Bazy danych MySQL w Javie

dzień 1

v3.1

Plan

1. Wprowadzenie do baz danych
2. Przygotowanie do pracy z MySQL
3. Trochę teorii o MySQL
4. MySQL i Java
5. Tworzenie bazy danych
6. Dodawanie danych do tabeli
7. Pobieranie danych z tabeli

Wprowadzenie do baz danych

Co to są bazy danych?

Bazy danych są to aplikacje, których celem jest przetrzymywanie, analizowanie i zwracanie danych.

Aplikacja bazodanowa musi umożliwiać:

- definicję danych,
- dodawanie, usuwanie i modyfikację danych,
- zarządzanie dostępami do danych.

Dlaczego stosujemy bazy danych?

Bazy danych stosujemy, gdy mamy zmienną liczbę danych w naszym projekcie.

Przechowywanie danych, w przeznaczonej do tego bazie, pozwoli nam na szybkie zarządzanie taką kolekcją i łatwe współdzielenie jej z innymi programami.

Bazy danych pomagają też w następujących zagadnieniach:

- gromadzeniu i przechowywaniu ogromnych zbiorów danych (są bardzo dobrze zoptymalizowane pamięciowo),
- szybkim przeszukiwaniu i sortowaniu zbiorów danych,
- łączeniu danych w relacje.

Typy baz danych

Hierarchiczne

Mówi się o hierarchicznych bazach danych, jeżeli – pomiędzy danymi zachowanymi w takim systemie – następuje relacja rodzic–dziecko. Hierarchiczny typ baz danych został stworzony przez IBM w 1968 roku i nie jest już stosowany.

Relacyjne

Bazy danych, które skupiają się na relacjach między danymi. Dane w takich bazach są przedstawiane jako dwuwymiarowe tabele, gdzie każda kolumna to atrybut, a rząd to dane.

Typy baz danych

Obiektowe

Bazy danych stworzone w oparciu o ideę programowania obiektowego. W pamięci przetrzymywane są obiekty odpowiadające poszczególnym klasom. Przydatne przy przetrzymywaniu plików multimedialnych. Nie są zbyt popularne, ponieważ są drogie w utrzymaniu.

Nierelacyjne

Najnowsze podejście do baz danych. Przechowujemy dane jako pary klucz–wartość, gdzie wartości nie mają ujednoliconej struktury. Łatwo skalowalne i szybkie przy dużych zestawach danych.

Najpopularniejsze bazy danych

Oracle

Relacyjna baza danych oparta na języku SQL (trochę rozbudowanym). Jest raczej używana przez duże firmy, istnieją tylko jej płatne wersje.

MySQL

Relacyjna baza danych oparta na języku SQL. Darmowa do zastosowań niekomercyjnych. Popularna w małych i średnich firmach.

MongoDB

Nierelacyjna baza danych oparta na przetrzymywaniu całych dokumentów. Staje się bardzo popularna w zastosowaniach webowych.

PostgreSQL

Relacyjna baza danych oparta na języku SQL. Wydana na licencji open source. Wygodna do stawiania prostych stron.

Dlaczego uczymy się MySQL?

Baza MySQL jest najczęściej stosowana w mniejszych firmach. Implementuje standard języka SQL.

Zalety

- dobrze skalowalna i bardzo szybka,
- łatwa w zarządzaniu,
- bezpieczna.

Typy relacji

W relacyjnych bazach danych występują trzy relacje między tabelami:

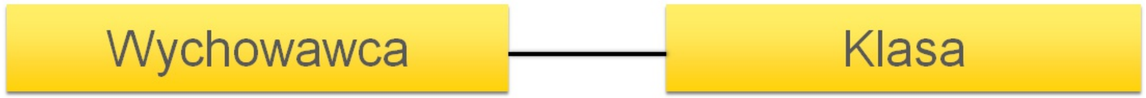
- jeden do jednego,
- jeden do wielu,
- wiele do wielu.

Relacja jeden do jednego

Relacja, w której jeden element z danej tabeli może być połączony tylko z jednym elementem z innej tabeli.

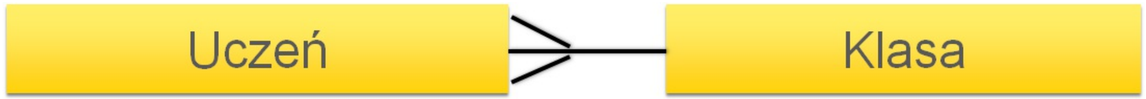
Przykład

- Nauczyciel może być wychowawcą tylko jednej klasy.
- Klasa może mieć tylko jednego wychowawcę.



Relacja jeden do wielu

Relacja, w której jeden element z danej tabeli, może być połączony z wieloma elementami z innej tabeli.



Przykład

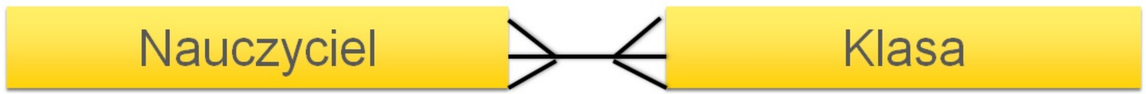
- Uczeń może należeć tylko do jednej klasy.
- Klasa może mieć wielu uczniów.

Relacja wiele do wielu

Relacja, w której wiele elementów z danej tabeli może być połączona z wieloma elementami z innej tabeli.

Przykład

- Klasa może być uczona przez wielu nauczycieli.
- Jeden nauczyciel może uczyć wiele klas.



Przygotowanie do pracy z MySQL

Instalacja i konfiguracja bazy danych MySQL

Systemy, skonfigurowane przez Was przy pomocy skryptu instalacyjnego przed rozpoczęciem kursu, mają już zainstalowany serwer MySQL.

Jeżeli jednak nie masz zainstalowanej bazy danych, na kolejnych slajdach znajdziesz wskazówki dotyczące instalacji na poszczególne systemy operacyjne.

Instalacja i konfiguracja w systemie Linux

Wystarczy nam komenda:

```
sudo apt-get install mysql-server
```

Instalator przeprowadzi nas przez wszystkie kroki potrzebne do zainstalowania i skonfigurowania bazy danych.

Instalacja i konfiguracja w systemie Windows

Wystarczy, że ściągniemy instalator ze strony:

http://dev.mysql.com/downloads/windows/installer

Instalator przeprowadzi nas przez wszystkie kroki potrzebne do zainstalowania i skonfigurowania bazy danych.

Instalacja i konfiguracja w systemie macOS

Wystarczy, że ściągniemy instalator ze strony:

http://dev.mysql.com/downloads/mysql

Instalator przeprowadzi nas przez wszystkie kroki potrzebne do zainstalowania i skonfigurowania bazy danych.

Sposoby dostępu do bazy danych

Jest wiele narzędzi wspomagających dostęp do baz danych i ich obsługę.

Oto najpopularniejsze z nich:

- konsola **mysql**,
- klient **mysqladmin**,
- panel **phpMyAdmin**,
- program **MySQL Workbench**,
- wrappery dla poszczególnych języków programowania.

Dostęp do MySQL – konsola

Chociaż konsola dostępu **`mysql`** jest najprostszym graficznie systemem dostępu do naszej bazy danych, to daje największe możliwości.

Konsola nie ma ograniczeń czasowych w wykonaniu zapytania, co przydatne jest w zapytaniach do dużych zbiorów danych.

Pamiętajmy, że każde zapytanie w konsoli mysql musi kończyć się średnikiem.

Dostęp do skonfigurowanej bazy danych:

- login: root
- hasło: coderslab

Dostęp do MySQL – konsola

Komenda uruchamiająca konsolę:

```
mysql -h hostname -u username -p [-D databasename]
```

Odpowiedź konsoli:

```
Welcome to the MySQL monitor. Commands end with ; or \g.
Your MySQL connection id is 32
…
Type 'help;' or '\h' for help. Type '\c' to clear
the current input statement.
mysql> _
```

Dostęp do MySQL – uruchamianie konsoli

Najważniejsze opcje przy uruchamianiu konsoli

- **`-h hostname`**

  Podajemy adres hosta, na którym znajduje się nasza baza (domyślnie **`localhost`**).

- **`-u username`**

  Podajemy nazwę użytkownika.

- **`-p`**

  Podczas logowania zostaniemy poproszeni o hasło. Można je też podać bezpośrednio bez spacji po argumencie "**`-p`**", np. **`-pcoderslab`**, gdzie hasło to: "**`coderslab`**".

- **`-D database`**

  Po zalogowaniu od razu zostanie załadowana podana baza danych (domyślnie nie zostanie załadowana żadna baza). Aby wybrać bazę ręcznie, po zalogowaniu używamy komendy: **`use nazwa_bazy;`**

Dostęp do MySQL – klient mysqladmin

Klient **mysqladmin** jest programem, który ma zaimplementowane komendy do najczęściej powtarzanych zadań administracyjnych.

Wpisanie poniższej komendy zwróci nam wszystkie opcje:

```
mysqladmin
```

Dostęp do MySQL – klient mysqladmin

Najczęściej używane opcje:

- **`-u username`**

  Wykonuje dane polecenie jako podany użytkownik.

- **`-p`**

  Pyta o hasło przed wykonaniem czynności.

- **`create nazwaBazy`**

  Tworzy nową bazę danych o podanej nazwie.

- **`drop nazwaBazy`**

  Niszczy bazę o podanej nazwie.

- **`password noweHaslo`**

  Zmienia hasło podanego użytkownika.

- **`ping`**

  Sprawdza działanie podanego hosta MySQL.

- **`status`**

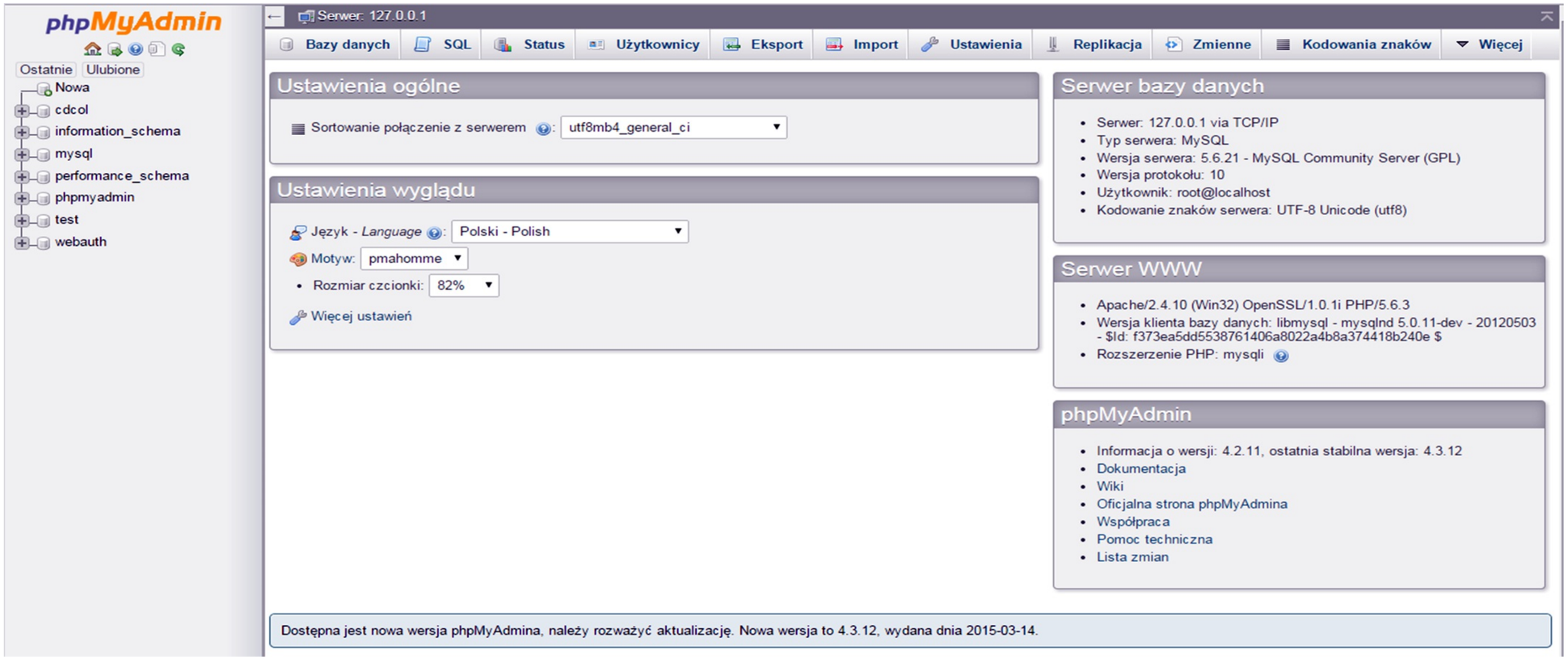
  Wyświetla statystyki naszej bazy danych.

Dostęp do MySQL – panel phpMyAdmin

- Jest to w pełni funkcjonalny, napisany w PHP, klient graficzny do zarządzania bazą danych.
- Często jedyny sposób dostępu do baz danych na wykupionych serwerach.
- Program jest darmowy.
- Można go pobrać ze strony http://www.phpmyadmin.net

Dostęp do MySQL – panel phpMyAdmin

Okno programu phpMyAdmin:



Dostęp do MySQL – MySQL Workbench

- Jest to w pełni funkcjonalny, darmowy klient graficzny do zarządzania bazą danych.
- Aby go zainstalować w systemie Ubuntu należy napisać w konsoli:

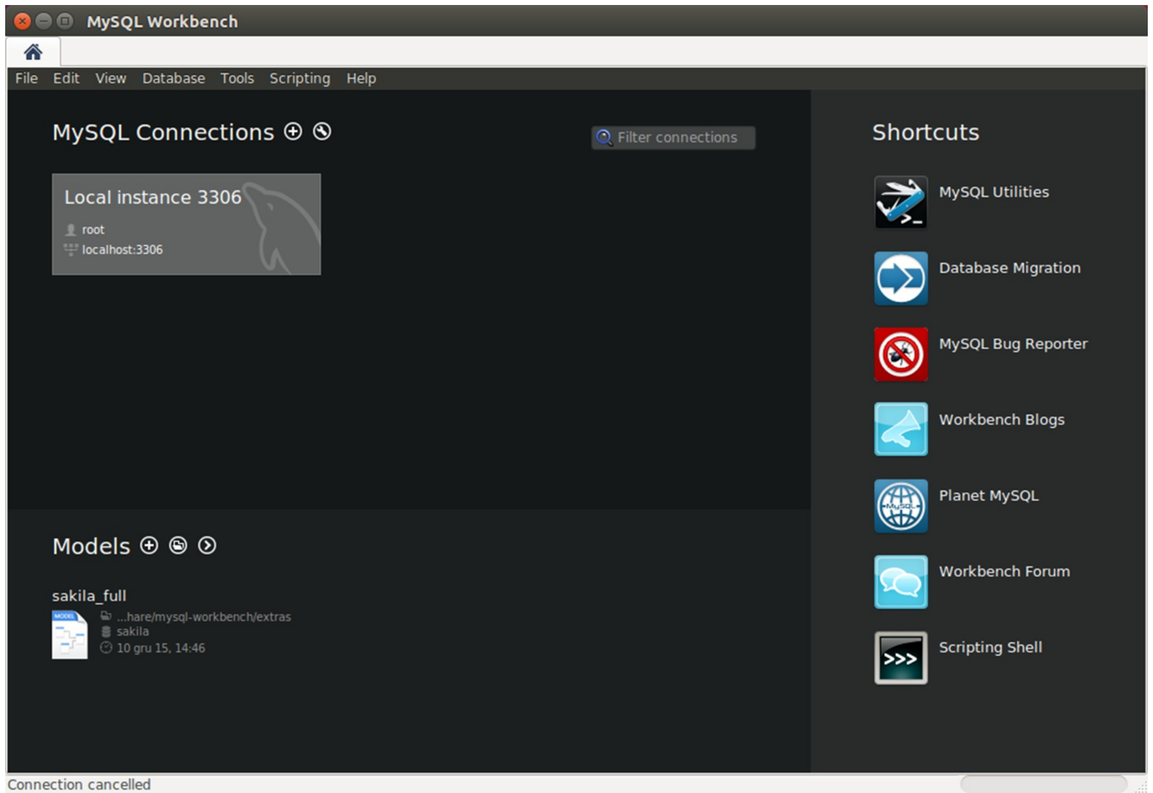
```
sudo apt-get install mysql-workbench
```

- Aby zainstalować program w innych systemach operacyjnych, należy wejść na poniższą stronę, wybrać system z listy i ściągnąć program (dla Windows najodpowiedniejszą wersją będzie prawdopodobnie MSI 64-bit): https://dev.mysql.com/downloads/workbench

  Instalator przeprowadzi przez wszystkie etapy instalacji oprogramowania.

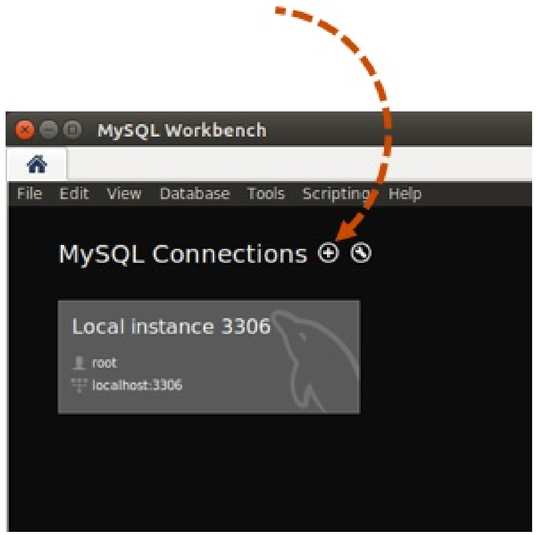
Dostęp do MySQL – MySQL Workbench

Okno MySQL Workbench:

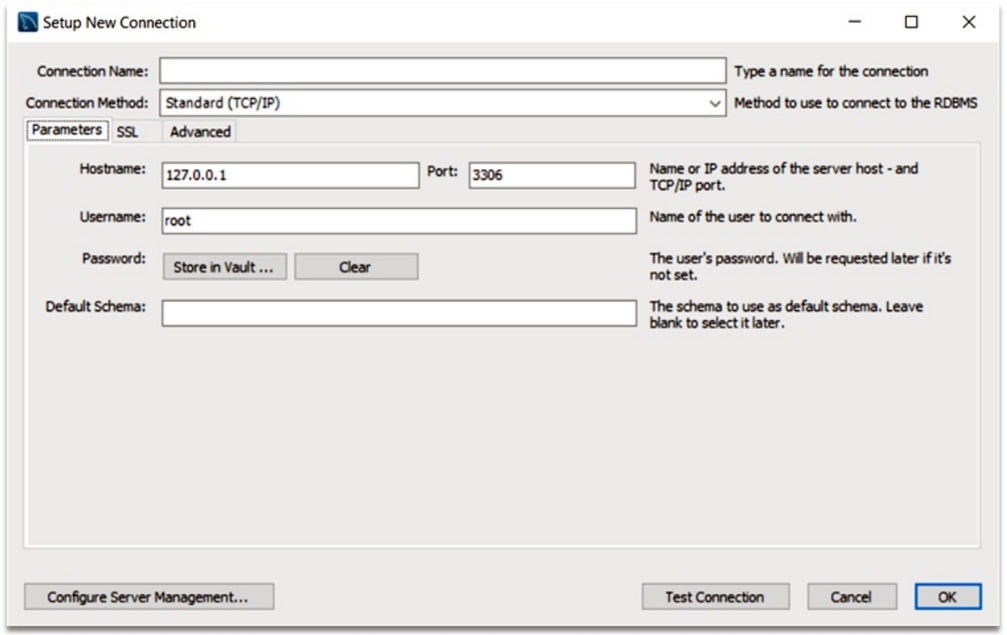


Dostęp do MySQL – MySQL Workbench

Aby skonfigurować aplikację, należy dodać nowy serwer poprzez kliknięcie w poniższą ikonę na ekranie:



Kliknięcie otworzy okienko konfiguracyjne:



Dostęp do MySQL – MySQL Workbench

- W pole **Connection name** należy wpisać nazwę serwera. Nie jest ono wymagane.
- W pole **Hostname** należy wpisać adres serwera bazy danych. Jeśli serwer znajduje się na tym samym komputerze (czyli tak, jak w przypadku naszego kursu), zostaw **127.0.0.1**.
- W pole **Username** należy wpisać nazwę użytkownika. Osoby, które korzystały ze skryptu instalacyjnego wpisują **root**.
- Kliknij przycisk **Store in Vault**, po czym wpisz hasło użytkownika **root**. Osoby, które korzystały ze skryptu instalacyjnego wpisują **coderslab**.
- Kliknij **Test Connection**, by sprawdzić, czy wszystko jest w porządku, potem **OK**.
- W razie problemów poproś wykładowcę o pomoc.

Tworzenie backupu bazy danych

Bardzo ważne jest tworzenie backupów baz danych, aby w przypadku awarii serwera nie utracić przechowywanych w nich danych.

Do tworzenia backupu bazy danych służy narzędzie **`mysqldump`**.

Podstawowe użycie tego narzędzia wygląda następująco:

```
mysqldump -u [user_name] -p [database_name] > [file].sql
```

Powyższa komenda utworzy plik o podanej nazwie, w którym będzie pełny zapis naszej bazy danych (w postaci zapytań **MySQL**).

Wczytywanie backupu

Aby wczytać backup, należy po prostu przesłać wszystkie znajdujące się w nim informacje (w postaci zapytań) do silnika **MySQL**.

Robimy to przy pomocy komendy:

```
mysql -u [user_name] -p [database_name] < [file].sql
```

Baza danych o odpowiedniej nazwie musi być wcześniej stworzona i musi być pusta!

Można też odpowiednio skonfigurować backup, że jeśli baza nie istnieje, to ma zostać stworzona.

Trochę teorii o MySQL

Silniki baz danych

Bazy danych z czasem rozrosły się, wspomagając dużo różnych rozwiązań i technologii. Powstało zatem wiele silników wspierających MySQL.

Pod pojęciem "silniki" możemy rozumieć w tym kontekście moduły, odpowiedzialne za tworzenie i zarządzanie tabelami z danymi. Cechują je różne zastosowania i funkcjonalności.

Najpopularniejsze silniki MySQL

- MyISAM
- IBMDB2I
- InnoDB
- MEMORY
- MERGE
- FEDERATED
- ARCHIVE
- CSV
- EXAMPLE
- BLACKHOLE

Opis silników dla MySQL

MyISAM

Podstawowy silnik dla MySQL. Niezależny od systemu operacyjnego (tabele są przenaszalne). Szybki i wydajny, nie wspiera transakcji. Polecany przy częstych operacjach **`SELECT`** i **`INSERT`**.

MEMORY

Silnik trzymający wszystkie dane w pamięci. Szybki, ale tracący wszystkie dane, jeżeli proces MySQL nie zostanie poprawnie zamknięty. Nie wspomaga wszystkich typów danych.

CSV

Silnik zapamiętujący wszystkie dane w postaci plików **`.csv`**. Nie wspomaga indeksów.

Opis silników dla MySQL

InnoDB

Silnik wydany na licencji open source. Główną zaletą jest pełne wsparcie transakcji (wykonywania zapytań do bazy jako zależny od siebie zbiór zapytań). Powinien być wybierany, jeżeli w bazie będzie dużo operacji typu **`UPDATE`** lub jeżeli są potrzebne transakcje.

EXAMPLE

Template pozwalające tworzyć nowe silniki.

BLACKHOLE

Silnik identyczny w działaniu do MyISAM, nie przetrzymuje jednak żadnych danych. Używany do przeprowadzania testów.

MySQL i Java

Sterowniki JDBC

`JDBC (Java Database Connectivity)` – to wbudowana w Javę technologia umożliwiająca połączenie z bazami danych.

Sterownik JDBC to biblioteka udostępniana w postaci plików **jar**, której zadaniem jest tłumaczenie odwołań z poziomu języka Java na właściwe zapytania szczegółowe dla konkretnego systemu bazodanowego.

Sterowniki tworzone są przez autorów systemów baz danych i muszą być zgodne ze specyfikacją określoną przez twórców Javy.

Dzięki takiemu podejściu możemy w niezauważalny dla naszej aplikacji sposób wymienić bazę danych na inną.

Dostępne są one dla wszystkich najpopularniejszych baz danych, m.in.:

- **MySQL**,
- **Oracle**,
- **PostgreSQL**,
- **Microsoft SQL**.

JDBC

Do połączenia z bazami danych w Javie potrzebujemy odpowiedniego sterownika. W naszym przypadku będzie to sterownik do MySQL dla języka Java: **`mysql-connector-java-5.1.40-bin.jar`** (numer może się różnić).

Jest to sterownik tworzony przez producenta bazy danych – firmę Oracle.

Aby go dołączyć do naszego projektu, możemy go pobrać z oficjalnej strony:

https://dev.mysql.com/downloads/connector/j/

Pobrane archiwum w formacie **`*.tar`**, lub **`*.zip`**, należy rozpakować a następnie dołączyć sterownik (plik z rozszerzeniem **`*.jar`**) do naszego projektu.

Dołączania bibliotek uczyliśmy się już na przykładzie biblioteki **`commons-lang3`**.

Praca z bazą danych

Łączenie się z bazą

Pierwszym krokiem do używania bazy danych jest podłączenie się do niej. W tym celu należy (zgodnie z poniższym schematem) utworzyć połączenie, które jest obiektem klasy **`Connection`**:

```
Connection nazwa_połączenia = DriverManager.getConnection(url, user, password);
```

Praca z bazą danych

Łączenie się z bazą

Pierwszym krokiem do używania bazy danych jest podłączenie się do niej. W tym celu należy (zgodnie z poniższym schematem) utworzyć połączenie, które jest obiektem klasy **`Connection`**:

```
Connection nazwa_połączenia = DriverManager.getConnection(url, user, password);
```

`url` – adres połączenia do bazy danych

Praca z bazą danych

Łączenie się z bazą

Pierwszym krokiem do używania bazy danych jest podłączenie się do niej. W tym celu należy (zgodnie z poniższym schematem) utworzyć połączenie, które jest obiektem klasy **`Connection`**:

```
Connection nazwa_połączenia = DriverManager.getConnection(url, user, password);
```

`url` – adres połączenia do bazy danych

`user` – użytkownik bazy danych

Praca z bazą danych

Łączenie się z bazą

Pierwszym krokiem do używania bazy danych jest podłączenie się do niej. W tym celu należy (zgodnie z poniższym schematem) utworzyć połączenie, które jest obiektem klasy **`Connection`**:

```
Connection nazwa_połączenia = DriverManager.getConnection(url, user, password);
```

`url` – adres połączenia do bazy danych

`user` – użytkownik bazy danych

`password` – hasło do bazy danych

Praca z bazą danych

Przykład:

```
Connection conn = DriverManager.getConnection(
        "jdbc:mysql://localhost:3306/my_db?useSSL=false&characterEncoding=utf8",
        "root",
        "coderslab");
```

Praca z bazą danych

Przykład:

```
Connection conn = DriverManager.getConnection(
        "jdbc:mysql://localhost:3306/my_db?useSSL=false&characterEncoding=utf8",
        "root",
        "coderslab");
```

jdbc:mysql: – określamy, że chodzi nam o połączenie do bazy MySQL,

Praca z bazą danych

Przykład:

```
Connection conn = DriverManager.getConnection(
        "jdbc:mysql://localhost:3306/my_db?useSSL=false&characterEncoding=utf8",
        "root",
        "coderslab");
```

`jdbc:mysql:` – określamy, że chodzi nam o połączenie do bazy MySQL,

`localhost` – adres serwera bazy danych,

Praca z bazą danych

Przykład:

```
Connection conn = DriverManager.getConnection(
        "jdbc:mysql://localhost:3306/my_db?useSSL=false&characterEncoding=utf8",
        "root",
        "coderslab");
```

`jdbc:mysql:` – określamy, że chodzi nam o połączenie do bazy MySQL,

`localhost` – adres serwera bazy danych,

`3306` – port serwera bazy danych (**`3306`** to standardowy port dla MySQL),

Praca z bazą danych

Przykład:

```
Connection conn = DriverManager.getConnection(
        "jdbc:mysql://localhost:3306/my_db?useSSL=false&characterEncoding=utf8",
        "root",
        "coderslab");
```

jdbc:mysql: – określamy, że chodzi nam o połączenie do bazy MySQL,

localhost – adres serwera bazy danych,

3306 – port serwera bazy danych (**3306** to standardowy port dla MySQL),

my_db – nazwa bazy danych,

Praca z bazą danych

Przykład:

```
Connection conn = DriverManager.getConnection(
        "jdbc:mysql://localhost:3306/my_db?useSSL=false&characterEncoding=utf8",
        "root",
        "coderslab");
```

jdbc:mysql: – określamy, że chodzi nam o połączenie do bazy MySQL,

localhost – adres serwera bazy danych,

3306 – port serwera bazy danych (**3306** to standardowy port dla MySQL),

my_db – nazwa bazy danych,

useSSL=false – szyfrowanie połączenia (omówione na dalszych slajdach),

Praca z bazą danych

Przykład:

```
Connection conn = DriverManager.getConnection(
        "jdbc:mysql://localhost:3306/my_db?useSSL=false&characterEncoding=utf8",
        "root",
        "coderslab");
```

jdbc:mysql: – określamy, że chodzi nam o połączenie do bazy MySQL,

localhost – adres serwera bazy danych,

3306 – port serwera bazy danych (**3306** to standardowy port dla MySQL),

my_db – nazwa bazy danych,

useSSL=false – szyfrowanie połączenia (omówione na dalszych slajdach),

characterEncoding=utf8 – ustawienie kodowania,

Praca z bazą danych

Przykład:

```
Connection conn = DriverManager.getConnection(
        "jdbc:mysql://localhost:3306/my_db?useSSL=false&characterEncoding=utf8",
        "root",
        "coderslab");
```

`jdbc:mysql:` – określamy, że chodzi nam o połączenie do bazy MySQL,

`localhost` – adres serwera bazy danych,

`3306` – port serwera bazy danych (**`3306`** to standardowy port dla MySQL),

`my_db` – nazwa bazy danych,

`useSSL=false` – szyfrowanie połączenia (omówione na dalszych slajdach),

`characterEncoding=utf8` – ustawienie kodowania,

`root` – nazwa użytkownika, **`coderslab`** – hasło.

Konstrukcja: try with resources

try with resources – to konstrukcja, która upraszcza zarządzanie zasobami (np. otwarte połączenia, pliki).

Możemy ją stosować dla wszystkich klas implementujących interfejs **AutoCloseable** (więcej na temat interfejsów dowiemy się kolejnych modułach).

Listę klas implementujących ten interfejs znajdziemy pod adresem:

https://docs.oracle.com/javase/8/docs/api/java/lang/AutoCloseable.html

Konstrukcja: try with resources

Konstrukcję **try with resources** tworzymy wg poniższego schematu:

```
try (resource) { }
```

Możemy również zdefiniować więcej niż jeden zasób, oddzielając je znakiem średnika np.:

```
try (resource1; resource2;
resource3) { }
```

Dzięki zastosowaniu tej konstrukcji, zasoby w niej zawarte, zostaną automatycznie zamknięte.

Ręczne zamknięcie zasobu odbywa się przez wywołanie metody **close()**.

Przykład połączenia

Tworzenie nowego połączenia

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

    try (Connection conn = DriverManager.getConnection(
        "jdbc:mysql://localhost:3306/my_db?useSSL=false&characterEncoding=utf8",
        "root", "coderslab")) {

    } catch (SQLException e) {
        e.printStackTrace();
    }
}
```

Przykład połączenia

Tworzenie nowego połączenia

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

    try (Connection conn = DriverManager.getConnection(
        "jdbc:mysql://localhost:3306/my_db?useSSL=false&characterEncoding=utf8",
        "root", "coderslab")) {

    } catch (SQLException e) {
        e.printStackTrace();
    }
}
```

Tworzymy obiekt połączenia korzystając z konstrukcji **try with resources**.

Przykład połączenia

Tworzenie nowego połączenia

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

    try (Connection conn = DriverManager.getConnection(
        "jdbc:mysql://localhost:3306/my_db?useSSL=false&characterEncoding=utf8",
        "root", "coderslab")) {

    } catch (SQLException e) {
        e.printStackTrace();
    }
}
```

Dodajemy obsługę wymaganego wyjątku.

Przykład połączenia

Tworzenie nowego połączenia

Jeżeli nie korzystamy z **try with resources** musimy sami zadbać o zamknięcie połączenia. Wykorzystujemy w tym celu blok **finally** konstrukcji **try – catch**:

```
public static void main(String[] args) {
    Connection conn = null;
    try { conn = DriverManager.getConnection(
            "jdbc:mysql://localhost:3306/my_db?useSSL=false&characterEncoding=utf8",
            "root", "coderslab");
    } catch (SQLException e) {  e.printStackTrace(); }
        finally {
        if (conn != null) {
            try {conn.close(); } catch (SQLException e) { e.printStackTrace();  }
    }}}
```

Przykład połączenia

Tworzenie nowego połączenia

Jeżeli nie korzystamy z **try with resources** musimy sami zadbać o zamknięcie połączenia. Wykorzystujemy w tym celu blok **finally** konstrukcji **try – catch**:

```
public static void main(String[] args) {
    Connection conn = null;
    try { conn = DriverManager.getConnection(
            "jdbc:mysql://localhost:3306/my_db?useSSL=false&characterEncoding=utf8",
            "root", "coderslab");
    } catch (SQLException e) {  e.printStackTrace(); }
        finally {
        if (conn != null) {
            try {conn.close(); } catch (SQLException e) { e.printStackTrace();  }
    }}}
```

conn = DriverManager.getConnection – tworzymy obiekt połączenia.

Przykład połączenia

Tworzenie nowego połączenia

Jeżeli nie korzystamy z **try with resources** musimy sami zadbać o zamknięcie połączenia. Wykorzystujemy w tym celu blok **finally** konstrukcji **try – catch**:

```
public static void main(String[] args) {
    Connection conn = null;
    try { conn = DriverManager.getConnection(
            "jdbc:mysql://localhost:3306/my_db?useSSL=false&characterEncoding=utf8",
            "root", "coderslab");
    } catch (SQLException e) {  e.printStackTrace(); }
        finally {
        if (conn != null) {
            try {conn.close(); } catch (SQLException e) { e.printStackTrace();  }
    }}}
```

Dodajemy obsługę wymaganego wyjątku.

Przykład połączenia

Tworzenie nowego połączenia

Jeżeli nie korzystamy z **try with resources** musimy sami zadbać o zamknięcie połączenia. Wykorzystujemy w tym celu blok **finally** konstrukcji **try – catch**:

```
public static void main(String[] args) {
    Connection conn = null;
    try { conn = DriverManager.getConnection(
            "jdbc:mysql://localhost:3306/my_db?useSSL=false&characterEncoding=utf8",
            "root", "coderslab");
    } catch (SQLException e) {  e.printStackTrace(); }
        finally {
        if (conn != null) {
            try {conn.close(); } catch (SQLException e) { e.printStackTrace();  }
    }}}
```

Dodajemy blok **finally**.

Przykład połączenia

Tworzenie nowego połączenia

Jeżeli nie korzystamy z **try with resources** musimy sami zadbać o zamknięcie połączenia. Wykorzystujemy w tym celu blok **finally** konstrukcji **try – catch**:

```
public static void main(String[] args) {
    Connection conn = null;
    try { conn = DriverManager.getConnection(
            "jdbc:mysql://localhost:3306/my_db?useSSL=false&characterEncoding=utf8",
            "root", "coderslab");
    } catch (SQLException e) {  e.printStackTrace(); }
        finally {
        if (conn != null) {
            try {conn.close(); } catch (SQLException e) { e.printStackTrace();  }
    }}}
```

Sprawdzamy czy połączenie nie jest równe **null** .

Przykład połączenia

Tworzenie nowego połączenia

Jeżeli nie korzystamy z **try with resources** musimy sami zadbać o zamknięcie połączenia. Wykorzystujemy w tym celu blok **finally** konstrukcji **try – catch**:

```
public static void main(String[] args) {
    Connection conn = null;
    try { conn = DriverManager.getConnection(
            "jdbc:mysql://localhost:3306/my_db?useSSL=false&characterEncoding=utf8",
            "root", "coderslab");
    } catch (SQLException e) {  e.printStackTrace(); }
        finally {
        if (conn != null) {
            try {conn.close(); } catch (SQLException e) { e.printStackTrace();  }
    }}}
```

Zamykamy połączenie przy pomocy metody **close()** dodając obsługę wymaganego wyjątku.

Możliwe błędy/ostrzeżenia

- Nie załączyliśmy poprawnie do naszego projektu sterownika bazy danych.

```
Got an exception!
No suitable driver found for
jdbc:mysql://localhost:3306/my_db
```

- Podczas nawiązywania połączenia otrzymujemy ostrzeżenie:

```
Fri Feb 03 05:54:12 PST 2017 WARN: Establishing SSL connection
without server's identity verification is not recommended.
According to MySQL 5.5.45+, 5.6.26+ and 5.7.6+ requirements SSL
connection must be established by default if explicit option
isn't set. For compliance with existing applications not using
SSL the verifyServerCertificate property is set to 'false'. You
need either to explicitly disable SSL by setting useSSL=false, or
set useSSL=true and provide truststore for server certificate
verification.
```

Możliwe błędy/ostrzeżenia

Otrzymaliśmy ostrzeżenie informujące nas, że powinniśmy korzystać z szyfrowanego połączenia z naszą bazą danych. Dla lokalnego połączenia nie jest to konieczne.

Aby ostrzeżenie zniknęło, należy zmodyfikować adres połączenia, dołączając za nazwą bazy odpowiednią adnotację:

```
"jdbc:mysql://localhost:3306/my_db?useSSL=false"
```

Tworzenie bazy danych

Słowa zarezerwowane

Tworząc tabele i kolumny należy unikać słów zarezerwowanych.

- Dla **MySQL** ich zestaw znajdziemy pod adresem:

  https://dev.mysql.com/doc/refman/5.7/en/keywords.html

- Dla **PostgreSQL** ich zestaw znajdziemy pod adresem:

  https://www.postgresql.org/docs/8.1/static/sql-keywords-appendix.html

  Możemy również skorzystać z serwisów udostępniających możliwość sprawdzania czy dane słowo jest zarezerwowane. Przykładowy serwis:

  http://www.petefreitag.com/tools/sql_reserved_words_checker/

Tworzenie nowej bazy danych

Bazę tworzymy poprzez zapytanie SQL:

`CREATE DATABASE <nazwa_nowej_bazy>;`

Musimy pamiętać, że jeżeli zapytanie SQL wywołujemy w konsoli – musi kończyć się średnikiem. W kodzie Javy nie jest to konieczne.

Użytkownik MySQL, który chce utworzyć bazę danych – musi mieć uprawnienia do jej tworzenia.

Często jeden projekt może mieć kilka baz danych, np.:

- do testów,
- dla developmentu,
- produkcyjną.

Ustawienie kodowania

Na etapie tworzenia bazy danych możemy dopisać instrukcję pozwalającą ustawić odpowiednie kodowanie dla znaków.

```
CREATE DATABASE <nazwa_nowej_bazy>
CHARACTER SET utf8mb4
COLLATE utf8mb4_unicode_ci;
```

Ustawienie kodowania

Na etapie tworzenia bazy danych możemy dopisać instrukcję pozwalającą ustawić odpowiednie kodowanie dla znaków.

```
CREATE DATABASE <nazwa_nowej_bazy>
CHARACTER SET utf8mb4
COLLATE utf8mb4_unicode_ci;
```

Ustawiamy kodowanie na utf8mb4 (czterobajtowy znak unicode).

Ustawienie kodowania

Na etapie tworzenia bazy danych możemy dopisać instrukcję pozwalającą ustawić odpowiednie kodowanie dla znaków.

```
CREATE DATABASE <nazwa_nowej_bazy>
CHARACTER SET utf8mb4
COLLATE utf8mb4_unicode_ci;
```

`collate utf8mb4` – definiujemy reguły porównywania i sortowania dla określonych znaków (reguły porównywania i sortowania mogą się różnić w zależności od języka),
`ci` – oznacza **`case insensitive,`**

Tabele w bazie danych

Każda baza ma w swojej strukturze tabele.

Tabela w bazie to podzbiór przechowujący dane związane ze sobą. Jeśli naszą bazą danych jest sklep, to przykładowo może posiadać następujące tabele:

- tabela z danymi pracowników sklepu,
- tabela zamówień w sklepie,
- tabela produktów w sklepie,
- tabela z aktualnymi promocjami.

Każda tabela składa się z kolumn i wierszy.

Kolumna przechowuje dane związane z rekordem, dane te są odpowiedniego typu, np.:

- **dla użytkownika**:

  imię, nazwisko, mail, adres IP, login, hasło;

- **dla zamówienia**:

  numer, suma, data złożenia, data opłacenia, data wysyłki.

Wiersz to pojedynczy rekord przechowywany w tabeli.

Typy danych w MySQL

Typy danych, które trzymamy w tabelach, możemy podzielić na trzy główne grupy:

- zmienne liczbowe,
- zmienne daty i czasu,
- napisy.

Każdy typ danych ma określony zakres.

Tworzac kolumnę, powinniśmy zadeklarować dla niej taki typ, aby był odpowienio dopasowany do danych, które mają znaleźć się tej kolumnie.

Jeśli będziemy korzystać z liczb od **`0–100`** skorzystajmy z **`TINYINT`** zamiast np. **`INT`**, aby zająć mniej miejsca.

Typy danych w MySQL – liczby

- **INT** – podstawowa zmienna liczbowa. Przechowuje liczby całkowite w przedziale: od **-2 147 483 648** do **2 147 483 647**. Możemy nie korzystać z liczb ujemnych, wówczas przedział ten wynosi od **0** do **4 294 967 295**.

- **TINYINT** – przechowuje liczby całkowite z przedziału od **-128** do **127**. Jeśli nie będziemy korzystać z liczb ujemnych, wówczas przedział ten wynosi od **0** do **255**.

- **SMALLINT** – przechowuje liczby całkowite w przedziale: od **-32 768** do **32 767**. Jeśli wykluczymy liczby ujemne, wówczas przedział ten wynosi od **0** do **65 636**.

- **MEDIUMINT** – przechowuje liczby całkowite od **-8 388 608** do **8 388 607**. Gdy nie korzystamy z liczb ujemnych, wówczas przedział ten wynosi od **0** do **16 777 215**.

Typy danych w MySQL – liczby cd.

- **`BIGINT`** – przechowuje liczby całkowite w przedziale: od **`-9 223 372 036 854 775 808`** do **`9 223 372 036 854 775 807`**. Możemy nie korzystać z liczb ujemnych, wówczas przedział ten wynosi od **`0`** do **`18 446 744 073 709 551 615`**.

- **`FLOAT(M,D)`** – zmienna reprezentująca liczbę zmiennoprzecinkową. **`M`** – liczba wyświetlanych cyfr, **`D`** – liczba cyfr po przecinku (maksymalnie 24 miejsca). Np. **`FLOAT(5,3)`** oznacza pięć cyfr w liczbie, z tego trzy po przecinku, np. **`34.023`**.

- **`DOUBLE(M,D)`** – liczba zmiennoprzecinkowa o większej dokładności. Może trzymać do 53 miejsc po przecinku. **`REAL`** jest synonimem typu **`DOUBLE`**.

- **`DECIMAL(M,D)`** – liczba zmiennoprzecinkowa, do której nie używamy kompresji, przechowuje dokładną liczbę. **`NUMERIC`** jest synonimem **`DECIMAL`**. Typu **`DECIMAL`** powinniśmy używać do przechowywania np. cen lub kursów walut, nie używamy nigdy do tego **`FLOAT`**, który przechowuje wartość przybliżoną.

Typy danych w MySQL – data i czas

- **`DATE`** – data w formacie **`YYYY-MM-DD`**. Może trzymać daty od **`1000-01-01`** do **`9999-12-31`**.

- **`DATETIME`** – data w formacie **`YYYY-MM-DD HH:MI:SS`**. może trzymać daty od **`1000-01-01 00:00:00`** do **`9999-12-31 23:59:59`**.

- **`TIMESTAMP`** – data przechowywana w postaci liczby milisekund, które upłynęły od 1 stycznia 1970 roku do podanej daty. Podstawowy format **`YYYY-MM-DD HH:MI:SS`**. Może trzymać daty między **`1970-01-01`** a **`2038-01-09`**.

- **`TIME`** – trzyma czas w formacie **`HH:MI:SS`**.

- **`YEAR`** – do wersji MySQL 5.7.5 trzymał rok w formacie dwu- i czterocyfrowym. W obecnej wersji wspomagany jest już tylko format czterocyfowy.

Typy danych w MySQL – napisy

- **`CHAR(M)`** – napis mający z góry określoną liczbę znaków, parametr **`M`** przyjmuje wartość między **`0`** a **`255`**. Wypełniany spacjami, jeżeli napis będzie krótszy. Spacje są automatycznie usuwane przy pobieraniu danych z bazy. Zajmuje stałą ilość pamięci dla danej długości.

- **`VARCHAR(M)`** – napis o zmiennej liczbie znaków, nie większej jednak niż podany parametr **`M`** (o wartości od **`0`** do **`65536`**). Zajmuje zmienną ilość pamięci dla danej długości.

Typy danych w MySQL – napisy

Poniższa tabela przedstawia różnicę w przechowywaniu danych **CHAR** i **VARCHAR**.

Value	CHAR(4)	Storage Required	VARCHAR(4)	Storage Required
''	'----'	4 bytes	''	1 byte
'ab'	'ab--'	4 bytes	'ab'	3 bytes
'abcd'	'abcd'	4 bytes	'abcd'	5 bytes
'abcdefgh'	'abcd'	4 bytes	'abcd'	5 bytes

Typy danych w MySQL – napisy

- **BLOB (Binary Large Object)** lub **TEXT** – pole zawierające maksymalnie **65 535** znaków. **BLOB** od zmiennej **TEXT** różni się tym, że porównanie zmiennej typu **BLOB** nie jest wrażliwe na wielkość znaków.

- **TINYBLOB** lub **TINYTEXT** – zmienna **BLOB** lub **TEXT**, ale o maksymalnej długości **255** znaków.

- **MEDIUMBLOB** lub **MEDIUMTEXT** – zmienna **BLOB** lub **TEXT**, ale o maksymalnej długości **16 777 215** znaków.

- **LONGBLOB** lub **LONGTEXT** – zmienna **BLOB** lub **TEXT**, ale o maksymalnej długości **4 294 967 295** znaków.

Atrybuty danych w MySQL

Atrybuty dodawane są do kolumny. Uzupełniają one typy danych o dodatkowe założenia.

Nałożenie odpowiednich atrybutów może całkowicie zmienić zastosowanie danej kolumny w tabeli naszych danych.

Najczęściej używane atrybuty:

- **`PRIMARY KEY`** – klucz główny. Atrybut stosowany do wskazania kolumny, która będzie jednoznacznie identyfikowała każdy wpis. Zazwyczaj do stworzenia klucza głównego używamy zmiennej typu **`INT`** z włączoną opcją **`AUTO_INCREMENT`**.

Klucz główny nie powinien być zmieniany po utworzeniu. Może nam to zniszczyć układ całej bazy.

Atrybuty danych w MySQL

- **`UNSIGNED`** – stosowany przy zmiennych liczbowych. Oznacza, że chcemy korzystać jedynie z liczb dodatnich danego typu liczbowego, przez co zwiększamy jego zakres.
- **`ZEROFILL`** – stosowany przy zmiennych liczbowych. Powoduje dopełnienie liczby zerami poprzedzającymi.
- **`CHARACTER SET`** – stosowany przy napisach. Ustawia odpowiednie kodowanie dla napisów. Powinno się go podawać dla każdej kolumny przechowującej tekst.
- **`BINARY`** – używany przy zmiennych typu **`CHAR`** lub **`VARCHAR`**. Powoduje, że sortowanie jest case-sensitive.

Atrybuty danych w MySQL

- **`DEFAULT`** – ustawia domyślną wartość, która zostanie wpisana do klumny, jeśli nie podamy wartości.

- **`NULL / NOT NULL`** – pozwala (lub nie pozwala) na brak danych w tej kolumnę.

- **`AUTO_INCREMENT`** – stosowany przy zmiennych liczbowych. Powoduje, że wartość w tej kolumnie zwiększa się domyślnie o jeden przy każdym wpisie. Nie ma konieczności podawania jej wartości, tabela automatycznie nada kolejną wartość.

  Więcej informacji możesz znaleźć na stronie: https://dev.mysql.com/doc/refman/8.0/en/replication-options-master.html#sysvar_auto_increment_increment

Tworzenie nowej tabeli

Znamy już typy danych kolumn w tabeli, możemy zatem utworzyć nową tabelę za pomocą następującego zapytania SQL:

```
CREATE TABLE nazwa_tabeli
(
nazwa_kolumny_1 typ_danych(size) [atrybuty],
nazwa_kolumny_2 typ_danych(size) [atrybuty],
nazwa_kolumny_3 typ_danych(size) [atrybuty],
....
);
```

Przyjęte jest, że pierwsza kolumna takiej tabeli to `ID` – najczęściej `INT` z atrybutem `AUTO_INCREMENT`.

Tworzenie nowej tabeli

Przykład

```
CREATE TABLE users
(
user_id int AUTO_INCREMENT,
user_name varchar(255),
user_email varchar(255) UNIQUE,
PRIMARY KEY(user_id)
);
```

Tworzenie nowej tabeli

Przykład

```
CREATE TABLE users
(
user_id int AUTO_INCREMENT,
user_name varchar(255),
user_email varchar(255) UNIQUE,
PRIMARY KEY(user_id)
);
```

user_id będzie kluczem głównym w naszej tabeli, wartość będzie unikalna (**PRIMARY KEY**). Ten numer będzie identyfikował każdy wpis, jego wartość będzie zwiększana automatycznie o jeden (**AUTO_INCREMENT**).

Tworzenie nowej tabeli

Przykład

```
CREATE TABLE users
(
user_id int AUTO_INCREMENT,
user_name varchar(255),
user_email varchar(255) UNIQUE,
PRIMARY KEY(user_id)
);
```

user_id będzie kluczem głównym w naszej tabeli, wartość będzie unikalna (**PRIMARY KEY**). Ten numer będzie identyfikował każdy wpis, jego wartość będzie zwiększana automatycznie o jeden (**AUTO_INCREMENT**).

user_email ma atrybut **UNIQUE** – chcemy, aby w naszym systemie mógł być tylko jeden użytkownik z danym mailem. Tabela nie zezwoli na zapisanie dwóch rekordów z takim samym polem **user_email**.

Praca z połączeniem

Komunikacja z bazą danych odbywa się poprzez wykonywanie zapytań (**queries**).

Zapytania do bazy będziemy wykonywać, używając do tego celu obiektu **Statement**.

```
Statement stmt = conn.createStatement();
```

Będziemy wykonywać metody **executeQuery** oraz **executeUpdate**:

```
executeQuery("zapytanie");
```

Jako wynik metody otrzymamy obiekt **ResultSet**, który będzie przechowywał wynik.

```
executeUpdate("zapytanie");
```

Jako wynik otrzymamy wartość typu **int**, która oznacza liczbę zmodyfikowanych wierszy.

Tworzenie nowej tabeli w Javie

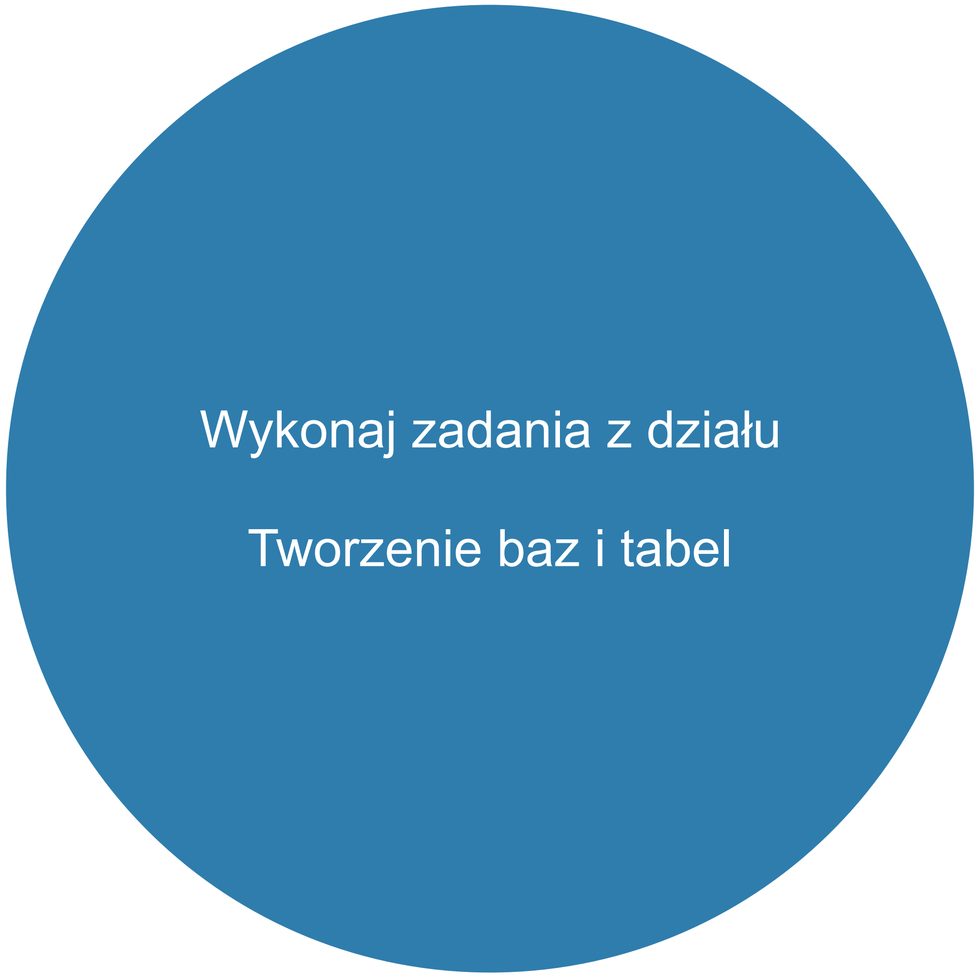
```
String sql = "CREATE TABLE users (user_id int AUTO_INCREMENT,"
        + " user_name varchar(255),"
        + " user_email varchar(255) UNIQUE, "
        + " PRIMARY KEY(user_id))";
try (Connection conn =
        DriverManager.getConnection("jdbc:mysql://localhost:3306/my_database"
                + "?useSSL=false",
        "root", "root"); Statement stat = conn.createStatement()) {
    stat.executeUpdate(sql);
} catch (SQLException e) {
    e.printStackTrace();
}
```

Tworzenie nowej tabeli w Javie

```
String sql = "CREATE TABLE users (user_id int AUTO_INCREMENT,"
        + " user_name varchar(255),"
        + " user_email varchar(255) UNIQUE, "
        + " PRIMARY KEY(user_id))";
try (Connection conn =
        DriverManager.getConnection("jdbc:mysql://localhost:3306/my_database"
                + "?useSSL=false",
        "root", "root"); Statement stat = conn.createStatement()) {
    stat.executeUpdate(sql);
} catch (SQLException e) {
    e.printStackTrace();
}
```

Wykonujemy zapytanie.

Zadania



Dodawanie danych do tabeli

Dodawanie elementów do tabeli

Dane do tabeli dodajemy za pomocą zapytania **INSERT INTO**:

```
INSERT INTO table_name(col_name_1, col_name_2, col_name_3, …)
VALUES (value1, value2, value3, ...);
```

Jeżeli po nazwie tabeli nie podamy nazw kolumn, dane będą zapisywane do kolejnych kolumn tabeli (zgodnie z jej definicją).

Dodawanie elementów do tabeli

```
INSERT INTO users VALUES (10, "Jacek", "jacek@gmail.com");
Query OK, 1 row affected (0.06 sec)
INSERT INTO users VALUES ("Wojtek", "wojtek@gmail.com");
ERROR 1136 (21S01): Column count doesnt match value count at row 1
INSERT INTO users(user_name, user_email) VALUES("Wojtek", "wojtek@gmail.com");
Query OK, 1 row affected (0.06 sec)
```

Dodawanie elementów do tabeli

```
INSERT INTO users VALUES (10, "Jacek", "jacek@gmail.com");
Query OK, 1 row affected (0.06 sec)
INSERT INTO users VALUES ("Wojtek", "wojtek@gmail.com");
ERROR 1136 (21S01): Column count doesnt match value count at row 1
INSERT INTO users(user_name, user_email) VALUES("Wojtek", "wojtek@gmail.com");
Query OK, 1 row affected (0.06 sec)
```

Błąd spowodowany tym, że liczba kolumn (3) nie jest równa liczbie przekazanych danych (2).

Dodawanie elementów do tabeli w Javie

```
try (Connection conn =
    DriverManager.getConnection("jdbc:mysql://localhost:3306/my_database"
                                + "?useSSL=false&characterEncoding=utf8",
                                "root", "coderslab");
    Statement stat = conn.createStatement()) {
    String name = "user1";
    String email = "user1@coderslab.pl";
    String sql = "INSERT INTO users(user_name, user_email)
                    VALUES(" + "\"" + name + "\"," + "\"" + email + "\");";
    stat.executeUpdate(sql);
} catch (SQLException e) {
        e.printStackTrace();
}
```

Dodawanie elementów do tabeli w Javie

```
try (Connection conn =
    DriverManager.getConnection("jdbc:mysql://localhost:3306/my_database"
                                + "?useSSL=false&characterEncoding=utf8",
                                "root", "coderslab");
    Statement stat = conn.createStatement()) {
    String name = "user1";
    String email = "user1@coderslab.pl";
    String sql = "INSERT INTO users(user_name, user_email)
                    VALUES(" + "\"" + name + "\"," + "\"" + email + "\");";
    stat.executeUpdate(sql);
} catch (SQLException e) {
        e.printStackTrace();
}
```

Łączenie w zapytaniu parametrów, które mogą być pobrane np. z konsoli, jest niebezpieczne ze względu na możliwość podania wartości, która mogłaby zniszczyć naszą bazę danych. Za chwilę dowiemy się, co to znaczy i jak się przed tym uchronić.

SQL Injection

Łączenie parametrów w zapytaniu SQL jest mało wygodne ze względu na zagnieżdżone znaki cudzysłowów:

```
String sql = "INSERT INTO users(user_name, user_email) VALUES(" + name +
             + "," + email + ");";
```

A dodatkowo może być też niebezpieczne, gdy dane pobieramy od użytkownika.

Może nastąpić wstrzyknięcie do naszego zapytania dodatkowego kodu, który wykona niepożądane manipulacje na naszym kodzie. Takie działanie nazywamy **SQL Injection**.

SQL Injection to bardzo częsty sposób ataku na bazy danych.

SQL Injection

Przeanalizujmy poniższy kod:

```
System.out.print("Wpisz szukaną frazę:");
Scanner scan = new Scanner(System.in);
String userName = scan.nextLine();
String sql = "SELECT * FROM users WHERE user_name=" + userName + ";";
Statement stat = conn.createStatement();
stat.executeQuery(sql);
conn.close();
```

SQL Injection

Przeanalizujmy poniższy kod:

```
System.out.print("Wpisz szukaną frazę:");
Scanner scan = new Scanner(System.in);
String userName = scan.nextLine();
String sql = "SELECT * FROM users WHERE user_name=" + userName + ";";
Statement stat = conn.createStatement();
stat.executeQuery(sql);
conn.close();
```

Pobieramy dane od użytkownika.

SQL Injection

Przeanalizujmy poniższy kod:

```
System.out.print("Wpisz szukaną frazę:");
Scanner scan = new Scanner(System.in);
String userName = scan.nextLine();
String sql = "SELECT * FROM users WHERE user_name=" + userName + ";";
Statement stat = conn.createStatement();
stat.executeQuery(sql);
conn.close();
```

Pobieramy dane od użytkownika.

Wstawiamy pobrane dane do zapytania.

SQL Injection

Co się stanie, jeżeli ktoś wpisał do naszego formularza taką wartość:

```
Paul; DROP TABLE users;
```

Nasze zapytanie SQL będzie wyglądać następująco:

```
SELECT * FROM users WHERE name="Paul";
DROP TABLE users;
```

Zapytanie może być również zakończone znakiem **--**, czyli znakiem rozpoczynającym komentarz, spowoduje to pominięcie dalszej części zapytania, jeżeli ono istnieje.

```
Paul; DROP TABLE users;--
```

SQL Injection

Co się stanie, jeżeli ktoś wpisał do naszego formularza taką wartość:

```
Paul; DROP TABLE users;
```

Nasze zapytanie SQL będzie wyglądać następująco:

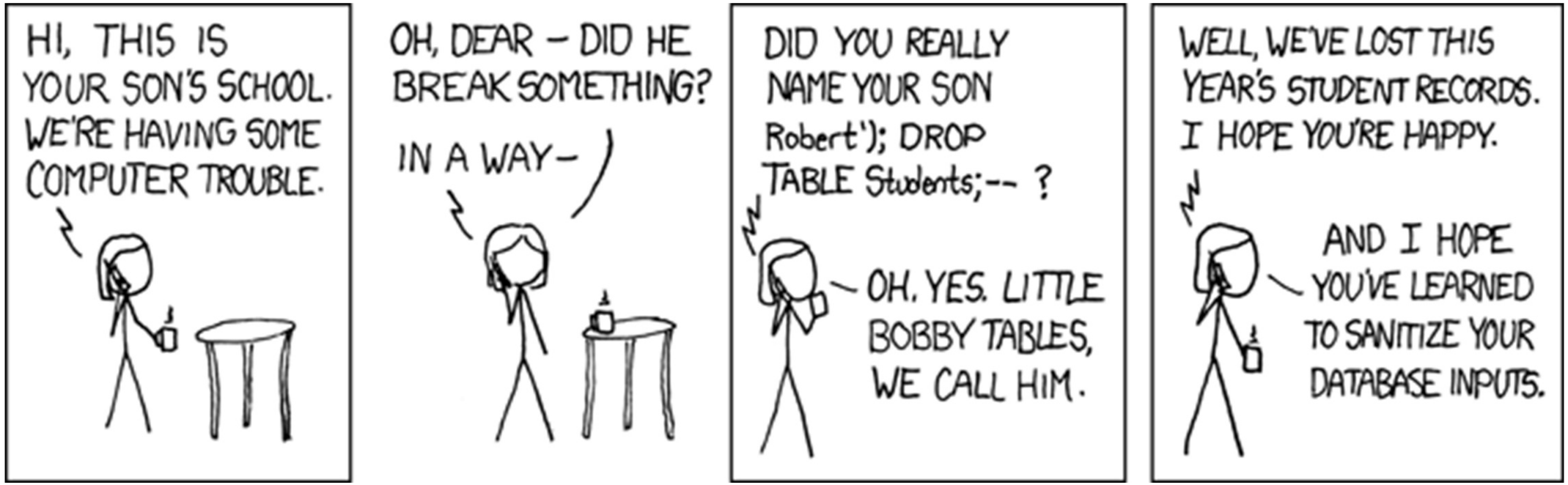
```
SELECT * FROM users WHERE name="Paul";
DROP TABLE users;
```

Usunie tabelę **`users`**.

Zapytanie może być również zakończone znakiem **--**, czyli znakiem rozpoczynającym komentarz, spowoduje to pominięcie dalszej części zapytania, jeżeli ono istnieje.

```
Paul; DROP TABLE users;--
```

SQL Injection



Źródło: https://xkcd.com/327

PreparedStatement

`PreparedStatement` to klasa Javy, dzięki której możemy przygotować szablon zapytania SQL. Następnie taki szablon wypełniamy odpowiednimi danymi i uruchamiamy. Dane przekazane do szablonu są automatycznie filtrowane.

Zalety **`PreparedStatement`**:

- czytelna forma zapisu,
- dane przesyłane do bazy są **odporne na ataki SQL Injection**.

PreparedStatement – przygotowanie

Obiekt **`PreparedStatement`** tworzymy przez użycie metody **`prepareStatement()`** na obiekcie **`Connection`**, przekazując jako argument zapytanie do bazy (szablon):

```
String sql = "SELECT * FROM users WHERE user_name = " + userName + ";"
```

Nasze zapytanie będzie wyglądało następujaco:

```
PreparedStatement preStmt =
    conn.prepareStatement("SELECT * FROM users WHERE user_name = ?");
preStmt.setString(1, userName);
```

PreparedStatement – przygotowanie

Obiekt **`PreparedStatement`** tworzymy przez użycie metody **`prepareStatement()`** na obiekcie **`Connection`**, przekazując jako argument zapytanie do bazy (szablon):

```
String sql = "SELECT * FROM users WHERE user_name = " + userName + ";"
```

Nasze zapytanie będzie wyglądało następujaco:

```
PreparedStatement preStmt =
    conn.prepareStatement("SELECT * FROM users WHERE user_name = ?");
preStmt.setString(1, userName);
```

`user_name = ?` – w miejscu znaku zapytania pojawią się dane, które będziemy wstawiać.

PreparedStatement – przygotowanie

Obiekt **PreparedStatement** tworzymy przez użycie metody **prepareStatement()** na obiekcie **Connection**, przekazując jako argument zapytanie do bazy (szablon):

```
String sql = "SELECT * FROM users WHERE user_name = " + userName + ";"
```

Nasze zapytanie będzie wyglądało następujaco:

```
PreparedStatement preStmt =
    conn.prepareStatement("SELECT * FROM users WHERE user_name = ?");
preStmt.setString(1, userName);
```

user_name = ? – w miejscu znaku zapytania pojawią się dane, które będziemy wstawiać.

Wstawiamy zmienną **userName** jako pierwszy parametr – typu **String**.

PreparedStatement – używanie i wywoływanie

```
PreparedStatement preStmt =
    conn.prepareStatement("SELECT * FROM users WHERE user_name = ?");
preStmt.setString(1, userName);
preStmt.executeQuery();
```

PreparedStatement – używanie i wywoływanie

```
PreparedStatement preStmt =
    conn.prepareStatement("SELECT * FROM users WHERE user_name = ?");
preStmt.setString(1, userName);
preStmt.executeQuery();
```

Tworzymy obiekt klasy **PreparedStatement**.

PreparedStatement – używanie i wywoływanie

```
PreparedStatement preStmt =
    conn.prepareStatement("SELECT * FROM users WHERE user_name = ?");
preStmt.setString(1, userName);
preStmt.executeQuery();
```

Tworzymy obiekt klasy **`PreparedStatement`**.

Możemy mieć dowolną liczbę parametrów ustawianych w zapytaniu.

PreparedStatement – używanie i wywoływanie

```
PreparedStatement preStmt =
    conn.prepareStatement("SELECT * FROM users WHERE user_name = ?");
preStmt.setString(1, userName);
preStmt.executeQuery();
```

Tworzymy obiekt klasy **`PreparedStatement`**.

Możemy mieć dowolną liczbę parametrów ustawianych w zapytaniu.

Wywołujemy zapytanie.

Ostatni wstawiony element (w Javie)

Po każdej operacji wstawienia lub edycji możemy otrzymać **`ID`** (wartość klucza głównego) elementu, na którym pracowaliśmy.

```
String generatedColumns[] = { "ID" };
PreparedStatement preStmt = conn.prepareStatement(sql, generatedColumns);
```

```
ResultSet rs = preStmt.getGeneratedKeys();
if (rs.next()) {
    long id = rs.getLong(1);
    System.out.println("Inserted ID: " + id);
}
```

Ostatni wstawiony element (w Javie)

Po każdej operacji wstawienia lub edycji możemy otrzymać **`ID`** (wartość klucza głównego) elementu, na którym pracowaliśmy.

```
String generatedColumns[] = { "ID" };
PreparedStatement preStmt = conn.prepareStatement(sql, generatedColumns);
```

```
ResultSet rs = preStmt.getGeneratedKeys();
if (rs.next()) {
    long id = rs.getLong(1);
    System.out.println("Inserted ID: " + id);
}
```

Jest to sposób, aby poznać wartość klucza głównego dla właśnie wpisanego elementu.

Zadania

Pobieranie danych z tabeli

Wczytywanie elementów z tabeli

Dane z tabeli pobieramy za pomocą zapytania **SELECT**:

```
SELECT col_name_1, col_name_2
FROM table_name;
```

Aby wybrać wszystkie kolumny, możemy użyć symbolu gwiazdki ("*"):

```
SELECT * FROM table_name;
```

Należy jednak pamiętać, że jeśli nie jest to konieczne, to nie pobieramy wszystkich kolumn z tabeli.

Wczytywanie elementów z tabeli

Po wywołaniu zapytania, np.:

```
SELECT * FROM users;
```

elementy zwracane są w następującej postaci:

```
+---------+-----------+
| user_id | user_name |
+---------+-----------+
|       1 | Wojtek    |
|       2 | Jan       |
|       3 | Paweł     |
|       4 | Janusz    |
+---------+-----------+
4 rows in set (0.00 sec)
```

ResultSet

Czym jest ResultSet?

Jest to obiekt, który zawiera wyniki naszego zapytania. Można go porównać do tablicy wyników. ResultSet ma następujące metody:

- **`next()`**

  Zwraca wartość typu **`boolean`**, która informuje nas, czy są jeszcze jakieś elementy.

- **`getString("column_name")`**

  Zwraca wartość typu **`String`**, znajdującą się w kolumnie o podanej nazwie. Istnieją analogiczne metody dla innych typów.

Wczytywanie elementów z tabeli (Java)

```
Statement stmt = conn.createStatement();
ResultSet rs = stmt.executeQuery("SELECT * FROM users");
while (rs.next()) {
    String firstName = rs.getString("user_name");
    int id = rs.getInt("user_id");
    System.out.println(id + " " + firstName);
}
```

Wczytywanie elementów z tabeli (Java)

```
Statement stmt = conn.createStatement();
ResultSet rs = stmt.executeQuery("SELECT * FROM users");
while (rs.next()) {
    String firstName = rs.getString("user_name");
    int id = rs.getInt("user_id");
    System.out.println(id + " " + firstName);
}
```

Wykonujemy zapytanie.

Wczytywanie elementów z tabeli (Java)

```
Statement stmt = conn.createStatement();
ResultSet rs = stmt.executeQuery("SELECT * FROM users");
while (rs.next()) {
    String firstName = rs.getString("user_name");
    int id = rs.getInt("user_id");
    System.out.println(id + " " + firstName);
}
```

Wykonujemy zapytanie.

Pętla jest powtarzana tak długo, aż metoda zwróci **`false`**.

Wczytywanie elementów z tabeli (Java)

```
Statement stmt = conn.createStatement();
ResultSet rs = stmt.executeQuery("SELECT * FROM users");
while (rs.next()) {
    String firstName = rs.getString("user_name");
    int id = rs.getInt("user_id");
    System.out.println(id + " " + firstName);
}
```

Wykonujemy zapytanie.

Pętla jest powtarzana tak długo, aż metoda zwróci **`false`**.

Pobieramy dane typu **`String`** z kolumny o nazwie **`user_name`**.

Wczytywanie elementów z tabeli (Java)

```
Statement stmt = conn.createStatement();
ResultSet rs = stmt.executeQuery("SELECT * FROM users");
while (rs.next()) {
    String firstName = rs.getString("user_name");
    int id = rs.getInt("user_id");
    System.out.println(id + " " + firstName);
}
```

Wykonujemy zapytanie.

Pętla jest powtarzana tak długo, aż metoda zwróci **`false`**.

Pobieramy dane typu **`String`** z kolumny o nazwie **`user_name`**.

Pobieramy dane typu **`int`** z kolumny o nazwie **`user_id`**.

Klauzula WHERE

Możemy zawęzić wyniki wyszukiwania przez dodanie klauzuli **WHERE** do naszego zapytania **SELECT**.

```
SELECT col_name_1, col_name_2
FROM table_name
WHERE col_name_1 = <szukana wartość>;
```

Przykład

```
SELECT * FROM users WHERE user_name = "Wojtek";
```

Klauzula WHERE

Elementy zwracane są w sposób identyczny jak zapytanie pobierające wszystkie rekordy.

```
SELECT * FROM users WHERE user_name LIKE "Jan%";
```

```
+---------+-----------+
| user_id | user_name |
+---------+-----------+
|       2 | Jan       |
|       4 | Janusz    |
+---------+-----------+
2 rows in set (0.00 sec)
```

Operacje porównania w MySQL

=	Równe.
<>	Nierówne (w nowszych wersjach można użyć **!=**).
>	Większe niż.
>=	Większe równe.
<	Mniejsze niż.
<=	Mniejsze równe.
BETWEEN a AND b	Pomiędzy podanym zakresem (wliczając podany zakres).
LIKE	Szuka podanego wzorca (tylko napisy) 'Pi%' – wartość kolumny rozpoczyna się od **Pi** '%my' – wartość kolumny kończy się na **my** '%biały%' – wartość kolumny zawiera **biały**.
IN(a,b,c)	Znajduje się w zmiennych podanych w nawiasach.
NOT	Zaprzeczenie – może poprzedzać inne operacje.
OR / AND	Operatory logiczne łączące poszczególne wyrażenia.

Klauzula AS

Jeżeli z jakiegoś powodu (np. złączenia **JOIN**) w wynikach wyszukiwania mamy dwie kolumny o takiej samej nazwie to będziemy mieli dostęp tylko do jednej z nich.

Możemy zawsze nadać kolumnie nową nazwę (alias) na czas tego wyszukania.

Robimy to za pomocą klauzuli **AS**:

```
SELECT column_name AS column_alias FROM table_name;
```

Przykład

```
SELECT user_id AS id FROM Users;
```

Klauzula AS

Jeżeli z jakiegoś powodu (np. złączenia **JOIN**) w wynikach wyszukiwania mamy dwie kolumny o takiej samej nazwie to będziemy mieli dostęp tylko do jednej z nich.

Możemy zawsze nadać kolumnie nową nazwę (alias) na czas tego wyszukania.

Robimy to za pomocą klauzuli **AS**:

```
SELECT column_name AS column_alias FROM table_name;
```

Przykład

```
SELECT user_id AS id FROM Users;
```

Od tej pory kolumna **user_id** będzie widziana w wynikach jako **id**.

Klauzula ORDER BY

Możemy sortować znalezione wyniki względem jednej kolumny (lub więcej). Służy do tego klauzula **ORDER BY**. Sortowanie następuje według kolejności w zapytaniu.

```
SELECT col_name_1, col_name_2
FROM table_name
ORDER BY col_name_1 [ASC/DESC],
col_name_2 [ASC/DESC];
```

Przykład

```
SELECT * FROM users
ORDER BY user_name ASC;
```

```
+---------+-----------+
| user_id | user_name |
+---------+-----------+
|       1 | Antek     |
|       3 | Beata     |
|       2 | Wojtek    |
+---------+-----------+
3 rows in set (0.00 sec)
```

Klauzula ORDER BY

Możemy sortować znalezione wyniki względem jednej kolumny (lub więcej). Służy do tego klauzula **ORDER BY**. Sortowanie następuje według kolejności w zapytaniu.

```
SELECT col_name_1, col_name_2
FROM table_name
ORDER BY col_name_1 [ASC/DESC],
col_name_2 [ASC/DESC];
```

Wybieramy jedną z możliwości:
ASC – rosnąco (ascending),
DESC – malejąco (descending).

Przykład

```
SELECT * FROM users
ORDER BY user_name ASC;
```

```
+---------+-----------+
| user_id | user_name |
+---------+-----------+
|       1 | Antek     |
|       3 | Beata     |
|       2 | Wojtek    |
+---------+-----------+
3 rows in set (0.00 sec)
```

Funkcje agregujące

W zapytaniach SQL mamy możliwość wykorzystania m.in. następujących funkcji:

- **AVG** – średnia,
- **COUNT** – liczba wystąpień,
- **MAX** – maksymalna wartość,
- **MIN** – minimalna wartość,
- **SUM** – suma.

Przykłady:

```
SELECT SUM(col_name_1)
FROM table_name;
```

```
SELECT COUNT(*) FROM users;
```

```
+----------+
| COUNT(*) |
+----------+
|        3 |
+----------+
1 row in set (0,00 sec)
```

Funkcje agregujące

Funkcje agregujące mogą również zawierać dodatkowe warunki, określone np. przy użyciu klauzuli **WHERE**.

```
SELECT AVG(balance) FROM users WHERE user_name = "Marek";
```

```
+--------------+
| AVG(balance) |
+--------------+
|       2.5000 |
+--------------+
1 row in set (0,01 sec)
```

Klauzula GROUP BY

Załóżmy, że użytkownicy z naszej tabeli **`users`**, przynależą do kilku zespołów. Informacja o zespole znajduje się w dodatkowej kolumnie o nazwie **`user_group`**.

Naszym zadaniem jest obliczenie liczby użytkowników w poszczególnej grupie.

Możemy do tego celu skorzystać z klauzuli **`GROUP BY`**.

```
SELECT user_group, COUNT(*) FROM users
GROUP BY user_group;
```

```
+-------------+----------+
| user_group  | count(*) |
+-------------+----------+
| first_group |        5 |
| second_group|        9 |
+-------------+----------+
2 rows in set (0,00 sec)
```

Klauzula HAVING

Gdybyśmy chcieli pobrać tylko te grupy, które mają więcej niż jednego użytkownika, należy w tym celu skorzystać z klauzuli HAVING.

```
SELECT user_group, COUNT(*) AS size
FROM users GROUP BY user_group;
```

```
+------------+------+
| user_group | size |
+------------+------+
| first      |    2 |
| second     |    2 |
| third      |    1 |
+------------+------+
3 rows in set (0,00 sec)
```

```
SELECT user_group, COUNT(*) AS size
FROM users GROUP BY user_group
HAVING size > 1;
```

```
+------------+------+
| user_group | size |
+------------+------+
| first      |    2 |
| second     |    2 |
+------------+------+
2 rows in set (0,00 sec)
```

Klauzula HAVING

Gdybyśmy chcieli pobrać tylko te grupy, które mają więcej niż jednego użytkownika, należy w tym celu skorzystać z klauzuli HAVING.

```
SELECT user_group, COUNT(*) AS size
FROM users GROUP BY user_group;
```

```
+------------+------+
| user_group | size |
+------------+------+
| first      |    2 |
| second     |    2 |
| third      |    1 |
+------------+------+
3 rows in set (0,00 sec)
```

```
SELECT user_group, COUNT(*) AS size
FROM users GROUP BY user_group
HAVING size > 1;
```

```
+------------+------+
| user_group | size |
+------------+------+
| first      |    2 |
| second     |    2 |
+------------+------+
2 rows in set (0,00 sec)
```

Dla wygody nazywamy wynik funkcji agregującej.

Klauzula HAVING

Gdybyśmy chcieli pobrać tylko te grupy, które mają więcej niż jednego użytkownika, należy w tym celu skorzystać z klauzuli HAVING.

```
SELECT user_group, COUNT(*) AS size
FROM users GROUP BY user_group;
```

```
+------------+------+
| user_group | size |
+------------+------+
| first      |    2 |
| second     |    2 |
| third      |    1 |
+------------+------+
3 rows in set (0,00 sec)
```

```
SELECT user_group, COUNT(*) AS size
FROM users GROUP BY user_group
HAVING size > 1;
```

```
+------------+------+
| user_group | size |
+------------+------+
| first      |    2 |
| second     |    2 |
+------------+------+
2 rows in set (0,00 sec)
```

Dla wygody nazywamy wynik funkcji agregującej.

Określamy warunek.

LIMIT

Często chcemy pobrać tylko część danych.

Na przykład chcemy pobrać informację o dziesięciu ostatnio dodanych użytkownikach. Służy do tego polecenie **`LIMIT`**.

Przykład

```
SELECT * FROM users LIMIT 10;
```

OFFSET

Możemy również określić, od którego rekordu zaczynamy pobierać dane. Służy do tego polecenie **OFFSET**:

W tym przykładzie pobierzemy pięć wierszy (chyba że w bazie będzie ich mniej), zaczniemy od wiersza drugiego.

```
SELECT * FROM users LIMIT 5 OFFSET 2;
```

Krótsza forma polecenia

```
SELECT * FROM users LIMIT 2, 5;
```

Zwróć uwagę, że w skróconej formie najpierw określamy wiersz, od którego zaczynamy pobierać dane, a następnie liczbę wierszy do pobrania.

Zadania