

# Warsztaty II Java

v3.1

# Plan

1. Obiektowa praca z bazą danych
2. Active Record
3. Szkoła programowania
4. Zadania

# 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 rozwiązujące 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 współpracy z bazą danych jest poznanie zależności między klasami a tabelami w bazie danych.

W większości przypadków mamy takie założenia:

- na każdą tablicę w naszej bazie danych przypada jedna klasa, która ją reprezentuje,
- klasa ma taką samą nazwę jak tabela, oraz takie same atrybuty (odpowiadające kolumnom tabeli),
- każdy obiekt tej klasy jest reprezentacją jednego rzędu z tabeli,
- obiekt może być w dwóch stanach:
  - zsynchronizowanym,
  - niesynchronizowanym.

# 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 następujących przypadkach:

- wczytania rzędu z bazy do obiektu,
- zapisania obiektu do bazy danych.

# Rozsynchronizowanie obiektu

Obiekt jest niesynchronizowany z bazą danych w następujących przypadkach:

- dane trzymane w jego atrybutach **nie są takie same** jak dane w odpowiadającym mu rzędzie,
- w bazie danych nie ma rzędu odpowiadającego atrybutom obiektu.

Brak synchronizacji może się pojawić na skutek:

- 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 niesynchronizowany z bazą danych pod koniec działania naszego programu, to zmiany w nim zawarte nie zostaną zapisane do bazy danych!**

# Active Record



# Active Record

Jest to jeden z prostszych wzorców, służących do komunikacji z bazą danych. Dla każdej tabeli, której używamy w naszym programie, implementujemy osobną klasę. Klasa ma atrybuty odpowiadające kolumnom w tabeli. Dodatkowo klasa implementuje metody służące ustawianiu i pobieraniu wszystkich swoich atrybutów (getter i setter) oraz metody służące do komunikacji z bazą danych.

Zazwyczaj są to:

- **save()** – zapisuje obiekt do tabeli (jako nowy rząd),
- **update()** – zapisuje obiekt do tabeli (jako zmodyfikowany, istniejący wcześniej rząd tej tabeli),
- **delete()** – usuwa obiekt z tabeli (czyli usuwa rząd o **id** takim samym jak zapisane w obiekcie).

**Obiekt klasy reprezentować będzie nam jeden rząd w tabeli. Nasz obiekt służy 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 zestaw statycznych metod, które mają nam pomóc w wyszukiwaniu lub załadowaniu większej ilości danych.

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

Zazwyczaj są to:

- **`loadAll()`** – wczytuje wszystkie rzędy z tabeli, na podstawie każdego rzędu tworzy nowy obiekt, następnie zwraca tablicę 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.

# Active Record – przykład

Jako przykład możemy pokazać przechowywanie informacji o użytkownikach w bazie danych.

Klasa ma przechowywać następujące dane:

- **id** użytkownika (ustawiane przez bazę danych – zazwyczaj **auto\_increment**),
- **imię** użytkownika,
- **hasło** (zahaszowane),
- **email** użytkownika (unikalny w naszym systemie).

Chcemy, żeby nasza klasa miała możliwość:

- zapisu nowego użytkownika do bazy danych,
- edycji istniejącego użytkownika,
- usunięcia użytkownika,
- wczytania użytkownika po jego **id**,
- wczytania użytkownika po jego **emailu** (potrzebne do logowania),
- wczytania wszystkich użytkowników,
- zmiany wszystkich atrybutów użytkownika.

# Tabela users

W bazie danych będziemy mieli tabelę **users** opisaną następująco:

| 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    |                |
| password | varchar(60)  | NO   |     | NULL    |                |

# Tabela users

W bazie danych będziemy mieli tabelę **users** opisaną następująco:

| 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    |                |
| password | varchar(60)  | NO   |     | NULL    |                |

**auto\_increment** – klucz główny

# Tabela users

W bazie danych będziemy mieli tabelę **users** opisaną następująco:

| 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    |                |
| password | varchar(60)  | NO   |     | NULL    |                |

**auto\_increment** – klucz główny

**UNI** – Kolumna **email** jest unikalna. Zakładamy, że nie może być dwóch użytkowników z takim samym adresem.

# Klasa User

Tworzymy klasę **User** (klasy, które będą miały swoją reprezentację w bazie danych, możemy trzymać w oddzielnym pakiecie).

W klasie dodajemy następujące atrybuty (wszystkie prywatne):

- **id**
- **username**
- **password**
- **email**

```
package pl.coderslab.models;  
public class User {  
    private int id;  
    private String username;  
    private String password;  
    private String email;  
}
```



# User – id

Bardzo ważnym atrybutem klasy jest **id**.

W niektórych przypadkach wartość **id** będzie wynosiła **0**. Stanie się tak jeśli:

- obiekt nie ma jeszcze rzędu w bazie danych (np. gdy stworzyliśmy nowy obiekt, ale jeszcze nie zapisaliśmy go do bazy danych),
- usuwamy z bazy danych wcześniej wczytany obiekt.

SQL nigdy nie nada takiego klucza głównego.

Gdy obiekt ma odpowiadający sobie rząd w bazie danych, to atrybut **id** będzie trzymać w sobie wartość klucza głównego. Dzięki temu wiemy, do którego rzędu w tabeli jest przypisany obiekt.

Dane wczytywane z bazy zawsze będą miały wartość **id** inną niż **0**.



# User – konstruktor

Na początku stworzymy konstruktor. Będzie przyjmować wartości dla **username**, **password**, **email**. Przyjmujemy hasło w postaci tekstu, a dopiero setter dla pola zakoduje prawidłowo podane hasło.

```
public User(String username, String email, String password) {  
    this.username = username;  
    this.email = email;  
    this.setPassword(password);  
}
```

# User – konstruktor

Na początku stworzymy konstruktor. Będzie przyjmować wartości dla **username**, **password**, **email**. Przyjmujemy hasło w postaci tekstu, a dopiero setter dla pola zakoduje prawidłowo podane hasło.

```
public User(String username, String email, String password) {  
    this.username = username;  
    this.email = email;  
    this.setPassword(password);  
}
```

Używamy settera do zakodowania hasła – jego implementacją zajmiemy się za chwilę.

# Konstruktor bezparametrowy

Dodajmy do naszej klasy również konstruktor bezparametrowy.

```
public User() {}
```

Tworząc obiekt za pomocą konstruktora bezparametrowego jego atrybutom zostaną nadane wartości domyślne:

## Wartości domyślne – przypomnienie

```
User user = new User();  
user.id  
user.username  
user.hashPass  
user.email
```

# Konstruktor bezparametrowy

Dodajmy do naszej klasy również konstruktor bezparametrowy.

```
public User() {}
```

Tworząc obiekt za pomocą konstruktora bezparametrowego jego atrybutom zostaną nadane wartości domyślne:

## Wartości domyślne – przypomnienie

```
User user = new User();
```

```
user.id
```

```
user.username
```

```
user.hashPass
```

```
user.email
```

Typ prosty **int**, będzie miał wartość domyślną **0**.

# Konstruktor bezparametrowy

Dodajmy do naszej klasy również konstruktor bezparametrowy.

```
public User() {}
```

Tworząc obiekt za pomocą konstruktora bezparametrowego jego atrybutom zostaną nadane wartości domyślne:

## Wartości domyślne – przypomnienie

```
User user = new User();  
user.id  
user.username  
user.hashPass  
user.email
```

Typ prosty **int**, będzie miał wartość domyślną **0**.

Jako typy obiektowe będą miały wartość domyślną **null**.

# Gettery i settery

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

Nie będziemy pisali settera dla atrybutu **id**.

Nie chcemy, żeby ktoś poza naszą klasą mógł zmieniać ten atrybut. Mogło by to spowodować błędy w naszej bazie danych (pamiętaj, że atrybut **id** trzyma w sobie klucz główny lub wartość domyślną **0**).

# Gettery i settery

Wyróżniać się będzie też setter dla hasła.

Będzie on od nowa haszował nasze hasło, żeby było przygotowane do zapisania w bazie danych.

Skorzystamy z implementacji algorytmu **Blowfish** – JBCrypt:

<http://www.mindrot.org/projects/jBCrypt/>

```
public void setPassword(String password) {  
    this.password = BCrypt.hashpw(password, BCrypt.gensalt());  
}
```

# 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 **Connection**, dzięki któremu będziemy mogli wywoływać zapytania SQL.

Na samym początku tej metody musimy sprawdzić, czy obiekt nie jest już w naszej bazie danych. Jak? Musimy dowiedzieć się, czy jego **id** jest równe **0**.



# Zapisywanie nowego obiektu do bazy danych

```
public void saveToDB(Connection conn) throws SQLException {
    if (this.id == 0) {
        String sql = "INSERT INTO users(username, email, password) VALUES (?, ?, ?)";
        String[] generatedColumns = { "ID" };
        PreparedStatement preparedStatement =
            conn.prepareStatement(sql, generatedColumns);
        preparedStatement.setString(1, this.username);
        preparedStatement.setString(2, this.email);
        preparedStatement.setString(3, this.password);
        preparedStatement.executeUpdate();
        ResultSet rs = preparedStatement.getGeneratedKeys();
        if (rs.next()) {
            this.id = rs.getInt(1);
        }
    }
}
```

# Zapisywanie nowego obiektu do bazy danych

```
public void saveToDB(Connection conn) throws SQLException {  
    if (this.id == 0) {  
        String sql = "INSERT INTO users(username, email, password) VALUES (?, ?, ?)";  
        String[] generatedColumns = { "ID" };  
        PreparedStatement preparedStatement =  
            = conn.prepareStatement(sql, generatedColumns);  
        preparedStatement.setString(1, this.username);  
        preparedStatement.setString(2, this.email);  
        preparedStatement.setString(3, this.password);  
        preparedStatement.executeUpdate();  
        ResultSet rs = preparedStatement.getGeneratedKeys();  
        if (rs.next()) {  
            this.id = rs.getInt(1);  
        }  
    }  
}
```

Zapisujemy obiekt do bazy, tylko wtedy gdy jego **id** jest równe **0**.

# Zapisywanie nowego obiektu do bazy danych

```
public void saveToDB(Connection conn) throws SQLException {
    if (this.id == 0) {
        String sql = "INSERT INTO users(username, email, password) VALUES (?, ?, ?)";
        String[] generatedColumns = { "ID" };
        PreparedStatement preparedStatement =
            conn.prepareStatement(sql, generatedColumns);
        preparedStatement.setString(1, this.username);
        preparedStatement.setString(2, this.email);
        preparedStatement.setString(3, this.password);
        preparedStatement.executeUpdate();
        ResultSet rs = preparedStatement.getGeneratedKeys();
        if (rs.next()) {
            this.id = rs.getInt(1);
        }
    }
}
```

Pobieramy wstawiony do bazy identyfikator, a następnie ustawiamy **id** obiektu.

# Use Case – zapisanie nowego użytkownika

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

- tworzymy nowy obiekt klasy **User**,
- wypełniamy odpowiednie dane, używając **setterów**,
- używamy metody **saveToDB()**, żeby zapisać dane do bazy.

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

# Wczytywanie obiektu z bazy danych

Czas na napisanie metody wczytującej jeden rząd z bazy danych i zamieniającej 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 **Connection** oraz **id** obiektu do wczytania, a będzie zwracała obiekt klasy **User** albo **null** (jeżeli podane **id** nie występuje w naszej bazie danych).

Inne metody, które wczytują jeden rząd z bazy, będą wyglądać podobnie (możemy szukać użytkownika zamiast po **id**, np. po emailu albo imieniu).



# Wczytywanie obiektu z bazy danych

```
static public User loadUserById(Connection conn, int id) throws SQLException {  
    String sql = "SELECT * FROM users where id=?";  
    PreparedStatement preparedStatement = conn.prepareStatement(sql);  
    preparedStatement.setInt(1, id);  
    ResultSet resultSet = preparedStatement.executeQuery();  
    if (resultSet.next()) {  
        User loadedUser = new User();  
        loadedUser.id = resultSet.getInt("id");  
        loadedUser.username = resultSet.getString("username");  
        loadedUser.password = resultSet.getString("password");  
        loadedUser.email = resultSet.getString("email");  
        return loadedUser;  
    }  
    return null;  
}
```

# Wczytywanie obiektu z bazy danych

```
static public User loadUserById(Connection conn, int id) throws SQLException {  
    String sql = "SELECT * FROM users where id=?";  
    PreparedStatement preparedStatement = conn.prepareStatement(sql);  
    preparedStatement.setInt(1, id);  
    ResultSet resultSet = preparedStatement.executeQuery();  
    if (resultSet.next()) {  
        User loadedUser = new User();  
        loadedUser.id = resultSet.getInt("id");  
        loadedUser.username = resultSet.getString("username");  
        loadedUser.password = resultSet.getString("password");  
        loadedUser.email = resultSet.getString("email");  
        return loadedUser;  
    }  
    return null;  
}
```

Metoda jest statyczna – możemy jej używać na klasie, zamiast na obiekcie.

# Wczytywanie obiektu z bazy danych

```
static public User loadUserById(Connection conn, int id) throws SQLException {  
    String sql = "SELECT * FROM users where id=?";  
    PreparedStatement preparedStatement = conn.prepareStatement(sql);  
    preparedStatement.setInt(1, id);  
    ResultSet resultSet = preparedStatement.executeQuery();  
    if (resultSet.next()) {  
        User loadedUser = new User();  
        loadedUser.id = resultSet.getInt("id");  
        loadedUser.username = resultSet.getString("username");  
        loadedUser.password = resultSet.getString("password");  
        loadedUser.email = resultSet.getString("email");  
        return loadedUser;  
    }  
    return null;  
}
```

Tworzymy nowy obiekt typu **User** i ustawiamy mu odpowiednie parametry. Jesteśmy w środku klasy, mamy zatem dostęp do właściwości prywatnych, mimo korzystania z metody statycznej.



# Wczytywanie obiektu z bazy danych

```
static public User loadUserById(Connection conn, int id) throws SQLException {  
    String sql = "SELECT * FROM users where id=?";  
    PreparedStatement preparedStatement = conn.prepareStatement(sql);  
    preparedStatement.setInt(1, id);  
    ResultSet resultSet = preparedStatement.executeQuery();  
    if (resultSet.next()) {  
        User loadedUser = new User();  
        loadedUser.id = resultSet.getInt("id");  
        loadedUser.username = resultSet.getString("username");  
        loadedUser.password = resultSet.getString("password");  
        loadedUser.email = resultSet.getString("email");  
        return loadedUser;  
    }  
    return null;  
}
```

Zwracamy obiekt użytkownika albo **null**.

# Use Case – wczytanie użytkownika

Aby przetestować, czy napisana przez nas metoda działa poprawnie, użyjmy scenariusza. Taki scenariusz będzie wyglądał następująco:

- wywołujemy statyczną metodę **loadUserById()**, podając **id** istniejące w bazie.

Następnie należy sprawdzić, czy:

- metoda zwróciła nam obiekt, a nie **null**?
- obiekt ma wszystkie dane takie same, jak zapisane w bazie danych?

Drugi przypadek sprawdzający działanie:

- wywołujemy statyczną metodę **loadUserById()**, podając **id** nieistniejące w bazie,

a następnie należy sprawdzić, czy:

- metoda zwróciła nam **null**?

# Wczytywanie wielu obiektów

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

Takich metod może być wiele, np. wyszukiwanie wszystkich użytkowników, których imię zaczyna się na podaną literę, lub urodzili się jakiegoś konkretnego dnia (jeżeli oczywiście trzymamy datę urodzenia w bazie danych) itd.

Metoda ta będzie pobierała obiekt klasy **Connection** jako argument, natomiast będzie zwracała tablicę obiektów klasy **User** albo pustą tablicę (jeżeli żaden rząd nie spełnia wymogów).

# Wczytywanie obiektu z bazy danych

```
static public User[] loadAllUsers(Connection conn) throws SQLException {  
    ArrayList<User> users = new ArrayList<User>();  
    String sql = "SELECT * FROM users";  
    PreparedStatement preparedStatement = conn.prepareStatement(sql);  
    ResultSet resultSet = preparedStatement.executeQuery();  
    while (resultSet.next()) {  
        User loadedUser = new User();  
        loadedUser.id = resultSet.getInt("id");  
        loadedUser.username = resultSet.getString("username");  
        loadedUser.password = resultSet.getString("password");  
        loadedUser.email = resultSet.getString("email");  
        users.add(loadedUser);  
    }  
    User[] uArray = new User[users.size()]; uArray = users.toArray(uArray);  
    return uArray;  
}
```

# Wczytywanie obiektu z bazy danych

```
static public User[] loadAllUsers(Connection conn) throws SQLException {  
    ArrayList<User> users = new ArrayList<User>();  
    String sql = "SELECT * FROM users";  
    PreparedStatement preparedStatement = conn.prepareStatement(sql);  
    ResultSet resultSet = preparedStatement.executeQuery();  
    while (resultSet.next()) {  
        User loadedUser = new User();  
        loadedUser.id = resultSet.getInt("id");  
        loadedUser.username = resultSet.getString("username");  
        loadedUser.password = resultSet.getString("password");  
        loadedUser.email = resultSet.getString("email");  
        users.add(loadedUser);  
    }  
    User[] uArray = new User[users.size()]; uArray = users.toArray(uArray);  
    return uArray;  
}
```

Pomocniczo wykorzystujemy obiekt klasy **ArrayList**, traktujmy go jako rozszerzalną tablicę. Więcej o kolekcjach dowiemy się w kolejnych modułach.



# Wczytywanie obiektu z bazy danych

```
static public User[] loadAllUsers(Connection conn) throws SQLException {
    ArrayList<User> users = new ArrayList<User>();
    String sql = "SELECT * FROM users";
    PreparedStatement preparedStatement = conn.prepareStatement(sql);
    ResultSet resultSet = preparedStatement.executeQuery();
    while (resultSet.next()) {
        User loadedUser = new User();
        loadedUser.id = resultSet.getInt("id");
        loadedUser.username = resultSet.getString("username");
        loadedUser.password = resultSet.getString("password");
        loadedUser.email = resultSet.getString("email");
        users.add(loadedUser);
    }
    User[] uArray = new User[users.size()]; uArray = users.toArray(uArray);
    return uArray;
}
```

Tworzymy nowy obiekt użytkownika i ustawiamy mu odpowiednie parametry. **Jesteśmy w środku klasy, mamy zatem dostęp do własności prywatnych, mimo działania w metodzie statycznej.**

# Wczytywanie obiektu z bazy danych

```
static public User[] loadAllUsers(Connection conn) throws SQLException {  
    ArrayList<User> users = new ArrayList<User>();  
    String sql = "SELECT * FROM users";  
    PreparedStatement preparedStatement = conn.prepareStatement(sql);  
    ResultSet resultSet = preparedStatement.executeQuery();  
    while (resultSet.next()) {  
        User loadedUser = new User();  
        loadedUser.id = resultSet.getInt("id");  
        loadedUser.username = resultSet.getString("username");  
        loadedUser.password = resultSet.getString("password");  
        loadedUser.email = resultSet.getString("email");  
        users.add(loadedUser);  
    }  
    User[] uArray = new User[users.size()]; uArray = users.toArray(uArray);  
    return uArray;  
}
```

Po stworzeniu i wypełnieniu obiektu danymi dodajemy go do listy.

# Wczytywanie obiektu z bazy danych

```
static public User[] loadAllUsers(Connection conn) throws SQLException {  
    ArrayList<User> users = new ArrayList<User>();  
    String sql = "SELECT * FROM users";  
    PreparedStatement preparedStatement = conn.prepareStatement(sql);  
    ResultSet resultSet = preparedStatement.executeQuery();  
    while (resultSet.next()) {  
        User loadedUser = new User();  
        loadedUser.id = resultSet.getInt("id");  
        loadedUser.username = resultSet.getString("username");  
        loadedUser.password = resultSet.getString("password");  
        loadedUser.email = resultSet.getString("email");  
        users.add(loadedUser);  
    }  
    User[] uArray = new User[users.size()]; uArray = users.toArray(uArray);  
    return uArray;  
}
```

Tworzymy tablicę elementów typu **User** o rozmiarze pobranym z listy.



# Wczytywanie obiektu z bazy danych

```
static public User[] loadAllUsers(Connection conn) throws SQLException {
    ArrayList<User> users = new ArrayList<User>();
    String sql = "SELECT * FROM users";
    PreparedStatement preparedStatement = conn.prepareStatement(sql);
    ResultSet resultSet = preparedStatement.executeQuery();
    while (resultSet.next()) {
        User loadedUser = new User();
        loadedUser.id = resultSet.getInt("id");
        loadedUser.username = resultSet.getString("username");
        loadedUser.password = resultSet.getString("password");
        loadedUser.email = resultSet.getString("email");
        users.add(loadedUser);
    }
    User[] uArray = new User[users.size()]; uArray = users.toArray(uArray);
    return uArray;
}
```

Przekształcamy listę na tablicę.

# Use Case – wczytanie wszystkich użytkowników

Aby przetestować, czy napisana przez nas metoda działa poprawnie, użyjmy scenariusza.

Taki scenariusz będzie wyglądał następująco:

- wywołujemy statyczną metodę **loadAllUsers()**.

Następnie należy sprawdzić, czy:

- metoda zwróciła tablicę,
- liczba obiektów w tablicy jest taka sama, jak liczba rzędów w bazie danych,
- 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

```
if (this.id == 0) {  
    ...  
} else {  
    String sql = "UPDATE users SET username=?, email=?, password=? where id = ?";  
    PreparedStatement preparedStatement = conn.prepareStatement(sql);  
    preparedStatement.setString(1, this.username);  
    preparedStatement.setString(2, this.email);  
    preparedStatement.setString(3, this.password);  
    preparedStatement.setInt(4, this.id);  
    preparedStatement.executeUpdate();  
}
```

# Modyfikacja obiektu

Aby przetestować, czy napisana przez nas metoda działa poprawnie, użyjmy scenariusza.

Taki scenariusz będzie wyglądał następująco:

- wywołujemy statyczną metodę **loadUserById()**, podając jej **id** istniejące w bazie.
- wprowadzamy zmiany do wczytanego użytkownika, za pomocą odpowiednich getterów i setterów.
- zapisujemy zmienione dane użytkownika do bazy.

Następnie należy sprawdzić, czy:

- wpis w bazie danych ma wszystkie dane odpowiednio poustawiane,
- obiekt ma ustawione poprawne **id**,
- 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. Musimy dowiedzieć się, czy jego **id** jest różne od **0**.

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

# Usunięcie obiektu

```
public void delete(Connection conn) throws SQLException {  
    if (this.id != 0) {  
        String sql = "DELETE FROM users WHERE id=?";  
        PreparedStatement preparedStatement = conn.prepareStatement(sql);  
        preparedStatement.setInt(1, this.id);  
        preparedStatement.executeUpdate();  
        this.id = 0;  
    }  
}
```



# Usunięcie obiektu

```
public void delete(Connection conn) throws SQLException {  
    if (this.id != 0) {  
        String sql = "DELETE FROM users WHERE id=?";  
        PreparedStatement preparedStatement = conn.prepareStatement(sql);  
        preparedStatement.setInt(1, this.id);  
        preparedStatement.executeUpdate();  
        this.id = 0;  
    }  
}
```

Usunęliśmy obiekt, zmieniamy zatem jego **id** na **0**.



# Use Case – usuwanie użytkownika

Aby przetestować, czy napisana przez nas metoda działa poprawnie, użyjemy następującego scenariusza:

- wywołujemy statyczną metodę **loadUserById()**, podając jej **id** istniejące w bazie.
- na wczytanym użytkowniku używamy metody **delete()**.

Następnie należy sprawdzić, czy:

- obiekt ma ustawione **id** na **0**?

# Szkoła programowania

# Szkoła programowania

Celem warsztatów jest napisanie obiektowej, bazodanowej warstwy aplikacji dla szkoły programowania.

Aplikacja będzie zawierać część potencjalnych funkcjonalności – przechowywanie rozwiązań do zadań, wykonywanych przez kursantów.

Podczas kolejnych warsztatów zajmiemy się oprogramowaniem interfejsu użytkownika dla naszej aplikacji.

**Weryfikacja działania** powinna nastąpić poprzez wywołanie metod w programach opisanych w dalszej części.

# Funkcjonalności

## Użytkownicy

Użytkownik ma być identyfikowany po emailu (nie może się powtarzać).

## Grupy

Użytkownik przynależy tylko do jednej grupy.  
Grupa może mieć wielu użytkowników.

## Zadanie

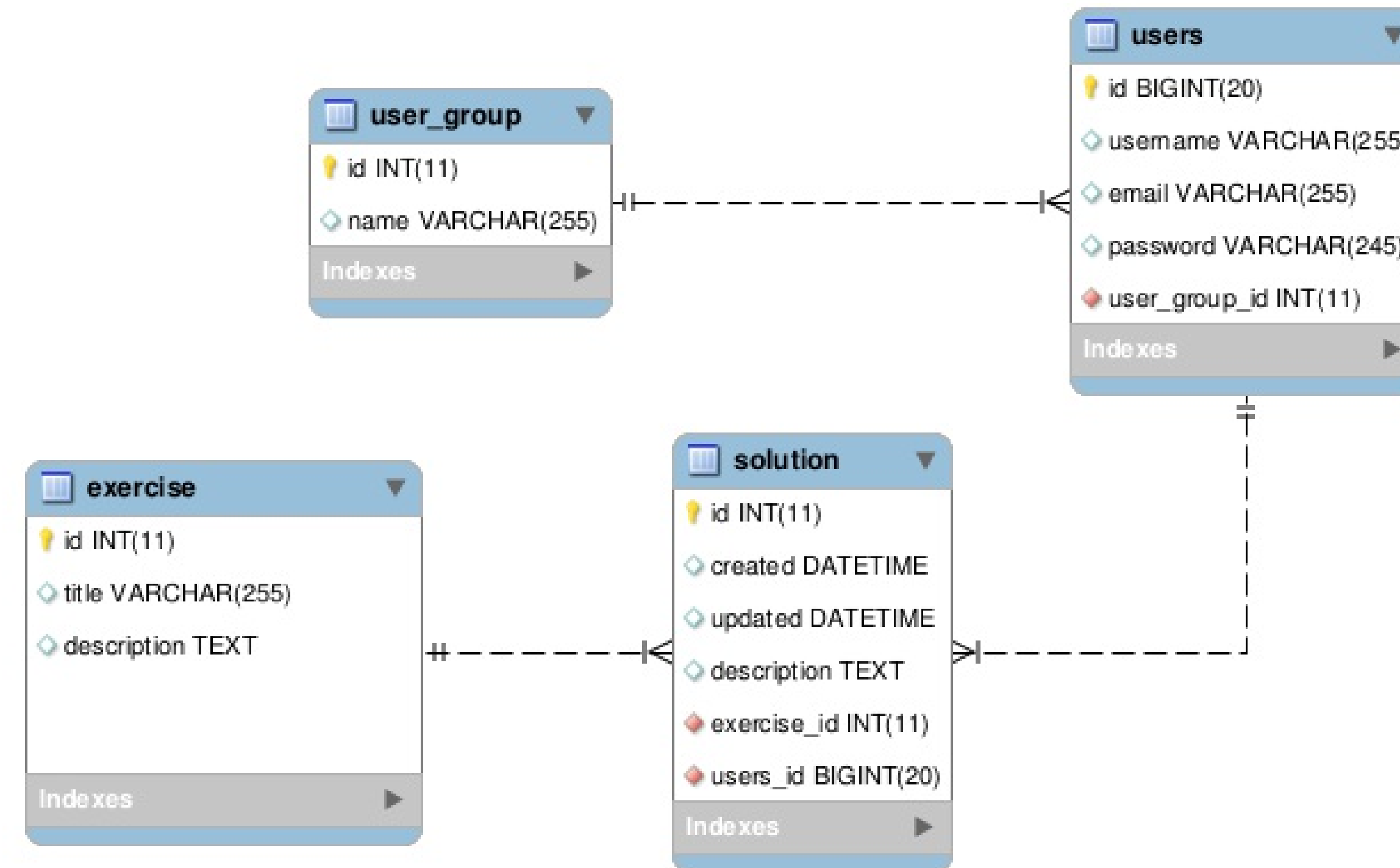
Zadanie do rozwiązania – może mieć więcej niż jedno rozwiązanie.

## Rozwiązanie zadania

Musi zawierać informacje:

- jakiego zadania dotyczy,
- kiedy zostało dodane,
- który użytkownik dodał rozwiązanie.

# Schemat danych



Schemat stanowi podstawę do dalszego samodzielnego rozwoju aplikacji.

# 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.).
- Utwórz klasę zawierającą metodę o sygnaturze:  

```
public static Connection getConnection() throws SQLException
```

  
, która będzie zwracała utworzone połączenie z bazą danych.



# Zadanie 2

## Ćwiczenia z wykładowcą

Podczas ćwiczeń z wykładowcą, stwórz szkielet aplikacji i klasę **User** (na podstawie schematu z prezentacji).

# Zadanie 3

## Ćwiczenia samodzielne

Stwórz wszystkie tabele w bazie danych potrzebne do działania programu.  
Pamiętaj o dodaniu kluczy głównych oraz powiązań między tabelami.

# Zadanie 4

## Pozostałe klasy

Utwórz implementację pozostałych klas:

- Group
- Exercise
- Solution

## Metody

Dla każdej klasy utwórz odpowiednie metody:

- loadAll
- loadById
- delete
- saveToDB

# Zadanie 5

## Dodatkowe metody:

Utwórz implementację dodatkowych metod realizujących zadania:

- pobranie wszystkich rozwiązań danego użytkownika (dopisz metodę **loadAllByUserId** do klasy **Solution**),
- pobranie wszystkich rozwiązań danego zadania, posortowanych od najnowszego do najstarszego (dopisz metodę **loadAllByExerciseId** do klasy **Solution**),
- pobranie wszystkich członków danej grupy (dopisz metodę **loadAllByGroupId** do klasy **User**).

# Programy administracyjne

# Zadanie 1

## Program 1 – zarządzanie użytkownikami

Program po uruchomieniu wyświetli na konsoli listę wszystkich użytkowników.

Następnie wyświetli

"Wybierz jedną z opcji:

- **add** – dodanie użytkownika,
- **edit** – edycja użytkownika,
- **delete** – usunięcie użytkownika,
- **quit** – zakończenie programu."

Po wpisaniu i zatwierdzeniu jednej z opcji program odpyta o odpowiednie dane:

- jeśli wybrano **add** – program zapyta o wszystkie dane, występujące w klasie **User**, oprócz **id**,
- w przypadku **edit** – o wszystkie dane występujące w klasie **User** oraz **id**,
- po wybraniu **delete** – zapyta o **id** użytkownika, którego należy usunąć.

Po wykonaniu dowolnej z opcji, program ponownie wyświetli listę danych i zada pytanie o wybór opcji.



# Zadanie 2

## Program 2 – zarządzanie zadaniami

Program po uruchomieniu wyświetli na konsoli listę wszystkich zadań.

Następnie wyświetli w konsoli napis

"Wybierz jedną z opcji:

- **add** – dodanie zadania,
- **edit** – edycja zadania,
- **delete** – edycja zadania,
- **quit** – zakończenie programu."

Po wpisaniu i zatwierdzeniu odpowiedniej opcji program odpyta o następujące dane:

- w przypadku **add** – o wszystkie dane występujące w klasie **Exercise** bez **id**,
- po wybraniu **edit** – wszystkie dane występujące w klasie **Exercise** oraz **id**,
- jeśli wybrano **delete** – zapyta o **id** zadania które należy usunąć.

Po wykonaniu dowolnej z opcji, program ponownie wyświetli listę danych i zada pytanie o wybór opcji.

# Zadanie 3

## Program 3 – zarządzanie grupami

Program po uruchomieniu wyświetli na konsoli listę wszystkich grup.

Następnie wyświetli w konsoli napis

"Wybierz jedną z opcji:

- **add** – dodanie grupy,
- **edit** – edycja grupy,
- **delete** – edycja grupy,
- **quit** – zakończenie programu."

Po wpisaniu i zatwierdzeniu odpowiedniej opcji program odpyta o dane i wykona odpowiednią operację:

- **add** – wszystkie dane występujące w klasie **Group**, bez **id**,
- **edit** – wszystkie dane występujące w klasie **Group** oraz **id**,
- **delete** – **id** grupy którą należy usunąć.

Po wykonaniu dowolnej z opcji, program ponownie wyświetli listę danych i zada pytanie o wybór opcji.

# Zadanie 4

## Program 4 – przypisywanie zadań

Program po uruchomieniu wyświetli w konsoli napis

"Wybierz jedną z opcji:

- **add** – przypisywanie zadań do użytkowników,
- **view** – przeglądanie rozwiązań danego użytkownika,
- **quit** – zakończenie programu."

Po wpisaniu i zatwierdzeniu odpowiedniej opcji program odpyta o dane:

- jeśli wybrano **add** – wyświetli listę wszystkich użytkowników, odpyta o **id**, następnie wyświetli listę wszystkich zadań i zapyta o **id** zadania, utworzy i zapisze obiekt typu **Solution**.

Pole **created** wypełni się automatycznie, a pola **updated** i **description** mają zostać puste.

- **view** – zapyta o **id** użytkownika, którego rozwiązania chcemy zobaczyć.

Po wykonaniu dowolnej z opcji, program ponownie zada pytanie o wybór opcji.

# Program Użytkownika

# Zadanie

## Dodawanie rozwiązań

Program przyjmie jeden parametr, podawany podczas uruchamiania z konsoli lub IDE, symbolizujący identyfikator Użytkownika.

Pamiętaj, że parametry takie pobieramy z tablicy **args** parametrów metody **main**.

```
public static void main(String[] args)
```

Program, po uruchomieniu, wyświetli w konsoli napis:

"Wybierz jedną z opcji:

- **add** – dodawanie rozwiązania,
- **view** – przeglądanie swoich rozwiązań."

# Zadanie

## Dodawanie rozwiązań

Po wybraniu odpowiedniej opcji, program odpyta o dane i wykona odpowiednią operację:

- po wybraniu **add** – wyświetli listę zadań, do których Użytkownik nie dodał jeszcze rozwiązania, a następnie odpyta o **id** zadania, do którego ma zostać dodane rozwiązanie. Pole **created** zostanie wypełnione automatycznie, więc Użytkownik zostanie odpytany jeszcze tylko o rozwiązanie zadania,
- w przypadku wybrania **quit** – program zakończy działanie.

Dla uproszczenia przyjmujemy, że dodanego rozwiązania nie możemy usuwać, ani edytować.

W przypadku próby dodania rozwiązania, które już istnieje, program ma wyświetlić odpowiedni komunikat.



# Zadanie dodatkowe

## Dodatkowe funkcjonalności

Zastanów się jakie dodatkowe programy warto by było dopisać.

Jeżeli masz pomysł na inne potrzebne metody lub klasy, to dopisz je do swojego programu.

Przykładowe możliwości rozwoju:

- oceny i komentarze do rozwiązań zadań,
- umiejętności przypisywane do Użytkowników.