

# Warsztaty II

# PHP

v 1.4

# Obiektowa praca z bazą danych

# Cel warsztatów

Celem warsztatów jest poznanie sposobu integracji baz danych z programowaniem obiektowym. Można to robić na wiele różnych sposobów.

Najpopularniejsze wzorce projektowe, które rozwiązują ten problem to:

- Row Data Gateway,
- Active Record,
- Data Mapper,

Podczas warsztatów wasze klasy będą implementować wzorzec **Active Record** dlatego ten wzorzec będzie na początku dokładnie opisany.

# Tablica w bazie danych a klasa

Kluczem do zrozumienia tego jak współpracujemy z bazą danych jest zrozumienie zależności pomiędzy naszymi klasami a tablicami w bazie danych.

W większości podejść mamy następujące założenia:

- Na każdą tablicę w naszej bazie danych przypada jedna klasa która ją reprezentuje.
- Klasa ma taką samą nazwę jak tablica i atrybuty odpowiadające kolumnom tablicy.
- Każdy obiekt tej klasy jest reprezentacją jednego rzędu z tablicy.
- Obiekt może być w dwóch stanach: **zsynchronizowany** i **niezsynchronizowany**.

# Synchronizacja obiektu

Obiekt jest **zsynchronizowany** z bazą danych jeżeli dane trzymane w jego atrybutach **są takie same** jak dane w odpowiadającym mu rzędzie.

Synchronizacja następuje w przypadku:

- Wczytania rzędu z bazy do obiektu,
- Zapisanie obiektu do bazy danych.

Obiekt jest **niezsynchronizowany** z bazą danych jeżeli dane trzymane w jego atrybutach **nie są takie same** jak dane w odpowiadającym mu rzędzie, lub **nie posiada** rzędu, który mu odpowiada.

Obiekt jest rozsynchronizowany w przypadku:

- Stworzenia nowego obiektu (nie ma rzędu mu odpowiadającego),
- Usunięcia rzędu z bazy danych,
- Zmiany któregoś atrybutu po ostatniej synchronizacji.

**Jeżeli któryś z naszych obiektów będzie niezsynchronizowany z bazą danych pod koniec działania naszego programu to zmiany w nim zawarte nie zostaną zapisane do bazy danych!**

# Active Record

# Active Record

**Active Record** jest jednym z prostszych wzorców, który służy nam do komunikacji z bazą danych. Dla każdej tablicy, którą używamy w naszym programie implementujemy osobną klasę. Klasa posiada atrybuty odpowiadające kolumnom naszej tabeli. Dodatkowo klasa implementuje metody służące nastawianiu i pobieraniu wszystkich swoich atrybutów (getery i setery) oraz metody służące do komunikacji z bazą danych.

Zazwyczaj są to:

- **update()** – zapisuje obiekt do tabeli (jako zmiany wcześniej istniejącego rzędu tej tabeli)
- **save()** – zapisuje obiekt do tabeli (jako nowy rząd)
- **delete()** – usuwa obiekt z tabeli (czyli usuwa rząd o id takim samym jak obiekt).

Obiekt tej klasy reprezentować będzie nam jeden rząd w naszej tabeli. Nasz obiekt służy nam zarówno do trzymania danych i komunikacji z bazą danych jak i implementacji wszelkiej logiki potrzebnej dalej w programie.



# Active Record

**Active Record** dodatkowo implementuje zazwyczaj gamę statycznych metod które mają nam pomóc w wyszukiwaniu lub załadowaniu większej ilości danych.

Zazwyczaj są to:

- **loadAll()** – wczytuje wszystkie rzędy z tabeli na podstawie każdego rzędu tworzy nowy obiekt, następnie zwraca tablice z wszystkimi stworzonymi obiektami,
- **loadById(id)** – wczytuje jeden rząd z tabeli (o podanym id) i zwraca obiekt który jest reprezentacją tego rzędu,
- **deleteAll()** – usuwa wszystkie dane z tablicy,

Bardzo często metoda **loadAll** i **loadById(id)** występują w różnych wariantach (np. wyszukuje dane na podstawie którejś kolumny).



# Active Record - przykład

Jako przykład możemy pokazać przechowywanie użytkowników w bazie danych.

Chcemy żeby nasza klasa przechowywała następujące dane:

- Id użytkownika (nastawiane przez bazę danych – zazwyczaj auto\_increment),
- Imię użytkownika,
- Hasło (zahashowane),
- Email użytkownika (unikalny w naszym systemie).

Chcemy, żeby nasza klasa miała możliwość:

- Zapisu nowego użytkownika do bazy danych,
- Edycji istniejącego użytkownika,
- Usunięcia istniejącego użytkownika,
- Wczytania użytkownika po jego ID,
- Wczytania użytkownika po jego emailu (potrzebne do logowania),
- Wczytania wszystkich użytkowników,
- Zmiany wszystkich jego atrybutów.

# Tablica Users

W bazie danych będziemy mieli tabelkę **Users**. Tabela będzie opisana w następujący sposób:

Field	Type	Null	Key	Default	Extra
id	int(11)	NO	PRI	NULL	auto_increment
email	varchar(255)	NO	UNI	NULL	
username	varchar(255)	NO		NULL	
hashed_password	varchar(60)	NO		NULL	

Kolumna mail jest unikalna  
(zakładamy że nie może być  
dwóch użytkowników z takim  
samym emailem)

Nasz klucz główny

# Klasa User

W naszym kodzie stworzymy klasę **User** (pamiętaj żeby wszystkie klasy trzymać w osobnym folderze np. **/src**).

Stwórz plik **User.php** (pamiętaj żeby plik nazywać tak samo jak klasę która w nim się znajduje – będzie to ważne z punktu widzenia Autolodera – więcej o nim dowiesz się podczas zajęć z zaawansowanego PHP).

Nasza klasa będzie miała następujące atrybuty (wszystkie prywatne):

- id,
- username,
- hashedPassword,
- email.

```
class User {  
    private $id;  
    private $username;  
    private $hashedPassword;  
    private $email;  
}
```

# User - id

Bardzo ważnym atrybutem naszej klasy jest atrybut **id**.

W przypadku w którym obiekt nie ma jeszcze rzędu w bazie danych (np. w przypadku kiedy stworzyliśmy nowy obiekt ale jeszcze nie zapisaliśmy go do bazy danych, albo usuwamy z bazy danych wcześniej wczytany obiekt) wartość naszego **id** będzie wynosiła **-1**. Wybieramy tą liczbę ponieważ SQL nigdy nie nada takiego klucza głównego.

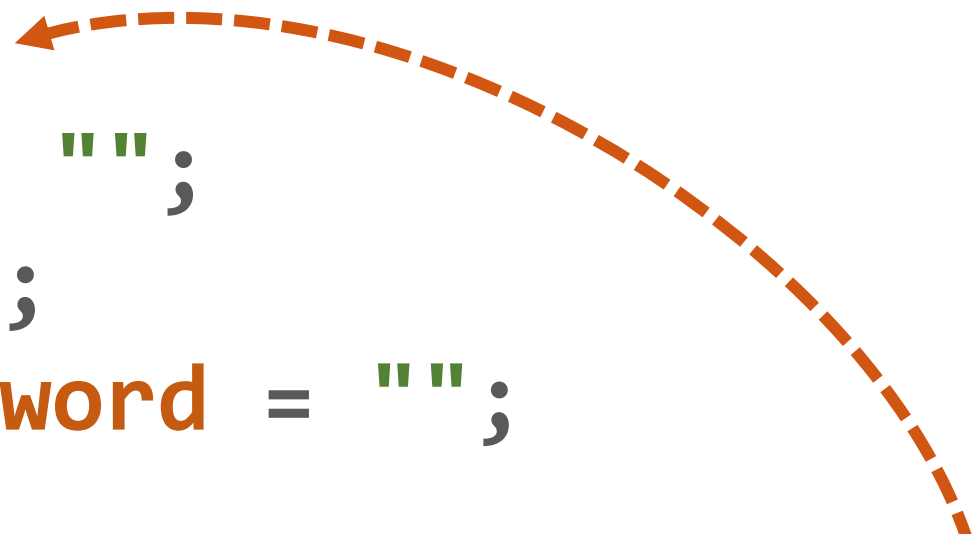
W przypadku kiedy obiekt ma odpowiadający sobie rząd w bazie danych ten atrybut będzie trzymać w sobie wartość **klucza głównego**, dzięki któremu będziemy wiedzieć do którego rzędu w tabeli będzie przypisany obiekt.

Wartość inną niż **-1** będziemy przypisywać tylko z danych pochodzących z bazy danych.

# User - konstruktor

Na początku napiszemy konstruktor. Konstruktor w naszej klasie nie będzie przyjmować żadnych argumentów i będzie nastawiał wszystkie jej atrybuty na domyślne (puste) wartości. Jedynie id nowo stworzonego użytkownika będzie nastawiane na -1.

```
public function __construct() {  
    $this->id = -1;  
    $this->username = "";  
    $this->email = "";  
    $this->hashedPassword = "";  
}
```



Nastawiamy id na -1 jako że ten obiekt nie jest połączony z żadnym rzędem w bazie danych

# Getery i setery

W następnym kroku piszemy potrzebne nam getery i setery. Dzięki nim będziemy mogli mieć dostęp do naszych atrybutów.

Nie będziemy pisali setera dla atrybutu id. Nie chcemy żeby nikt poza naszą klasą mógł zmieniać ten atrybut. Mogło by o spowodować błędy w naszej bazie danych (pamiętaj że atrybut id trzyma w sobie klucz główny albo wartość -1).

Wyróżniać się będzie też seter dla hasła. Będzie on od nowa haszował nasze hasło żeby było przygotowane do zapisania w bazie danych.

```
public function setPassword($newPassword)
{
    $newHashedPassword =
        password_hash($newPassword,
            PASSWORD_BCRYPT);

    $this->hashedPassword =
        $newHashedPassword;
}
```



# Zapisywanie nowego obiektu do bazy danych

Następnym krokiem jest umożliwienie zapisania nowego obiektu do bazy danych. W tym celu napiszemy metodę **saveToDB()**. Metoda ta będzie przyjmowała jeden argument – obiekt klasy `mysql` dzięki któremu będziemy mogli wywoływać zapytania SQL.

Na samym początku tej metody musimy sprawdzić czy obiekt nie jest już w naszej bazie danych. Zrobimy to sprawdzając czy jego `id` jest równe `-1`.



# Zapisywanie nowego obiektu do bazy danych

```
public function saveToDB(mysqli $connection){  
    if($this->id == -1){  
        //Saving new user to DB  
  
        $sql = "INSERT INTO Users(username, email, hashed_password)  
                VALUES ('$this->username', '$this->email', '$this->hashedPassword')";  
  
        $result = $connection->query($sql);  
  
        if($result == true){  
            $this->id = $connection->insert_id;  
            return true;  
        }  
    }  
    return false;  
}
```

Zapisujemy obiekt do bazy tylko  
jeżeli jego id jest równe -1

Jeżeli udało się nam zapisać  
obiekt do bazy to przypisujemy  
mu klucz główny jako id

# Use Case: zapianie nowego użytkownika

Aby przetestować czy napisana przez nas metoda działa poprawnie stworzymy przypadek rejestracji nowego użytkownika. Taki scenariusz będzie wyglądał następująco:

1. Tworzymy nowy obiekt klasy **User**,
2. Wypełniamy odpowiednie dane używając seterów,
3. Używamy metody **saveToDB** żeby zapisać dane do bazy.

Następnie należy sprawdzić czy:

1. W bazie danych pojawił się nowy wpis?
2. Czy wpis w bazie danych ma wszystkie dane odpowiednio ustawione?
3. Czy obiekt ma ustawione poprawne id?

**Pamiętaj że baza danych nie pozwoli Ci zapisać dwóch użytkowników o tym samym emailu!**

# Wczytywanie obiektu z bazy danych

Kolejną metodą do napisania będzie metoda wczytująca jeden rząd z bazy danych i zamieniająca go w obiekt.

Będzie to metoda **statyczna** naszej klasy (będziemy ją wywoływać na klasie a nie na obiekcie). Wszystkie metody wczytujące obiekty z bazy danych będą statyczne – nie potrzebujemy przecież żadnego użytkownika (instancji obiektu) żeby wczytać innych użytkowników.

Metoda ta będzie pobierała jako argument obiekt klasy `mysqli` i id obiektu do wczytania, a będzie zwracała obiekt klasy **User**, albo **NULL** (w przypadku jeżeli podane id nie występuje w naszej bazie danych).

Inne metody, które wczytują jednego użytkownika będą wyglądać podobnie (możemy np. szukać po mailu albo imieniu a nie po id).

# Wczytywanie obiektu z bazy danych

```
static public function loadUserId(mysql $connection, $id){  
    $sql = "SELECT * FROM Users WHERE id=$id";
```

Funkcja jest statyczna – możemy jej używać na klasie a nie na obiekcie.

```
    $result = $connection->query($sql);  
    if($result == true && $result->num_rows == 1){  
        $row = $result->fetch_assoc();
```

```
        $loadedUser = new User();  
        $loadedUser->id = $row['id'];  
        $loadedUser->username = $row['username'];  
        $loadedUser->hashedPassword = $row['hashed_password'];  
        $loadedUser->email = $row['email'];
```

Tworzymy nowy obiekt użytkownika i nastawiamy mu odpowiednie parametry. Jako, że jesteśmy w środku klasy mamy dostęp do własności prywatnych, mimo działania w metodzie statycznej.

```
        return $loadedUser;
```

Zwracamy obiekt użytkownika albo NULL

```
    }  
    return null;
```

```
}
```

# Use Case: wczytanie użytkownika

Aby przetestować czy napisana przez nas metoda działa poprawnie użyjmy scenariusza. Taki scenariusz będzie wyglądał w ten sposób:

1. Wywołujemy statyczną metodę **loadUserById()** podając jej id istniejące w bazie,

Następnie należy sprawdzić czy:

1. Czy metoda zwróciła nam obiekt a nie **NULL**?
2. Czy obiekt ma wszystkie dane takie same jak zapisane w bazie danych?

Możemy także zrobić drugi przypadek sprawdzający działanie:

1. Wywołujemy statyczną metodę **loadUserById()** podając jej id nie istniejące w bazie,

Następnie należy sprawdzić czy:

1. Czy metoda zwróciła nam **NULL**?



# Wczytywanie wielu obiektów

Kolejną metodą do napisania będzie metoda wczytująca wszystkie rzędy z bazy danych i zamieniająca je w obiekty.

Takich metod może być wiele, np. wyszukiwanie wszystkich użytkowników którzy mają imię zaczynające się na jakąś literę, są urodzeni danego dnia (jeżeli oczywiście trzymamy datę urodzenia w bazie danych) itd.

Metoda ta będzie pobierała jako argument obiekt klasy `mysqli`, a będzie zwracała tablicę obiektów klasy **User**, albo pustą tablicę (w przypadku jeżeli żaden rząd nie spełnia wymogów).

# Wczytywanie obiektu z bazy danych

```
static public function loadAllUsers(mysqli $connection){
```

```
    $sql = "SELECT * FROM Users";
```

```
    $ret = [];
```

Tworzymy pustą tablicę którą potem  
wypełnimy obiektami wczytanymi z  
bazy danych

```
    $result = $connection->query($sql);
```

```
    if($result == true && $result->num_rows != 0){
```

```
        foreach($result as $row){
```

```
            $loadedUser = new User();
```

```
            $loadedUser->id = $row['id'];
```

```
            $loadedUser->username = $row['username'];
```

```
            $loadedUser->hashedPassword = $row['hashed_password'];
```

```
            $loadedUser->email = $row['email'];
```

```
            $ret[] = $loadedUser;
```

```
        }
```

```
    }
```

```
    return $ret;
```

```
}
```

Po stworzeniu i wypełnieniu obiektu  
danymi wkładamy go do tablicy którą  
pod koniec zwracamy

Tworzymy nowy obiekt użytkownika i  
nastawiamy mu odpowiednie parametry.  
Jako że jesteśmy w środku klasy mamy  
dostęp do własności prywatnych, mimo  
działania w metodzie statycznej.



# Use Case: wczytanie wszystkich użytkowników

Aby przetestować czy napisana przez nas metoda działa poprawnie użyjmy scenariusza. Taki scenariusz będzie wyglądał w ten sposób:

1. Wywołujemy statyczną metodę **loadAllUsers()**,

Następnie należy sprawdzić czy:

1. Czy metoda zwróciła nam tablicę?
2. Czy ilość obiektów w tablicy jest taka sama jak ilość rzędów w bazie danych?
3. Czy obiekty mają wszystkie dane takie same jak zapisane w bazie danych (wystarczy sprawdzić jeden losowy obiekt)?

# Modyfikacja obiektu

Kolejną metodą do napisania będzie metoda zmieniająca dane obiektu, który już istnieje w bazie danych. Będzie to rozwinięcie napisanej już przez nas metody **saveToDB()**.

Na początku metody **saveToDB** sprawdzaliśmy czy obiekt nie jest jeszcze zapisany w bazie. Dopiszemy do tego sprawdzenia else w którym napiszemy kod aktualizujący dane znajdujące się w bazie.

# Modyfikacja obiektu

```
public function saveToDB(mysqli $connection){  
    if($this->id == -1){  
        ...  
    }  
    else{  
        $sql = "UPDATE Users SET username='$this->username',  
                email='$this->email',  
                hashed_password='$this->hashedPassword'  
                WHERE id=$this->id";  
  
        $result = $connection->query($sql);  
        if($result == true){  
            return true;  
        }  
    }  
    return false;  
}
```

Zwracamy true albo false.

# Use Case: modyfikacja użytkownika

Aby przetestować czy napisana przez nas metoda działa poprawnie użyjmy scenariusza. Taki scenariusz będzie wyglądał w ten sposób:

1. Wywołujemy statyczną metodę **loadUserById()** podając jej id istniejące w bazie,
2. Wprowadzamy zmiany do wczytanego użytkownika za pomocą odpowiednich geterów i seterów,
3. Zapisujemy zmienionego użytkownika do bazy.

Następnie należy sprawdzić czy:

1. Czy wpis w bazie danych ma wszystkie dane odpowiednie ponastawiane?
2. Czy obiekt ma ustawione poprawne id?
3. Czy nie dodał się nowy wpis w bazie?

# Usunięcie obiektu

Kolejną metodą do napisania będzie metoda usuwająca obiekt z bazy danych. Powinna być ona wywoływana na obiekcie, który jest już zapisany do bazy danych.

Na początku metody **delete** będziemy musieli sprawdzić czy obiekt jest już zapisany w bazie danych. Zrobimy to sprawdzając czy jego **id** jest różne od **-1**.

Jeżeli obiekt nie jest zapisany w bazie danych to metoda nie będzie nic robić.

# Usunięcie obiektu

```
public function delete(mysqli $connection){  
    if($this->id != -1){  
        $sql = "DELETE FROM Users WHERE id=$this->id";  
        $result = $connection->query($sql);  
        if($result == true){  
            $this->id = -1;  
            return true;  
        }  
        return false;  
    }  
    return true;  
}
```

Jako, że usnęliśmy obiekt to  
zmieniamy jego id na -1

Jeżeli obiektu nie było wcześniej  
w bazie danych to możemy od  
razu zwrócić true

# Use Case: usuwanie użytkownika

Aby przetestować czy napisana przez nas metoda działa poprawnie użyjmy scenariusza. Taki scenariusz będzie wyglądał w ten sposób:

1. Wywołujemy statyczną metodę **loadUserById()** podając jej id istniejące w bazie,
2. Na wczytanym użytkowniku używamy metody **delete()**.

Następnie należy sprawdzić czy:

1. Czy obiekt ma ustawione id na -1?
2. Czy wpis o podanym id został usunięty z bazy danych?





# Twitter

# Twitter

Celem warsztatów jest napisanie pełnej i funkcjonalnej aplikacji w stylu Twittera. Aplikacja ma implementować następujące funkcjonalności:

- **Użytkownicy:** dodawanie, modyfikacja niekluczowych informacji o sobie, usuwanie swojego konta. Użytkownik ma być identyfikowany po emailu (nie może się powtarzać).
- **Wpisy:** Każdy użytkownik może stworzyć nieograniczoną liczbę wpisów. Maksymalna długość wpisu to 140 znaków.
- **Komentarze:** pod każdym wpisem inni użytkownicy mają mieć możliwość wpisywania komentarzy. Maksymalna długość komentarza to 60 znaków.
- **Wiadomości:** Każdy użytkownik może wysłać innemu użytkownikowi wiadomość.

# Strony, które ma mieć aplikacja

## Strona główna

Strona wyświetlająca wszystkie Tweety jakie znajdują się w systemie (od najnowszego do najstarszego). Nad nimi ma być widoczny formularz do stworzenia nowego wpisu.

## Strona logowania

Strona ma przyjmować email użytkownika i jego hasło.

- Jeżeli są poprawne, to użytkownik jest przekierowany do strony głównej, jeżeli nie – do strony logowania, która ma wtedy wyświetlić komunikat o błędnym loginie lub hasle.

Strona logowania ma mieć też link do strony tworzenia użytkownika.

## Strona tworzenia użytkownika

Strona ma pobierać email i hasło.

- Jeżeli takiego emaila nie ma jeszcze w systemie, to dodać go i zalogować (przekierować na stronę główną).
- Jeżeli taki email jest, to przekierować znowu do strony tworzenia użytkownika i wyświetlić komunikat o zajętych adresie email.

## Strona wyświetlania użytkownika

Strona ma pokazać wszystkie wpisy danego użytkownika (dodatkowo pod każdym liczbę komentarzy do danego wpisu).

Na tej stronie ma być też guzik, który umożliwi nam wysłanie wiadomości do tego użytkownika.

# Strony, które ma mieć aplikacja

## Strona wyświetlania postu

Ta strona ma wyświetlać:

- post,
- autora postu,
- wszystkie komentarze do każdego z postów.
- Formularz do tworzenia nowego komentarza przypisanego do tego postu

## Strona edycji użytkownika

Użytkownik ma mieć możliwość edycji informacji o sobie i zmiany hasła. Pamiętaj o tym, że użytkownik może edytować tylko i wyłącznie swoje informacje.

## Strona z wiadomościami

Użytkownik ma mieć możliwość wyświetlenia listy wiadomości, które otrzymał i wysłał.

- Wiadomości wysłane mają wyświetlać odbiorcę, datę wysłania i początek wiadomości (pierwsze 30 znaków).
- Wiadomości odebrane mają wyświetlać nadawcę, datę wysłania i początek wiadomości (pierwsze 30 znaków).

Wiadomości jeszcze nieprzeczytane powinny być jakoś oznaczone.

## Stronę wiadomości

Wszystkie informacje o wiadomości: nadawca, odbiorca, treść.

# Informacje

- Bez zalogowania mają być dostępne tylko:
  - strona logowania,
  - strona tworzenia nowego użytkownika.

- Pod koniec zajęć wyślij wykładowcy informację zawierającą następujące dane:
  - adres repozytorium,  
na którym znajduje się twój kod,
  - lista zaimplementowanych funkcjonalności,
  - zrzut bazy danych.

# Zadania



# Zadanie 1

## Przygotowanie

- Przygotuj folder pod aplikację.
  - Załóż nowe repozytorium Git na GitHubie i nową bazę danych.
  - Pamiętaj o robieniu backupów bazy danych, (najlepiej co każde ćwiczenie) i tworzeniu commitów (również co każde ćwiczenie).
- Stwórz plik .gitignore i dodaj do niego wszystkie podstawowe dane: (pliki \*.\*~, katalog z danymi twojego IDE jeżeli istnieje itp.).
  - Stwórz plik, który będzie służył do łączenia się z bazą danych.



# Zadanie 2

## Ćwiczenia z wykładowcą

- Podczas ćwiczeń z wykładowcą stworzysz szkielet aplikacji i klasę User (na podstawie schematu z prezentacji).

# Zadanie 3

## Wpisy

- Czas na dodanie głównej funkcjonalności do naszej strony – czyli wpisy.
- Stwórz w bazie danych tabelę, która będzie przechowywała wpisy.
- Pamiętaj o stworzeniu relacji między tą tabelą a tabelą użytkowników. Użytkownik może mieć wiele wpisów, wpis może mieć tylko jednego Usera.

Stwórz klasę Tweet.

Ma ona zawierać co najmniej:

- **id:** int, private
- **userId:** int, private
- **text:** string, private
- **creationDate:** date, private

Jeżeli chcesz w jakiś sposób rozwinąć funkcjonalność to jest to dozwolone.

# Zadanie 3 – klasa Tweet

Ma implementować następujące funkcjonalności:

- Set i get do wszystkich atrybutów (do id tylko get).
- Konstruktor nastawiający id na -1, a resztę danych zerujący.
- Funkcję loadTweetById (wzoruj się na klasie User).
- Funkcję loadAllTweetsByUserId (ma wczytać wszystkie tweety stworzone przed danego użytkownika).
- Funkcję loadAllTweets (wzoruj się na klasie User).
- Funkcję saveToDB (wzoruj się na register z klasy User).
- Jeżeli widzisz jeszcze jakieś potrzebne funkcje to możesz je dopisać.

# Zadanie 3 – ciąg dalszy

- Zmodyfikuj stronę główną tak, aby wyświetlała wszystkie wpisy.
  - Załaduj je do tabelki za pomocą funkcji `loadAllTweets`, a potem wypisz informację o każdym tweedzie, używając odpowiednich geterów.
  - Zmodyfikuj stronę główną tak, aby miała formularz do stworzenia nowego wpisu. Pamiętaj o obsłudze tego formularza na tej samej stronie. Ma on tworzyć nowy tweet przypisany do zalogowanego Usera.
- Zmodyfikuj stronę użytkownika tak, żeby wyświetlała wszystkie jego wpisy.
  - Stwórz stronę, która wyświetli wszystkie informacje o wpisie.

# Zadanie 4

## Komentarze

- Dodajemy możliwość pisania komentarzy pod wpisami.
- Stwórz w bazie danych tabelę trzymającą komentarze.
- Stwórz relację między komentarzami a wpisami.

Stwórz klasę Comment.

Ma ona zawierać:

- **Id:** int, private
- **Id\_usera:** int, private
- **Id\_postu:** int, private
- **Creation\_date:** datetime, private
- **Text:** string, private

# Zadanie 4 – ciąg dalszy

Aplikacja ma także implementować następujące funkcjonalności:

- Set i get do wszystkich atrybutów (do id tylko get).
- Konstruktor nastawiający id na -1 a resztę danych zerujący.
- Funkcję loadCommentById i loadAllCommentsByPostId
- Funkcję saveToDB (wzoruj się na register z klasy User).
- Aplikacja ma pokazywać komentarze posortowane od najnowszego do najstarszego.
- Zmodyfikuj stronę wpisów tak, aby wyświetlały wszystkie swoje komentarze i miały formularz do stworzenia nowego komentarza.
- Komentarz ma wyświetlić informację o autorze (musisz wczytać danego użytkownika).

# Zadanie 5

## Wiadomości

Czas na wysyłanie wiadomości.

- Stwórz w bazie danych tabelę, która będzie trzymała wiadomości.
- Połącz ją z tabelą użytkowników relacją wiele do dwóch. Czyli mają powstać dwie relacje wiele do jednego. Wiadomość ma dwóch użytkowników nadawcę i odbiorcę, a użytkownik ma wiele wiadomości.
- W tabeli stwórz pole trzymające informację, czy wiadomość została przeczytana np.:
  - 1 – wiadomość przeczytana,
  - 0 – wiadomość nieprzeczytana).

Stwórz klasę wiadomości, wzorując się na poprzednich zadaniach.



# Zadanie 5 – ciąg dalszy

- Stwórz stronę wyświetlającą wszystkie wiadomości, które użytkownik otrzymał i wysłał.
- Do strony wyświetlającej użytkownika dodaj guzik przekierowujący na stronę z formularzem do wysłania wiadomości do tego użytkownika (nie powinno się dać wysłać wiadomości do samego siebie!).
- Pamiętaj o tym, że nowo stworzona wiadomość powinna być oznaczona jako nieprzeczytana.

- Dodaj stronę, która wyświetli informację o wiadomości (jeżeli otwiera ją odbiorca, pamiętaj żeby oznaczyć wiadomość jako przeczytaną).

**Jeżeli coś nie jest jasne, zapytaj wykładowcę.**