

Spring + JDBC: dostęp do baz danych'

- Wejdź na <https://start.spring.io/> aby wygenerować szkielet projektu
 - W polu **Group** wpisz `wizut.tpsi`
 - W polu **Artifact** wpisz `lab9`
 - Dodaj biblioteki: **Web, Thymeleaf, DevTools, JDBC, H2**
 - Wybierz wersję 1.5.12
- Otwórz projekt w NetBeans
- Jeśli pracujesz na komputerze w sali laboratoryjnej, zmień wersję Javy w `pom.xml` na 1.7.

Konfiguracja dostępu do bazy danych

Baza danych H2

- W tym ćwiczeniu skorzystamy z wbudowanej bazy **H2** - dlatego przy generowaniu projektu dodaliśmy zależności **H2** i **JDBC**. Spring Boot przy starcie wykryje bazę danych **H2**, wystartuje ją i automatycznie skonfiguruje dostęp do niej. Baza danych będzie przechowywana w pamięci, po zatrzymaniu aplikacji zostanie zniszczona.
- W przypadku korzystania z innej bazy, parametry połączenia należałoby wpisać w pliku `application.properties`.

Inicjalizacja bazy danych

- Baza danych będzie tworzona automatycznie podczas startu naszej aplikacji, przy czym domyślnie będzie ona pusta (bez tabel i jakichkolwiek danych).
- Aby utworzyć tabele i wypełnić je wstępnie danymi, umieścimy w projekcie dwa pliki:
 - `schema.sql` - służy do tworzenia tabel,
 - `data.sql` - służy do wypełnienia tabel danymi.
- Pobierz załączone pliki `data.sql` i `schema.sql`, zapisz je w folderze `/src/main/resources`. Spring przy starcie wykryje je i wykona zawarte w nich polecenia.
- Aby dane z plików zostały poprawnie odczytane, musimy wskazać użyte w nich kodowanie znaków. Do pliku `application.properties` dodaj linijkę:
`spring.datasource.sql-script-encoding=UTF-8`

Struktura danych

- Utworzona baza jest bardzo prosta, zawiera jedną tabelę o nazwie `blog_post`. Każdy wpis na blogu składa się z pól: `id`, `title`, `content`.

Zadanie 1

- Przygotuj klasę `BlogPost`, zawierającą pola `id` (typu `Long`), `title` i `content` (typu `String`).
- Klasa powinna zawierać gettery, settery oraz pusty, bezargumentowy konstruktor.

Pobieranie danych z bazy

- Stwórz klasę `BlogRepository`. Umieścimy w niej kod związany z pobieraniem i zapisywaniem danych do bazy.
- Umieść na klasie adnotację `@Repository` - oznacza ona, że klasa ta będzie komponentem Spring typu `repository`.
- Dodaj pole `dataSource` i umieść na nim adnotację `@Autowired` - Spring automatycznie wstrzyknie nam skonfigurowany obiekt pozwalający na połączenie z bazą danych. Kod klasy powinien w tej chwili wyglądać następująco:

```
@Repository
public class BlogRepository {

    @Autowired
    private DataSource dataSource;

}
```

- Dodaj do klasy metodę `getAllPosts()`, która posłuży do pobrania listy postów z bazy:

```
public List<BlogPost> getAllPosts() {
    List<BlogPost> posts = new ArrayList<>();

    // tutaj pobierzemy posty z bazy danych...

    return posts;
}
```

- Dane wyciągniemy z bazy prostym zapytaniem SQL:

```
String sql = "select * from blog_post";
```

- Najpierw musimy nawiązać połączenie z bazą:

```
Connection con = dataSource.getConnection();
```

- Następnie stworzyć obiekt `Statement`:

```
Statement st = con.createStatement();
```

- I wreszcie wykonać zapytanie:

```
ResultSet rs = st.executeQuery(sql);
```

- Ponieważ wszystkie trzy obiekty (Connection, Statement i ResultSet) muszą zostać poprawnie zamknięte po zakończeniu komunikacji z bazą, użyjemy konstrukcji try with resources:

```
try(Connection con = dataSource.getConnection();  
    Statement st = con.createStatement();  
    ResultSet rs = st.executeQuery(sql);) {
```

```
}
```

- Pozostaje nam odczytać wiersz po wierszu wyniki zapytania:

```
while(rs.next()) {  
    Long id = rs.getLong("id");  
    String title = rs.getString("title");  
    String content = rs.getString("content");
```

```
}
```

- I wreszcie na podstawie każdego wiersza skonstruować obiekt BlogPost i dodać go do listy:

```
BlogPost post = new BlogPost(id, title, content);  
posts.add(post);
```

- Kompletny kod metody powinien wyglądać tak:

```
public List<BlogPost> getAllPosts() throws SQLException {  
    List<BlogPost> posts = new ArrayList<>();  
  
    String sql = "select * from blog_post";  
  
    try(Connection con = dataSource.getConnection();  
        Statement st = con.createStatement();  
        ResultSet rs = st.executeQuery(sql);) {  
  
        while(rs.next()) {  
            Long id = rs.getLong("id");  
            String title = rs.getString("title");  
            String content = rs.getString("content");  
  
            BlogPost post = new BlogPost(id, title, content);  
            posts.add(post);  
        }  
    }  
}
```

```

        return posts;
    }

```

- Zwróć uwagę na deklarację `throws SQLException` - każda z metod operujących na bazie może rzucić takim wyjątkiem. Nie łapiemy go, lecz pozwalamy, by został wysłany wyżej, np. do kontrolera, który będzie mógł go odpowiednio obsłużyć (np. pokazać stronę z komunikatem o błędzie).

Zadanie 2

Na stronie głównej chcemy wyświetlić jeden pod drugim wszystkie wpisy z bloga (np. nagłówek `<h2>` z tytułem wpisu, poniżej paragraf `<p>` z zawartością wpisu).
 - Przygotuj kontroler i stronę główną aplikacji. - Do kontrolera wstrzyknij repozytorium `BlogRepository`. - W metodzie kontrolera obsługującej stronę główną wywołaj metodę `getAllPosts()` z repozytorium, aby odczytać posty z bazy. - Przekaż posty do widoku i wyświetl.

Zapisywanie danych do bazy

Repozytorium - metoda `createPost()`

- Dodamy teraz do repozytorium metodę `createPost()`, która posłuży do zapisywania nowych postów do bazy danych:

```

public void createPost(BlogPost post) throws SQLException {
}

```

- Argumentem metody jest obiekt klasy `BlogPost`. Wyciągniemy z niego dane i przygotujemy polecenie SQL:

```

String sql = String.format(
    "insert into blog_post(title, content) values('%s', '%s')",
    post.getTitle(), post.getContent()
);

```

- Teraz nawiążemy połączenie z bazą i wykonamy zapytanie, podobnie jak przy odczycie danych. Kompletny kod metody powinien wyglądać tak:

```

public void createPost(BlogPost post) throws SQLException {
    String sql = String.format(
        "insert into blog_post(title, content) values('%s', '%s')",
        post.getTitle(), post.getContent()
    );

    try(Connection con = dataSource.getConnection();
        Statement st = con.createStatement() ) {
        st.executeUpdate(sql);
    }
}

```

Formularz webowy

- Na stronie głównej, powyżej listy wpisów, umieść formularz z polami `title` i `content`:

```
<form action="/newpost" method="POST">
  Tytuł: <input type="text" name="title" size="150"/>
  Treść:
    <textarea name="content" cols="150" rows="5"/>
    <input type="submit" value="Dodaj wpis" />
</form>
```

Obsługa formularza w kontrolerze

- Do kontrolera dodaj metodę zmapowaną na adres `/newpost`, obsługującą żądania wysłane metodą POST:

```
@PostMapping("/newpost")
public String newPost(BlogPost post) throws SQLException {

}
```

- W metodzie tej wywołamy metodę repozytorium `createPost()`, po czym przekierujemy przeglądarkę na stronę główną:

```
@PostMapping("/newpost")
public String newPost(BlogPost post) throws SQLException {
    blogRepo.createPost(post);
    return "redirect:/";
}
```

- Przetestuj działanie dodawania wpisów.

Ataki SQL Injection

Test podatności aplikacji na SQL Injection

- Wypróbujemy teraz na naszej aplikacji klasyczny atak **SQL Injection**. Powiedzmy, że hacker wchodzi na naszą stronę, i zamiast grzecznie wpisać treść postu, wpisuje w polu `Treść` następujący łańcuch:

```
'); delete from blog_post where ('1'='1
```

- Sprawdź, co się stanie.
- Co się stało? Dlaczego? Chwila na zastanowienie

...

...

...

...

- A tu klasyka gatunku z czeluści internetu - atak na fotoradar i szkolny dziennik:



Figure 1:

Obrona przed SQL Injection

- Powyższy atak się udał, ponieważ po sklejeniu łańcucha zaszytego w kodzie z tekstem podanym w formularzu, otrzymaliśmy zapytanie:

```
insert into blog_post(title, content) values('Ataaaaak!!!!', ''); delete from blog_post
```

- Morał jest następujący: nigdy nie wolno używać parametrów pochodzących z zewnątrz do sklejanego zapytania SQL. Ochronić możemy się na dwa sposoby:
 - odpowiednio filtrując przekazane parametry,
 - używając `PreparedStatement`. W ćwiczeniu skorzystamy z tej drugiej opcji.
- Zmienimy metodę `createPost()`, tak by korzystała z `PreparedStatement` zamiast sklejać SQL z kawałków Stringów:



Figure 2:

```
public void createPost(BlogPost post) throws SQLException {
    String sql = "insert into blog_post(title, content) values(?, ?)";

    try(Connection con = dataSource.getConnection();
        PreparedStatement ps = con.prepareStatement(sql)) {

        ps.setString(1, post.getTitle());
        ps.setString(2, post.getContent());

        ps.executeUpdate();
    }
}
```

- Sprawdź, czy dodawanie wpisów działa. Spróbuj przeprowadzić atak SQL Injection.

Zadanie 3

- Zaimplementuj usuwanie dodanych wpisów. Przy każdym wpisie powinien pojawić się link **Usuń**.
- Pamiętaj, by użyć `PreparedStatement`