# Aplikacje WWW. Wykład #5

## 1. ORM

### 1.1 Czym jest ORM?

Czy ORM to*:

- Operational Risk Management?
- Object-Relational Mapping?
- Opóźnienie Rozwoju Mowy?

---

Termin **ORM** oznacza mapowanie obiektowo-relacyjne (ang. Object-Relational Mapping), które służy do mapowania obiektów aplikacji (w różnych językach programowania) na relacyjne bazy danych. Każda instancja takiej klasy reprezentuje rekord w tabeli, a cecha kolumnę.

**Cechy mechanizmu ORM**

- **mapowanie obiekt-tabela**,
- **abstrakcja warstwy manipulacji danymi w bazie**, zapewnienie operacji CRUD,
- **'identity map'** - każdy wiersz w bazie reprezentowany jest przez unikalny obiekt w pamięci,
- **zarządzanie sesją** - mechanizm zapewniający poprawne wykonanie operacji w bazie, zapewniający spójność często poprzez wykorzystanie mechanizmu transakcji,
- **zarządzanie relacjami** - zapewnienie API pozwalającego reprezentować wszystkie dostępne typy relacji w postaci kodu w danym języku programowania,
- **jednostka pracy (ang. unity of work)** - zmiany konieczne do wykonania w bazie danych w sposób spójny są zbierany i opakowane w transakcję, w celu zapewnienia zarówno spójności jak i wydajności,
- **leniwe ładowanie (ang. lazy loading)** - pozwala na wczytywanie danych bez wszystkich rekordów potomnych z tabel relacyjnych wtedy kiedy nie jest to konieczne,
- **pamięć podręczna (ang. caching)** - zazwyczaj jest to cache oparty o mechanizm sesji, może występować również w postaci pamięci podręcznej współdzielonej między sesjami.

Za pierwszy ORM (chociaż wtedy się tak jeszcze tego mechanizmu nie nazywało) uznaje się ObjectiveSQL, stworzony pod koniec lat 80. XX wieku (Sega i Yosukawa). Systemy ORM znane nam dzisiaj zostały zapoczątkowane przez takie rozwiązania jak Hibernate oraz Entity Framework.

Podstawą wielu współczesnych ORM-ów są dwa wzorce projektowe: **Active Record** oraz wzorzec **Data Mapper**.

Active Record to wzorzec projektowy i framework używany w aplikacjach Ruby on Rails, który upraszcza interakcję z bazą danych dzięki Object Relational Mapping (ORM). Umożliwia reprezentowanie tabeli bazy danych jako klasy i jej wierszy jako obiektów, co pozwala na manipulowanie danymi za pomocą metod obiektowych zamiast pisania ręcznych zapytań SQL. Active Record działa z różnymi systemami baz danych, takimi jak MySQL, PostgreSQL, SQLite i MariaDB.

Kluczowe cechy Active Record:

- **Object Relational Mapping (ORM):** Mapuje obiekty Ruby na tabele w bazie danych, redukując potrzebę pisania kodu dostępu do bazy danych.
- **Reprezentacja tabeli jako klasy:** Każda tabela bazy danych ma odpowiadającą jej klasę, a wiersze tabeli są obiektami tej klasy.
- **Prosta interakcja z bazą danych:** Zamiast skomplikowanych zapytań SQL, można używać metod obiektowych do tworzenia, odczytywania, aktualizowania i usuwania danych (CRUD).
- **Kompatybilność z wieloma bazami danych:** Działa z popularnymi systemami baz danych, zachowując ten sam interfejs zapytań.
- **Zoptymalizowane zapytania:** Używa technik takich jak includes do ładowania powiązanych danych w jednym lub dwóch zapytaniach, zamiast wielu, co zapobiega problemowi N+1 zapytań.

O wzorcu **Data Mapper** możesz poczytać u źródła, czyli na stronie jego twórcy, **Martina Fowlera**: https://www.martinfowler.com/eaaCatalog/dataMapper.html

## 1.2 Wady i zalety stosowania ORM

**Zalety stosowania ORM**

- niezależny od silnika bazy danych,
- brak konieczności znajomości szczegółów języka SQL w celu wykorzystania bazy danych w aplikacji,
- większość systemów ORM posiada wbudowane zabezpieczenia np. przed atakami typu SQL Injection,
- łatwiejsze utrzymanie aplikacji bez konieczności aktualizacji poleceń SQL oraz zmiana silnika bazy danych na bardziej wydajny,
- wbudowane dodatkowe mechanizmy zapewniające spójność danych, np. mechanizm migracji w frameworku Django.

**Wady/wyzwania stosowania ORM**

- niektóre funkcje specyficzne dla danego silnika bazy danych mogą nie być dostępne bezpośrednio poprzez API danego rozwiązania typu ORM, również specyficzne typy danych po stronie silnika bazy,
- część zapytań może nie być wygenerowana w sposób najbardziej optymalny,
- niewłaściwe wykorzystanie 'lazy loading' lub 'eager loading' może prowadzić do problemów wydajnościowych.

## 1.3 Popularne rozwiązania ORM dla wybranych technologii backendowych

**Java**

W przypadku technologii Java należy najpierw wspomnieć o standardach, które zostały określone dla wielu elementów aplikacji wytwarzanych w tym języku. W szczególności chodzi tutaj o Java EE czyli Java Enterprise Edition. Pomijając większość jego elementów, tym, który jest podstawą rozwiązań typu ORM w tym języku jest standard JPA (Java Persistence API). Jedną z zaangażowanych osób w zdefiniowanie ram tego standardu jest Gavin King, twórca biblioteki Hibernate, która została wspomniania w dalszej części tego podpunktu.

> W przypadku Javy mamy od pewnego czasu dość skomplikowaną sytuację nazewniczą dla wielu standardów. W związku z przeniesieniem praw dla standardu JEE z firmy Oracle do Eclipse Foundation w roku 2017, ze względu na prawa autorskie do nazwy Java, które pozostały przy Oracle, postanowiono na zmianę członu nazw z Java ... na Jakarta .... I tak mamy teraz JPA jako Jakarta Persistence API, JEE -

> Jakarta Enterprise Edition itp. Więcej można doczytać m.in. tutaj: https://www.baeldung.com/java-enterprise-evolution. Reasumując. Oznaczają one te same technologie, ale w kolejnych wersjach.

Standard JPA określa sposób manipulacji danymi.

Poniżej przykład deklaracji encji, interfejsu repozytorium oraz usługi dostarczającej operacje CRUD dla tej encji w standardzie Java Persistence API.

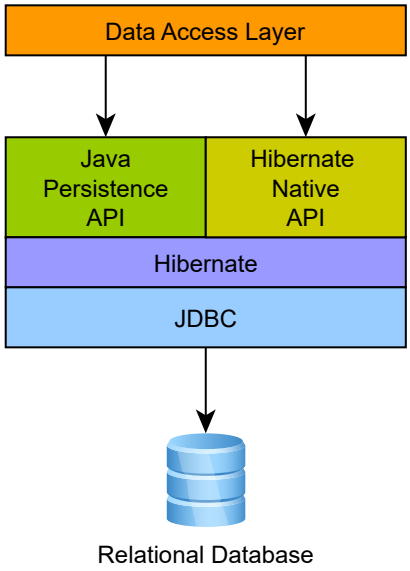
```
@Entity
public class Customer implements Serializable {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;
    private String email;

    // Getters and setters
}
```

```
public interface CustomerRepository extends JpaRepository<Customer, Long> {
}
```

```
@Service
public class CustomerService {
@Autowired
    private CustomerRepository customerRepository;

    public List<Customer> findAll() {
        return customerRepository.findAll();
    }

    public Customer save(Customer customer) {
        return customerRepository.save(customer);
    }

    public Optional<Customer> findById(Long id) {
        return customerRepository.findById(id);
    }

    public void deleteById(Long id) {
        customerRepository.deleteById(id);
    }
}
```

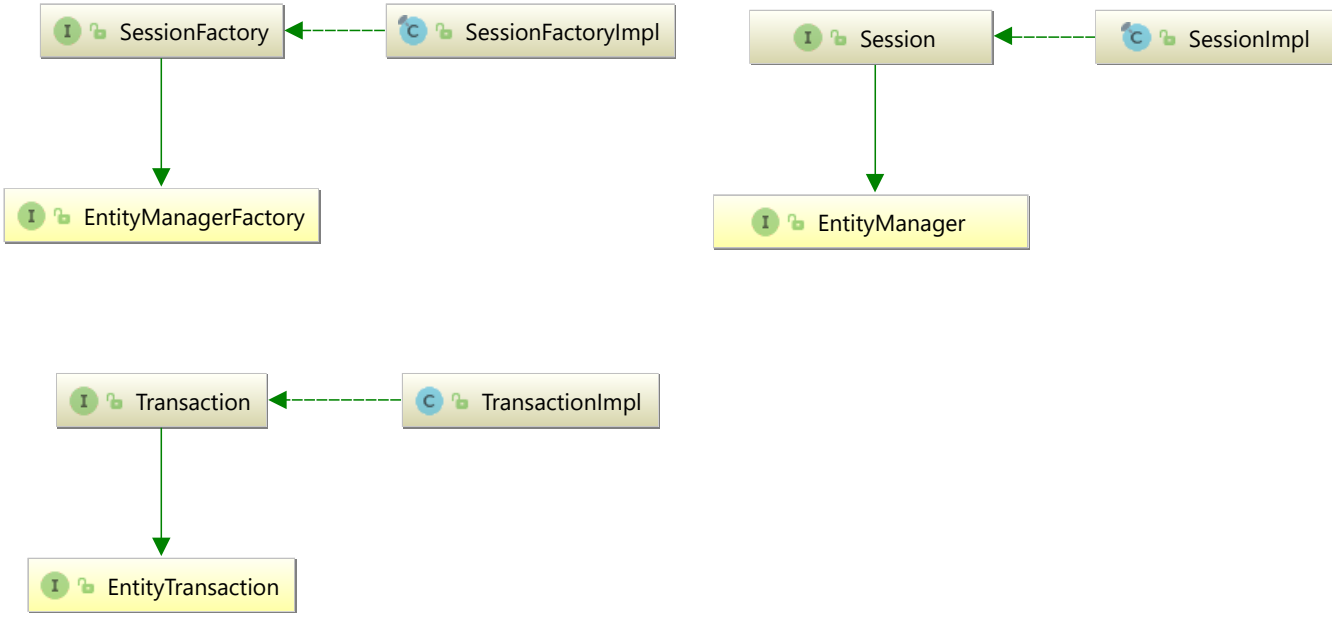
Wszystkie z powyższych przykładów pochodzą z: https://medium.com/@nagarjunmallesh/java-persistence-api-vs-jakarta-persistence-layer-a-comparative-study-with-code-implementation-4462b1abcfd1

Najbardziej popularnym, i uznawanym za najbardziej dojrzały, rozwiązaniem typu ORM dla aplikacji webowych wytwarzanych w technologii Java jest **Hibernate ORM**.




*źródło: hibernate.org*

Zgodnie ze specyfikacją Jakarta Persistence API, które Hibernate również implementuje wykorzystywane są poniższe elementy niezbędne do obsługi manipulacji danymi w bazie.




*źródło: hibernate.org*

Krótkie wyjaśnienie.

**SessionFactory (org.hibernate.SessionFactory)**

Jest to reprezentacja mapowanie obiektu domenowego (encji) na model bazy danych dla danego dialektu (silnika bazy danych). `EntityManagerFactory` jest ekwiwalentem `SessionFactory` ze standardu Jakarta Persistence

Tworzenie obiektu `SessionFactory` jest kosztowne i w ramach jednej aplikacji powinna istnieć tylko jedna instancja tej klasy. Służy ona do zarządzania pamięcią podręczną na poziomie warstwy dostepu do bazy danych, zarządzania pulą połączeń, transakcjami.

**Session (org.hibernate.Session)**

Obiekt, który reprezentuje wspomniany wcześniej 'Unit of work' i jest reprezentowany przez `EntityManager` w Jakarta Persistence. Działa jako fabryka instancji `org.hibernate.Transaction`

**Transaction (org.hibernate.Transaction)**

Obiekt, który wyznacza granice pojedynczej transakcji. Jest to ekwiwalent klasy `EntityTransaction` specyfikacji Jakarta Persistence.

Przykład deklaracji encji z wykorzystaniem biblioteki Hibernate.

```
@Entity(name = "Contact")
public static class Contact {

    @Id
    private Integer id;

    private Name name;

    private String notes;

    private URL website;

    private boolean starred;

    //Getters and setters are omitted for brevity
}

@Embeddable
public class Name {

    private String firstName;

    private String middleName;

    private String lastName;

    // getters and setters omitted
}
```

Pomijając część szczegółów implementacji stworzenie i zapisanie do bazy nowej instancji encji może zostać wykonane jak na piniższym przykładzie.

```
doInJPA(this::entityManagerFactory, entityManager -> {
    Name name = new Name();
```

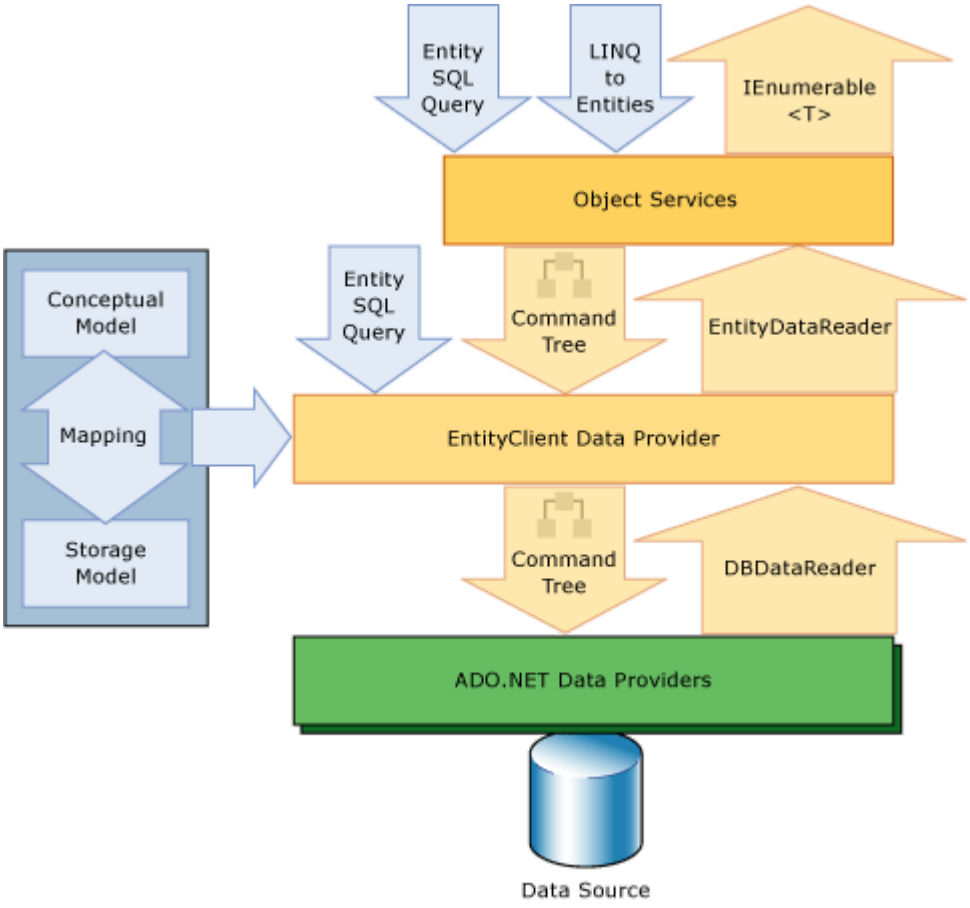
```
        name.setFirstName("John");
        name.setMiddleName("Robert");
        name.setLastName("Doe");

        Contact contact = new Contact();
        contact.setId(1);
        contact.setName(name);
        contact.setNotes("Example contact");
        contact.setWebsite(new URL("https://example.com"));
        contact.setStarred(false);
        entityManager.persist(concatc);
    });
```

Oprócz Hibernate dostępne są również inne rozwiązania typu ORM dla języka Java, ale nie są one tak popularne. Są to:

- Oracle TopLink (https://www.oracle.com/middleware/technologies/top-link.html)
- ObjectDB (https://www.objectdb.com/)
- EclipseLink (https://eclipse.dev/eclipselink/)

**.NET (C#)**

Najpopularniejszym rozwiązaniem typu ORM dla platformy .NET jest **Entity Framework** (patrz: https://learn.microsoft.com/pl-pl/ef/).




*źródło: https://learn.microsoft.com/pl-pl/dotnet/framework/data/adonet/ef/overview*

**Zalety Entity Framework:**

- Wysoki poziom abstrakcji
- Wsparcie dla migracji bazy danych
- Bogaty zestaw narzędzi LINQ do zapytań
- Automatyczne śledzenie zmian obiektów
- Wsparcie dla transakcji

Jest to obecnie najpopularniejsze rozwiązanie ORM dla platformy .NET, aktywnie rozwijane przez Microsoft i społeczność open source.

**Podejścia do modelowania**

- **Code First** - tworzenie modelu w kodzie C#, a następnie generowanie bazy
- **Database First** - generowanie klas na podstawie istniejącej bazy danych
- **Model First** - tworzenie modelu wizualnie i generowanie bazy oraz klas

Przykładowa implementacja encji oraz operacji bazodanowych z wykorzystaniem Entity Framework.

```
public class Contact
{
    public int Id { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string Email { get; set; }
}

// filepath: Data/ApplicationDbContext.cs
public class ApplicationDbContext : DbContext
{
    public DbSet<Contact> Contacts { get; set; }

    protected override void OnConfiguring(DbContextOptionsBuilder options)
        => options.UseSqlServer("Connection_String_Here");
}
```

```
// Dodawanie rekordu
using (var context = new ApplicationDbContext())
{
    var contact = new Contact
    {
        FirstName = "John",
        LastName = "Doe",
        Email = "john@example.com"
    };
    context.Contacts.Add(contact);
    context.SaveChanges();
}
```

```
// Pobieranie danych
var contacts = context.Contacts.Where(c => c.LastName == "Doe").ToList();
```

Więcej przykładów oraz sposobów deklaracji klas encji (np. z użyciem adnotacji) można znaleźć w oficjalnej dokumentacji: https://learn.microsoft.com/pl-pl/ef/core/modeling/

Entity Framework jest niewątpliwie najbardziej popularnym rozwiązaniem typu ORM dla .NET, jednak nie jedynym. Oto kilka wybranych dostępnych dla tej platformy.

**Dapper**

Lekki framework zapewniający bezpośrednią kontrolę nad SQL, nierzadko uznawany za najlepszy ORM dla języka C#.

Najważniejsze cechy:

- skoncentrowany na wydajności,
- minimalny narzut,
- kontrola nad surowym SQL-em,
- wsparcie dla asynchroniczności.

**NHibernate**

Bardzo dojrzały (jak sam Hibernate) framework ORM klasy enterprise. Oferuje zaawansowane opcje oraz wsparcie dla wielu typów bazy danych. Minusem może być dość wysoki stopień trudności w nauce i braki w dokumentacji.

Najważniejsze cechy:

- zaawansowane możliwości mapowania,
- bardzo duża liczba wspieranych silników baz danych,
- mechanizm pamięci podręcznej,
- implementacja mechanizmu 'unit of work',
- dojrzały ekosystem.

**Marten**

Rozwiązanie ORM skoncentrowane głównie dla silnika PostgreSQL, chociaż posiada również wsparcie dla przechowywania dokumentów w stylu NoSQL.

Najważniejsze cechy:

- przechowywanie dokumentów JSON (wykorzystuje wbudowane możliwości PostgreSQL),
- wspiera model relacyjny, model klucz-wartość oraz przechowywanie dokumentów w jednej bazie danych,
- zoptymalizowane zapytania, testy pokazują przewagę wydajności nad ORM LINQ w tym aspekcie.

**LinqConnect**

LinqConnect by Devart to ORM rozszerzający możliwości LINQ. Wspiera silniki takie jak SQL Server, Oracle, MySQL, PostgreSQL oraz SQLite.

Najważniejsze cechy:

- jest to dostawca LINQ (ang. LINQ provider), co zapewnia lepszą wydajność oraz wsparcie dla LINQ bez warstw pośredniczących między LINQ a SQL,
- wspiera modelowanie code-first, database-first, model-first oraz kombinacje database-first i model-first,
- zintegrowany z innymi technologiami firmy Microsoft takimi jak: Windows Forms, ASP.NET, WPF czy WCF RIA,
- wspiera lazy-loading oraz eager-loading na poziomie pojedynczego zapytania.

Warto również zwrócić uwagę na:

- **RepoDB**
- **OrmLite**
- **PetaPOCO**
- **XPO**

**PHP**

W przypadku języka PHP wybór systemu ORM jest nieco trudniejszy ze względu na brak takich standardów jak chociażby dla języka Java (JPA) czy .NET. Rozwiązanie ORM jest często już częścią konkretnego frameworka PHP. Istnieje jednak kilka autonomicznych bibliotek ORM.

Kilka wybranych, najbardziej popularnych rozwiązań ORM (również wbudowanych w ramach frameworka) dla języka PHP:

---

## **Doctrine** (https://www.doctrine-project.org/projects/orm.html)

To rozbudowana biblioteka zapewniająca szerokie wsparcie dla silników baz danych oraz rozbudowane możliwości mapowania. Wymaga jednak dość dużej ilości kodu do stworzenia.

Poniżej przykład, dla zobrazowanie różnic i podobieństw dla wybranych języków programowania, implementacji obsługi ORM poprzez Doctrine.

(Wszystkie przykłady pochodzą z oficjalnej dokumentacji projektu: https://www.doctrine-project.org/projects/doctrine-orm/en/3.5/tutorials/getting-started.html)

Deklaracja menadżera encji dla projektu.

```
<?php
// bootstrap.php
use Doctrine\DBAL\DriverManager;
use Doctrine\ORM\EntityManager;
use Doctrine\ORM\ORMSetup;
require_once "vendor/autoload.php";
// Create a simple "default" Doctrine ORM configuration for Attributes
$config = ORMSetup::createAttributeMetadataConfig( // on PHP < 8.4, use
ORMSetup::createAttributeMetadataConfiguration()
    paths: [__DIR__ . '/src'],
    isDevMode: true,
);
```

```
// or if you prefer XML
// $config = ORMSetup::createXMLMetadataConfig( // on PHP < 8.4, use
ORMSetup::createXMLMetadataConfiguration()
//    paths: [__DIR__ . '/config/xml'],
//    isDevMode: true,
//);
// configuring the database connection
$connection = DriverManager::getConnection([
    'driver' => 'pdo_sqlite',
    'path' => __DIR__ . '/db.sqlite',
], $config);
// obtaining the entity manager
$entityManager = new EntityManager($connection, $config);
```

Implementacja encji zależy od przyjętej strategii. Możliwe jest zdefiniowanie tzw. `anemic entities` oraz `rich entities`. Pierwsze z nich to klasy definiujące pola oraz zapewniające odpowiednie gettery oraz settery. Te drugie to encje implementowane w stylu DDD (Data Driven Development), które oprócz pól posiadają metody, które bardziej odpowiadają akcjom reprezentującym logikę biznesową, którą aplikacja ma realizować. Poniżej dwa prykłady (również z oficjalnej dokumentacji).

Encja typu `anemic`.

```
<?php
class User
{
    private $username;
    private $passwordHash;
    private $bans;
    public function getUsername(): string
    {
        return $this->username;
    }
    public function setUsername(string $username): void
    {
        $this->username = $username;
    }
    public function getPasswordHash(): string
    {
        return $this->passwordHash;
    }
    public function setPasswordHash(string $passwordHash): void
    {
        $this->passwordHash = $passwordHash;
    }
    public function getBans(): array
    {
        return $this->bans;
    }
    public function addBan(Ban $ban): void
    {
        $this->bans[] = $ban;
```

```
    }
}
```

Encja typu `rich` (DDD).

```
<?php
class User
{
    private $banned;
    private $username;
    private $passwordHash;
    private $bans;
    public function toNickname(): string
    {
        return $this->username;
    }
    public function authenticate(string $password, callable $checkHash): bool
    {
        return $checkHash($password, $this->passwordHash) && ! $this-
>hasActiveBans();
    }
    public function changePassword(string $password, callable $hash): void
    {
        $this->passwordHash = $hash($password);
    }
    public function ban(\DateInterval $duration): void
    {
        assert($duration->invert !== 1);
        $this->bans[] = new Ban($this);
    }
}
```

Do zarządzania samymi encjami wykorzystywane są natomiast obiekty typu DTO.

```
<?php
class User
{
    public function updateFromProfile(ProfileEditingDTO $profileFormDTO): void
    {
        // kod aktualizujący encję
    }
    public static function createFromRegistration(UserRegistrationDTO
$registrationDTO): self
    {
        // kod tworzący encję np. z formularza rejestracji
    }
}
```

Możliwe jest również wykorzystanie metajęzyka do definiowania encji poprzez Doctrine.

Przykład:

```php
<?php
// src/Product.php
use Doctrine\ORM\Mapping as ORM;
#[ORM\Entity]
#[ORM\Table(name: 'products')]
class Product
{
    #[ORM\Id]
    #[ORM\Column(type: 'integer')]
    #[ORM\GeneratedValue]
    private int|null $id = null;
    #[ORM\Column(type: 'string')]
    private string $name;
    // .. (other code)
}
```

I ostatecznie wykorzystanie Entity Managera do wykonania operacji na encji i bazie danych.

```php
<?php
// create_product.php <name>
require_once "bootstrap.php";
$newProductName = $argv[1];
$product = new Product();
$product->setName($newProductName);
$entityManager->persist($product);
$entityManager->flush();
echo "Created Product with ID " . $product->getId() . "\n";
```

---

## **RedBeanPHP** (https://redbeanphp.com/index.php)

Biblioteka, która zawiera się w jednym pliku i nie wymaga definicji encji, gdyż te są generowane w locie, na podstawie schematu bazy danych. Ma to swoje zalety (szybkość uruchomienia projektu, ilość niezbędnego do napisania kodu) oraz wady (polegamy na poprawnie zdefiniowanym schemacie bazy, definiuje konwencje nazw, których musimy się trzymać aby automatyczne mapowanie działało poprawnie).

Przykład z oficjalnej strony, który wystarczy do bardzo prostej operacji:

- importu biblioteki,
- wykonania połączenia (do metody `setup` należałoby podać konfigurację połączenia do bazy),
- deklaracji i stworzenia nowej tabeli o nazwie `post` z kolumną `text` jeżeli nie istnieje,
- wstawienie rekordu do tabeli i zwrócenie wygenerowanego `id`,
- wczytanie utworzonego rekordu dla danego id,
- usunięcie tego rekordu.

```php
require 'rb.php';
    R::setup();

    //for version 5.3 and higher
    //optional but recommended
    R::useFeatureSet( 'novice/latest' );

    $post = R::dispense( 'post' );
    $post->text = 'Hello World';

    //create or update
    $id = R::store( $post );

    //retrieve
    $post = R::load( 'post', $id );

    //delete
    R::trash( $post );
```

---

## **Propel** (https://propelorm.org/)

Główne cechy:

- wydajność,
- wsparcie dla MySQL, SQLite oraz PostgreSQL,
- inżynieria wsteczna,
- generowanie metod dostępowych dla wszystkich kolumn oraz relacji,
- implementacja unit of work,
- wysoce konfigurowalny,
- dobra dokumentacja.

---

## **Eloquent (komponent frameworka Laravel)** (https://laravel.com/docs/12.x/eloquent)

**Zalety:**

- bardziej czytelny kod w porównaniu do QueryBuilder,
- wsparcie dla soft-delete - zamiast usuwać dane, zostanie ustawiona flaga z datą usunięcia, a dane faktycznie pozostaną w bazie, nie będą widoczne w standardowych zapytaniach,
- wsparcie dla zdarzeń modeli - można zdefiniować operacje, które wykonają się przed lub po wskazanej operacji, jak triggery w bazie, ale tutaj osadzone w logice aplikacji.

**Wady:**

- wolniejszy od QueryBuilder (większy narzut),
- nie tak elastyczny jak QueryBuilder,

- może być trudny w debuggingu.

---

## Cake PHP (framework) z wbudowanym modułem ORM (https://book.cakephp.org/5/en/orm.html)

Wbudowanu moduł ORM zapewnia abstrakcję warstwy dostępu do danych w postaci repozytoriów, które zapewniają operacje manipulacji danymi, oraz encji, które definiują właściwości tabeli/rekordu.

**JavaScript**

Dla języka JavaScript możemy znaleźć takie rozwiązania ORM jak:

- Knex.js: SQL Query Builder
- Sequelize
- Bookshelf
- Waterline
- Objection.js
- Mongoose
- Typegoose
- TypeORM
- MikroORM
- Prisma

Z racji dość ograniczonego wykorzystania technik frontendowych w ramach realizowanego przedmiotu postanowiłem bardzo krótko przedstawić tylko jeden z nich, a mianowicie Sequelize (https://sequelize.org/).

Ta biblioteka może być wykorzystana dla języka TypeScript (typowana JavaScript) oraz biblioteki Node.js współpracująca z takimi bazami jak Oracle, PostgreSQL, MySQL, MariaDB, SQLite, SQL Server. Oferuje transakcje, relacje, lazy loading, eager loading oraz replikację.

Powielony przykład z głównej strony biblioteki:

```
import { Sequelize, DataTypes } from 'sequelize';

// inicjalizacja
const sequelize = new Sequelize('sqlite::memory:');

// deklaracja encji
const User = sequelize.define('User', {
  username: DataTypes.STRING,
  birthday: DataTypes.DATE,
});

// stworzenie instancji encji
const jane = await User.create({
  username: 'janedoe',
  birthday: new Date(1980, 6, 20),
});
```

```
// pobranie instancji encji
const users = await User.findAll();
```

**Python**

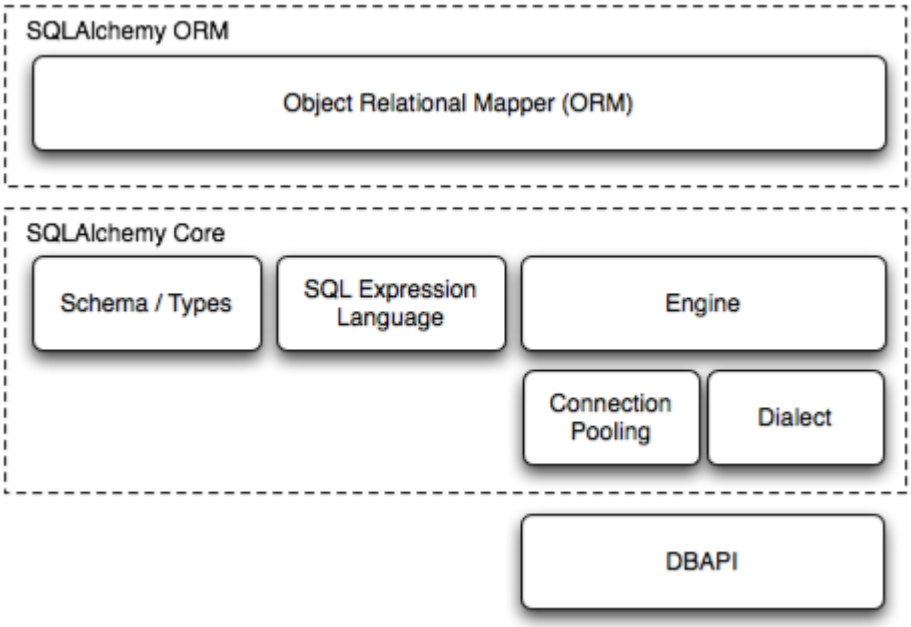
W przypadku Pythona najbardziej popularne rozwiązania typu ORM to:

- SQLAlchemy (https://www.sqlalchemy.org/)
- Django ORM (https://docs.djangoproject.com/en/5.2/topics/db/)
- Peewee (https://peewee.readthedocs.io/en/latest/)
- Pony ORM (https://ponyorm.org/)

## SQLAlchemy

SQLAlchemy jest biblioteką oferującą mechanizm mapowania obiektowo relacyjnego (ORM) oraz dodatkowe narzędzia pozwalające na zaawansowaną pracę z bazami danych poprzez kod języka Python. Jest też często wykorzytywana jako moduł ORM w frameworkach, które nie oferują własnego modułu ORM np. Flask.

Poniżej obrazowo przedstawione są komponenty tego frameworka.




*źródło: https://docs.sqlalchemy.org/en/20/intro.html#overview*

Poniżej przykład kilku operacji, począwszy od połączenia do bazy danych (lub jej stworzenia w przypadku poniższego silnika SQLite), stworzenia tabel, wstawienia danych, pobrania ich poprzez stworzenie encji ORM i kilku operacji z ich udziałem.

Kod częściowo pochodzi z oficjalnej dokuemntacji SQLAlchemy, a częściowo został stworzony przez prowadzącego.

```
from sqlalchemy import create_engine
from sqlalchemy import text
```

```
# inicjalizacja silnika bazy danych dla wybranego dialektu
engine = create_engine("sqlite:///.\\db.sqlite", echo=True)

# dummy zapytanie, nawet nie do bazy
with engine.connect() as conn:
    result = conn.execute(text("select 'hello world'"))
    print(result.all())

# przykład zapytania w postaci surowego SQL-a
with engine.connect() as conn:
    conn.execute(text("""CREATE TABLE IF NOT EXISTS student(
            indeks INTEGER PRIMARY KEY AUTOINCREMENT,
            imie varchar(50),
            nazwisko varchar(100))"""))
    conn.execute(
        text("INSERT INTO student (imie, nazwisko) VALUES ( :imie, :nazwisko)"),
        [{"imie": "Adam", "nazwisko": "Mickiewicz"},
                    {"imie": "Jan", "nazwisko": "Kowalski"}],
         )

    conn.commit()
    result = conn.execute(text("SELECT * FROM student"))
    print(result.all())

    # lub
    result = conn.execute(text("SELECT * FROM student"))
    for row in result:
        print(row, sep=',')

    # można również
    result = conn.execute(text("SELECT * FROM student"))
    for row in result.mappings():
        print(row)


# teraz wykorzystanie modułu ORM SQLAlchemy

from sqlalchemy import MetaData, insert, select
from sqlalchemy import Table, Column, Integer, String, ForeignKey

metadata_obj = MetaData()

# deklarujemy pierwszą tabelę
user_table = Table(
    "user_account",
    metadata_obj,
    Column("id", Integer, primary_key=True),
    Column("name", String(30)),
    Column("fullname", String),
)

# deklarujemy drugą tabelę

address_table = Table(
```

```
    "address",
    metadata_obj,
    Column("id", Integer, primary_key=True),
    Column("user_id", ForeignKey("user_account.id"), nullable=False),
    Column("email_address", String, nullable=False),
)

# wykonanie operacji tworzenia schematu w wybranym silniku bazy danych
metadata_obj.create_all(engine)
# wyświetlamy listę dostępnych tabel (dla tego zbioru metadanych)
metadata_obj.tables.keys()

# wstawienie danych do takiej tabeli

stmt = insert(user_table).values(name="spongebob", fullname="Spongebob
Squarepants")
print(stmt)
compiled = stmt.compile()
with engine.connect() as conn:
    result = conn.execute(stmt)
    conn.commit()

print(result.inserted_primary_key)

# pobieramy dane

# same polecenie
print(select(user_table))

# wykonanie
with engine.connect() as conn:
    result = conn.execute(select(user_table).compile())
    conn.commit()
    print(result.all())
```

Możliwa jest również deklaracja encjji w sposób, który jest bardziej zbliżony do przykładów z przedstawionych wcześniej frameworków oraz Django ORM.

```
from typing import List
from typing import Optional
from sqlalchemy.orm import Mapped
from sqlalchemy.orm import mapped_column
from sqlalchemy.orm import relationship

class User(Base):
    __tablename__ = "user_account"
    id: Mapped[int] = mapped_column(primary_key=True)
    name: Mapped[str] = mapped_column(String(30))
    fullname: Mapped[Optional[str]]
    addresses: Mapped[List["Address"]] = relationship(back_populates="user")
```

```
    def __repr__(self) -> str:
        return f"User(id={self.id!r}, name={self.name!r}, fullname=
{self.fullname!r})"

class Address(Base):
    __tablename__ = "address"
    id: Mapped[int] = mapped_column(primary_key=True)
    email_address: Mapped[str]
    user_id = mapped_column(ForeignKey("user_account.id"))
    user: Mapped[User] = relationship(back_populates="addresses")
    def __repr__(self) -> str:
        return f"Address(id={self.id!r}, email_address={self.email_address!r})"
```

*źródło: https://docs.sqlalchemy.org/en/20/tutorial/metadata.html*

## Django ORM

Moduł ORM wbudowany w framework Django zapewnia możliwość definiowania modeli (encji) oraz operacji na bazie danych. Encje deklarujemy poprzez rozszerzanie bazowej klasy `django.db.models.Model`, która zapewnia również dostęp do metod CRUD poprzez klasę `django.db.models.manager.Manager`. Zapytania wybierające wykonywane za pośrednictwem menedżera zwracają obiekt typu `QuerySet`, który jest koklekcją encji.

Każdy model zawiera przynajmniej jeden domyślny menedżer, który dostępne jest poprzez własność `objects` danej encji, np. `Post.objects.all()` wywołuje akcję `all()` domyślnego menedżera i zwraca obiekt typu `QuerySet` zawierającego wszystkie krotki modelu `Post` z bazy danych (kolekcja może też być pusta).

Obiekty typu `QuerySet` są domyślnie leniwe, co oznacza, że ich utworzenie nie wywołuje jeszcze żadnej akcji na bazie danych. Dopiero, kiedy faktycznie potrzebne są dane określone przez zapytanie, jest ono przygotowywane, optymalizowane oraz wykonywane.

Poniższy przykład:

```
>>> q = Post.objects.filter(text__contains="Kebab")
>>> q = q.filter(created_at__lte=datetime.date.today())
>>> print(q)
```

utworzy obiekt typu `QuerySet`, ale żadna operacja na bazie danych nie zostanie wykonana do linii, w której zostanie wywołana funkcja `print()` na tym obiekcie. Całość zostanie wykonana jako jedno zapytanie do bazy.

Należy tutaj wspomnieć również o pewnym wyjątku, gdyż wywołanie `Post.objects.get(1)` zwróci instancję modelu, a nie obiekt typu `QuerySet`.

Obiekty `QuerySet` wykorzystują pamięć podręczną, jednak nie w każdej sytuacji. Kiedy wyniki zapytania są pobierane po raz pierwszy trafiają one do pamięci podręcznej, ale jeżeli nie zapisujemy wyników zapytania w postaci zmiennej, możemy generować dużo więcej zapytań do bazy niż jest to potrzebne.

Przykłady pochodzą głównie z oficjalnej dokumentacji:
https://docs.djangoproject.com/en/5.2/topics/db/queries/)

```
# dwa oddzielne zapytania do bazy
print([e.headline for e in Entry.objects.all()])
print([e.pub_date for e in Entry.objects.all()])

# jedno zapytanie, kolejne z wykorzystaniem pamięci podręcznej
queryset = Entry.objects.all()
print([p.headline for p in queryset])  # Evaluate the query set.
print([p.pub_date for p in queryset])  # Reuse the cache from the evaluation.
```

Nie wszystkie zapytania będą jednak przechowywane w pamięci podręcznej. Poniżej przykład, gdzie przy ograniczeniu wyników zapytania, np. poprzez slice (który zostanie wykonany z użyciem SQL-owego wyrażenia `LIMIT`) spowoduje ponowne odpytanie bazy danych. Taka sytuacja ma miejsce, jeżeli nie wszystkie elementy `QuerySet` nie zostały faktycznie pobrane i przetworzone.

```
queryset = Entry.objects.all()
print(queryset[5])  # Queries the database
print(queryset[5])  # Queries the database again
```

A tu już zostanie odpytany cache.

```
queryset = Entry.objects.all()
[entry for entry in queryset]  # Queries the database
print(queryset[5])  # Uses cache
print(queryset[5])  # Uses cache
```

## Django ORM oferuje również:

- **wsparcie dla zapytań asynchronicznych** - zostały zapewnione odpowiedniki metod synchronicznych takich jak `get` czy `delete` w postaci metod `aget` czy `adelete` (więcej tutaj: https://docs.djangoproject.com/en/5.2/topics/db/queries/#asynchronous-queries oraz https://docs.djangoproject.com/en/5.2/topics/async/)
- **możliwość budowania bardziej skomplikowanych zapytań** poprzez wykorzystanie dedykowanej klasy `Q` pozwalającej na określanie warunku i następnie użycie operatorów logicznych dla tych warunków. Przykład:

```
from django.db.models import Q

# deklaracja warunków z użyciem Q może być łączona z dotychczasowymi
# warunkami, ale musi zostać przekazana jako wcześniejsze argumenty
# możemy je łączyć również z exclude, filter
Post.objects.get(
    Q(text__startswith="Who") | Q(text__startswith="What"), created_by=1
)
```

więcej: https://docs.djangoproject.com/en/5.2/ref/models/querysets/#django.db.models.Q

- **wsparcie dla wielu baz w jednym projekcie** - zobacz przykłady wykorzystania tu: https://docs.djangoproject.com/en/5.2/topics/db/multi-db/
- **wsparcie dla zaawansowanego wyszukiwania w tekście,** np. z wykorzystaniem wbudowanych możliwości silnika bazy PostgreSQL (stop words i inne). Zobacz: https://docs.djangoproject.com/en/5.2/topics/db/search/
- **wykonania surowego zapytania SQL jeżeli jest to konieczne**: https://docs.djangoproject.com/en/5.2/topics/db/sql/
- **wsparcie dla transakcji** - Django ORM domyślnie działa w trybie `autocommit` i oczywiście wspiera obsługę transakcji na poziomie silnika bazy danych. Jednak ta funkcjonalność idzie trochę dalej oferując możliwość opakowania widoku (czyli akcji wykonywanej dla danego żądania, tak w dużym uproszczeniu) w transakcję. Można to zrobić ustawiając globalnie parametr `ATOMIC_REQUESTS = True` dla każdej bazy danych, albo adnotując każde żądanie zgodnie z przykładami zawartymi w dokumentacji: https://docs.djangoproject.com/en/5.2/topics/db/transactions/
- **obsługę sygnałów**, które można wysyłać lub nasłuchiwać np. dla operacji usunięcia wybranego modelu i wielu innych domyślnych sygnałów. Więcej: https://docs.djangoproject.com/en/5.2/ref/signals/
- **możliwość wywołania popularnych funkcji bazodanowych** poprzez ich wrappery zawarte w pakiecie `django.db.models.functions`

## 1.4 Ciekawostki i zadania

1. * Wszystkie odpowiedzi są prawidłowe.
2. **Zadanie do wykonania**
   Znajdź w kodzie źródłowym frameworka Django (zainstalowanego w Twoim projekcie z laboratorium) funkcję odpowiedzialną za porównanie instancji dwóch modeli Django ORM (encji). Pierwsza osoba, która zgłosi się i wskaże odpowiedni fragment w kodzie źródłowym otrzyma `+` do oceny z projektu.
3. Spróbuj uruchomić przykład spod adresu:
   https://docs.sqlalchemy.org/en/20/orm/examples.html#module-examples.space_invaders