

Przydatne strony:

<http://pirate.shu.edu/~wachsmut/Teaching/CSAS2214/Virtual/Lectures/chat-client-server.html>
<http://pirate.shu.edu/~wachsmut/Teaching/CSAS2214/Virtual/Lectures/>

Jak połączyć Springboot z SQLite?

1. Dodaj zależności w pliku `pom.xml`

Musisz dodać zależność do sterownika SQLite oraz Spring Data JPA. W pliku `pom.xml` dodaj następujące zależności:

```
<dependencies>
  <!-- Spring Data JPA -->
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
  </dependency>

  <!-- Sterownik SQLite -->
  <dependency>
    <groupId>org.xerial</groupId>
    <artifactId>sqlite-jdbc</artifactId>
    <version>3.42.0.0</version>
  </dependency>
</dependencies>
```

2. Skonfiguruj plik `application.properties` lub `application.yml`

```
spring.datasource.url=jdbc:sqlite:path_to_your_db_file.db
spring.datasource.driver-class-name=org.sqlite.JDBC
spring.jpa.database-platform=org.hibernate.dialect.SQLiteDialect
spring.jpa.hibernate.ddl-auto=update
```

Uwaga: Zamień `path_to_your_db_file.db` na faktyczną ścieżkę do pliku bazy danych SQLite.

3. Ustaw klasę SQLiteDialect

SQLite nie ma oficjalnego dialektu w Hibernate, więc możesz potrzebować własnej klasy SQLiteDialect, która zaimplementuje dialekt Hibernate dla SQLite. Możesz utworzyć taką klasę:

```
import org.hibernate.dialect.Dialect;
import java.sql.Types;

public class SQLiteDialect extends Dialect {
    public SQLiteDialect() {
        registerColumnType(Types.BLOB, "blob");
        registerColumnType(Types.INTEGER, "integer");
        registerColumnType(Types.REAL, "real");
        registerColumnType(Types.VARCHAR, "text");
    }

    @Override
    public boolean hasAlterTable() {
        return false; // SQLite nie obsługuje niektórych operacji
ALTER TABLE
    }
}
```

4. Twórz repozytoria i encje

Tworzenie encji i repozytoriów działa jak w przypadku innych baz danych. Na przykład:

```
import jakarta.persistence.Entity;
import jakarta.persistence.Id;

@Entity
public class Person {
    @Id
    private Long id;
    private String name;

    // Gettery i settery
}
```

Repozytorium:

```
import org.springframework.data.jpa.repository.JpaRepository;

public interface PersonRepository extends JpaRepository<Person,
Long> {
}
```

5. Uruchom aplikację

Po tych zmianach możesz uruchomić aplikację, a Spring Boot powinien automatycznie nawiązać połączenie z bazą SQLite i zarządzać encjami.

Tym sposobem połączyłeś Spring Boot z SQLite!

Jak umożliwić wysłanie coś użytkownikowi na serwer Springboot?

Aby umożliwić wysłanie czegoś użytkownikowi na serwerze Spring Boot, musisz skonfigurować odpowiednie endpointy w aplikacji. Zazwyczaj odbywa się to poprzez utworzenie kontrolera, który obsługuje żądania HTTP i zwraca odpowiedź użytkownikowi. Oto prosty przykład, jak to zrobić:

Krok 1: Utwórz kontroler

Kontroler obsługuje zapytania od klienta (np. wysyła dane JSON, pliki, HTML itp.).

```
import org.springframework.web.bind.annotation.GetMapping;

import org.springframework.web.bind.annotation.RequestMapping;

import org.springframework.web.bind.annotation.RestController;


@RestController

@RequestMapping("/api")

public class MyController {

    @GetMapping("/hello")

    public String sendHello() {

        return "Hello, User!";

    }

}
```

W tym przypadku kontroler nasłuchuje na endpointzie /api/hello i zwraca zwykły tekst "Hello, User!" użytkownikowi, który wysłał zapytanie GET.

Krok 2: Uruchom aplikację

Aby uruchomić aplikację, upewnij się, że masz odpowiednią klasę główną:

```
import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication

public class MySpringBootApplication {

    public static void main(String[] args) {

        SpringApplication.run(MySpringBootApplication.class, args);

    }

}
```

Krok 3: Testowanie

Możesz teraz wysłać zapytanie GET do adresu `http://localhost:8080/api/hello` przy użyciu przeglądarki, Postmana lub cURL:

```
curl http://localhost:8080/api/hello
```

Jak wysłać plik?

Krok 1: Utwórz kontroler do wysyłania pliku

Tutaj użyjemy klasy `ResponseEntity` do zwrócenia pliku jako odpowiedzi. Założmy, że mamy plik, który chcemy wysłać, np. `example.txt`, znajdujący się w folderze `src/main/resources`.

```
import org.springframework.core.io.ClassPathResource;

import org.springframework.core.io.Resource;

import org.springframework.http.HttpHeaders;

import org.springframework.http.HttpStatus;

import org.springframework.http.ResponseEntity;

import org.springframework.web.bind.annotation.GetMapping;

import org.springframework.web.bind.annotation.RequestMapping;

import org.springframework.web.bind.annotation.RestController;


import java.io.IOException;


@RestController

@RequestMapping("/api")

public class FileController {


    @GetMapping("/download")

    public ResponseEntity<Resource> downloadFile() throws
IOException {

        // ładowanie pliku z resources

        Resource file = new ClassPathResource("example.txt");


        // Ustawienie nagłówków odpowiedzi HTTP

        HttpHeaders headers = new HttpHeaders();

        headers.add(HttpHeaders.CONTENT_DISPOSITION, "attachment;
filename=example.txt");
```

```
        // Zwracanie pliku w odpowiedzi

        return new ResponseEntity<>(file, headers, HttpStatus.OK);
    }
}
```

Krok 2: Plik w katalogu resources

Upewnij się, że plik, który chcesz wysłać, znajduje się w folderze `src/main/resources`. W tym przypadku plik ma nazwę `example.txt`.

Krok 3: Uruchom aplikację i testuj

Teraz możesz uruchomić aplikację Spring Boot i przetestować endpoint. Aby pobrać plik, wystarczy wysłać zapytanie GET do adresu:

`http://localhost:8080/api/download`

Poradnik: Tworzenie aplikacji klient-serwer w JavaFX

JavaFX to framework do tworzenia aplikacji GUI w Javie. Możemy go połączyć z mechanizmami klient-serwer, aby stworzyć interfejs użytkownika, który komunikuje się z serwerem, np. w celu wymiany danych. W tym poradniku stworzymy prostą aplikację klient-serwer, gdzie klient zbudowany w JavaFX będzie komunikował się z serwerem poprzez sockety.

rok 1: Konfiguracja projektu

Na początku musisz mieć środowisko IDE z zainstalowaną Javą oraz biblioteką JavaFX. Możesz skorzystać np. z IntelliJ IDEA lub Eclipse. Upewnij się, że w Twoim projekcie dodana jest JavaFX jako

zależność (w wypadku maven/gradle lub jako biblioteka w tradycyjnych projektach).

Krok 2: Tworzenie serwera

Na serwerze będziemy nasłuchiwać połączeń od klienta. Serwer wykorzysta klasy takie jak `ServerSocket` do nasłuchiwania oraz `Socket` do wymiany danych z klientem.

Kod serwera (`Server.java`):

```
import java.io.*;

import java.net.*;

public class Server {

    public static void main(String[] args) {

        try (ServerSocket serverSocket = new ServerSocket(5555)) {

            System.out.println("Serwer nasłuchuje na porcie
5555...");

            while (true) {

                Socket clientSocket = serverSocket.accept(); //
akceptowanie połączeń

                System.out.println("Połączono z klientem");

                // Tworzenie strumienia danych

                BufferedReader in = new BufferedReader(new
InputStreamReader(clientSocket.getInputStream()));

                PrintWriter out = new
PrintWriter(clientSocket.getOutputStream(), true);
```



```

        // Odczyt danych od klienta

        String messageFromClient = in.readLine();

        System.out.println("Wiadomość od klienta: " +
messageFromClient);


        // Wysyłanie odpowiedzi do klienta

        out.println("Serwer otrzymał: " +
messageFromClient);


        // Zamykanie połączenia

        clientSocket.close();

    }

} catch (IOException e) {

    e.printStackTrace();

}

}

}

```

Krok 3: Tworzenie klienta w JavaFX

Klient będzie zbudowany przy użyciu JavaFX, aby użytkownik mógł wprowadzać wiadomości do serwera. Komunikacja odbędzie się za pomocą socketów, tak jak w przypadku serwera.

Kod klienta z interfejsem JavaFX (Client.java):

```

import javafx.application.Application;

import javafx.scene.Scene;

import javafx.scene.control.*;

```

```
import javafx.scene.layout.VBox;

import javafx.stage.Stage;


import java.io.*;

import java.net.Socket;


public class Client extends Application {

    @Override

    public void start(Stage primaryStage) {

        // Tworzenie GUI

        TextField inputField = new TextField();

        Button sendButton = new Button("Wyślij");

        TextArea messagesArea = new TextArea();

        messagesArea.setEditable(false); // pole tekstowe do
wyświetlania wiadomości

        VBox vbox = new VBox(10, inputField, sendButton,
messagesArea);


        sendButton.setOnAction(event -> {

            String message = inputField.getText();

            if (!message.isEmpty()) {

                sendMessageToServer(message, messagesArea);

                inputField.clear();

            }

        })

    }

}
```

```
});

// Konfiguracja sceny

Scene scene = new Scene(vbox, 400, 300);

primaryStage.setScene(scene);

primaryStage.setTitle("JavaFX Klient");

primaryStage.show();

}

// Metoda do komunikacji z serwerem

private void sendMessageToServer(String message, TextArea
messagesArea) {

    try (Socket socket = new Socket("localhost", 5555);

        BufferedReader in = new BufferedReader(new
InputStreamReader(socket.getInputStream()));

        PrintWriter out = new
PrintWriter(socket.getOutputStream(), true)) {

        // Wysyłanie wiadomości do serwera

        out.println(message);

        // Odbieranie odpowiedzi od serwera

        String response = in.readLine();

        messagesArea.appendText("Serwer: " + response + "\n");

    } catch (IOException e) {
```

```
        e.printStackTrace();

        messagesArea.appendText("Błąd: nie można połączyć z
serverem\n");
    }
}

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

Poradnik SpringBoot

1. Konfiguracja projektu

a. Spring Initializr

Najprostszym sposobem na rozpoczęcie projektu Spring Boot jest skorzystanie ze Spring Initializr:

- Wejdź na start.spring.io.
- Wybierz język (Java), wersję Spring Boot, a następnie dodaj zależności, takie jak „Spring Web” dla aplikacji webowych, „Spring Data JPA” dla baz danych itp.
- Wygeneruj projekt i rozpakuj plik ZIP.

b. Gradle/Maven

Spring Boot wspiera zarówno Gradle, jak i Maven jako systemy zarządzania zależnościami. Jeśli pracujesz z Mavenem, twój plik `pom.xml` będzie zawierał podstawowe zależności, np.:

```
xml
Skopiuj kod

<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```



2. Struktura projektu

Po utworzeniu projektu, struktura projektu wygląda mniej więcej tak:

- `src/main/java`: Kod źródłowy aplikacji.
- `src/main/resources`: Pliki konfiguracyjne, np. `application.properties` lub `application.yml`.
- `pom.xml` (lub `build.gradle`): Plik konfiguracji zależności.

3. Tworzenie aplikacji Spring Boot

Główna klasa uruchomieniowa jest oznaczona adnotacją `@SpringBootApplication`, która łączy w sobie kilka innych adnotacji Springa:

```
java
Skopiuj kod


@SpringBootApplication
public class MyApplication {
    public static void main(String[] args) {
        SpringApplication.run(MyApplication.class, args);
    }
}
```



4. Tworzenie kontrolerów (Controllers)

Kontrolery w Spring Boot obsługują zapytania HTTP. Adnotacja `@RestController` definiuje klasę jako kontroler, który zwraca dane bezpośrednio w formacie JSON lub XML (zamiast np. stron HTML).

java

 Skopiuj kod


```
@RestController
public class MyController {

    @GetMapping("/hello")
    public String sayHello() {
        return "Hello, Spring Boot!";
    }
}
```

5. Konfiguracja plików właściwości

Konfigurację aplikacji można umieszczać w pliku `application.properties` lub `application.yml`, np.:

properties

 Skopiuj kod


```
server.port=8081
spring.datasource.url=jdbc:mysql://localhost:3306/mydb
spring.datasource.username=user
spring.datasource.password=pass
```

6. Połączenie z bazą danych za pomocą Spring Data JPA

Spring Boot upraszcza pracę z bazami danych dzięki Spring Data JPA. Musisz tylko dodać zależność `spring-boot-starter-data-jpa` do swojego projektu i skonfigurować połączenie z bazą danych w pliku `application.properties`.

Przykład prostej encji (Entity):

java

 Skopiuj kod

```
@Entity
public class User {

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

    private String name;
    private String email;

    // Getters and Setters

}
```



Tworzenie repozytorium (Repository) dla encji:

java

Skopiuj kod

```
public interface UserRepository extends JpaRepository<User, Long> {  
}
```

7. Obsługa wyjątków (Exceptions)

Spring Boot obsługuje globalne wyjątki za pomocą adnotacji `@ControllerAdvice`. Można zaimplementować klasę obsługującą błędy:

java

Skopiuj kod

```
@ControllerAdvice  
public class GlobalExceptionHandler {  
  
    @ExceptionHandler(value = Exception.class)  
    public ResponseEntity<Object> handleException(Exception ex) {  
        return new ResponseEntity<>("Wystąpił błąd: " + ex.getMessage(), HttpStatus.INTERNAL_SERVER_ERROR);  
    }  
}
```



8. Testowanie

Spring Boot oferuje narzędzia do testowania aplikacji, takie jak `@SpringBootTest`, które uruchamiają kontekst aplikacji podczas testów:

java

Skopiuj kod


```
@SpringBootTest  
class MyApplicationTests {  
  
    @Test  
    void contextLoads() {  
    }  
}
```


9. Bezpieczeństwo w Spring Boot (Spring Security)

Dodanie Spring Security do aplikacji wymaga dodania zależności `spring-boot-starter-security`. Po tym kroku Spring automatycznie dodaje podstawowe mechanizmy autoryzacji i uwierzytelniania.

Przykład prostej konfiguracji Spring Security:

java

 Skopiuj kod

```
@Configuration
@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            .authorizeRequests()
            .anyRequest().authenticated()
            .and()
            .httpBasic();
    }
}
```

