyli o interfejsie dla użytkownika. Wynika to z tego, że skupiamy się na backendowych zastosowaniach Javy, nie omawiamy zagadnień dotyczących warstwy wizualnej aplikacji webowych. Na ścieżce pojawi się jeden warsztat, gdzie stworzymy też warstwę wizualną, ale głównie skupiamy się na zagadnieniach backendowych.

Tematyka, którą będziemy poruszać w kolejnych warsztatach Zajavka będzie nas cały czas przybliżała do tego czym zajmuje się backend w aplikacjach webowych - pracą z danymi. Backend dane przetwarza, przelicza, przechowuje, usuwa i manipuluje. To wszystko jest realizowane dzięki warstwie **logiki biznesowej** i **warstwie danych**. Nie wiemy jeszcze na tym etapie, jak można skomunikować ze sobą frontend z backendem, ale do tego jeszcze przejdziemy.

Chcę tutaj natomiast zaznaczyć, że wcześniej uczyliśmy się podstawowych konstrukcji w Javie, a dopiero teraz zaczynamy powoli przechodzić do tego jak backend wygląda w rzeczywistości i jak pracuje z danymi. Podstawy Javy będą nam cały czas potrzebne, żeby móc te dane przetwarzać. Natomiast techniki, które teraz będziemy poznawać, będą nas przybliżać do tego, żeby móc przyjmować i wystawiać na frontend dane, które mają być wyświetlane na warstwie prezentacji, czyli klientowi końcowemu.

Dlatego w uproszczeniu można powiedzieć, że my na backendzie będziemy zajmować się warstwą dostępu do backendu, warstwą logiki biznesowej i warstwą danych, natomiast warstwą prezentacji będzie zajmował się frontend.

Wspomniane wcześniej warstwy można rozpisać bardziej szczegółowo, co zostało zrobione na poniższej grafice:



Obraz 3. Architektura Aplikacji

Można śmiało powiedzieć, że powyższa grafika przedstawia kwintesencję tego, czego się uczymy:

- Frontend - aplikacja, która uruchamia się użytkownikowi w przeglądarce internetowej,
- Controller Layer (*warstwa kontrolerów*) - ta część aplikacji będzie służyła do wystawiania backendu na świat i do tego, żeby frontend mógł z naszym backendem rozmawiać. Jeszcze tego nie umiemy, ale spokojnie, przejdziemy do tego,
- Service Layer (*warstwa usług, warstwa logiki biznesowej*) - ta część aplikacji będzie służyła do realizacji logiki biznesowej. Cały czas uczymy się technik, które pozwalają nam na realizację jakiejś logiki biznesowej,

- DAO Layer (*warstwa dostępu do danych*) - ta część aplikacji będzie odpowiedzialna za dostęp do szeroko rozumianych danych. Dane takie mogą być przetrzymywane w bazie danych lub nawet w pliku. W obrębie tego warsztatu oraz innych, gdzie poruszamy się w tematyce baz danych, tak naprawdę uczymy się cały czas o warstwie DAO oraz komunikacji z bazą danych.

Jeżeli chcesz przeczytać więcej na temat architektury warstwowej, możesz to zrobić np. [tutaj](#). Zagadnienia z tego artykułu, które nie są teraz zrozumiałe, będą wyjaśniane stopniowo na kolejnych etapach nauki.