Zakłady online

Warsztat na wolny tydzień

v3.0

Cel warsztatu

Celem warsztatu jest napisanie dwóch aplikacji. Pierwsza z nich ma udostępniać **API** z określonymi metodami.

Druga aplikacja będzie z tych metod korzystała. Będzie się ona łączyć z pierwszą w celu pobrania i przetworzenia danych.

Do stworzenia **API** wykorzystamy **Spring MVC** oraz dodatkowe biblioteki:

- **Jackson**
- **Swagger**
- **Faker**

Swagger

Odpowiednio utworzona dokumentacja interfejsu programistycznego jest nieodłącznym elementem każdej aplikacji, która udostępnia **API**.

Swagger to narzędzie, które automatycznie utworzy dokumentację do naszych metod, w postaci webowego interfejsu z możliwością łatwego testowania utworzonych przez nas metod.

Z pomocą biblioteki **Swagger** udostępniamy interaktywną dokumentację endpointów naszej aplikacji.

Pozwala to programiście zapoznać się ze wszystkimi metodami, dokładnymi parametrami, przyjmowanymi przez metody ,a co najważniejsze pozwala te metody wypróbować.

Swagger udostępnia również narzędzia pozwalające definiować metody dostępowe, na podstawie których automatycznie generuje kod w wybranych technologiach.

Strona projektu: https://swagger.io

Faker

Podczas tworzenia i testowania tworzonego **API** wykorzystamy bibliotekę **Faker**.

Przy jej pomocy wygenerujemy przykładowe dane, którymi będzie się posługiwać nasza aplikacja.

Jest to szczególnie przydatne, jeżeli mamy projekt prezentować i potrzebujemy danych wyglądających realnie.

Strona projektu: https://github.com/DiUS/java-faker

Przygotowanie

Tworzenie projektu

Tworzymy projekt Maven i definiujemy właściwości:

```
<properties>
  <org.springframework-version>
    4.3.7.RELEASE
  </org.springframework-version>
  <failOnMissingWebXml>false</failOnMissingWebXml>
  <maven.compiler.source>1.8</maven.compiler.source>
  <maven.compiler.target>1.8</maven.compiler.target>
</properties>
```

Tworzenie projektu

Tworzymy projekt Maven i definiujemy właściwości:

```
<properties>
  <org.springframework-version>
    4.3.7.RELEASE
  </org.springframework-version>
  <failOnMissingWebXml>false</failOnMissingWebXml>
  <maven.compiler.source>1.8</maven.compiler.source>
  <maven.compiler.target>1.8</maven.compiler.target>
</properties>
```

Wersja Springa. W chwili tworzenia prezentacji ta wersja jest najnowszą zalecaną.

Tworzenie projektu

Tworzymy projekt Maven i definiujemy właściwości:

```
<properties>
  <org.springframework-version>
    4.3.7.RELEASE
  </org.springframework-version>
  <failOnMissingWebXml>false</failOnMissingWebXml>
  <maven.compiler.source>1.8</maven.compiler.source>
  <maven.compiler.target>1.8</maven.compiler.target>
</properties>
```

Wersja Springa. W chwili tworzenia prezentacji ta wersja jest najnowszą zalecaną.

Informacja dla IntelliJ, że w przypadku braku pliku web.xml ma nie zwracać błędu.

Tworzenie projektu

Tworzymy projekt Maven i definiujemy właściwości:

```
<properties>
  <org.springframework-version>
    4.3.7.RELEASE
  </org.springframework-version>
  <failOnMissingWebXml>false</failOnMissingWebXml>
  <maven.compiler.source>1.8</maven.compiler.source>
  <maven.compiler.target>1.8</maven.compiler.target>
</properties>
```

Wersja Springa. W chwili tworzenia prezentacji ta wersja jest najnowszą zalecaną.

Informacja dla IntelliJ, że w przypadku braku pliku web.xml ma nie zwracać błędu.

Wersja javy dla mavena.

Tworzenie projektu

Uzupełniamy plik **pom.xml**, dodając wymagane zależności:

```
<dependencies>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-webmvc</artifactId>
        <version>${org.springframework-version}</version>
    </dependency>
    <dependency>
        <groupId>javax.servlet</groupId>
        <artifactId>javax.servlet-api</artifactId>
        <version>3.1.0</version>
        <scope>provided</scope>
    </dependency>
</dependencies>
```

Tworzenie projektu

Definiujemy klasę konfiguracji.

Na tym etapie nasz projekt nie różni się od tworzonych wcześniej aplikacji z wykorzystaniem Spring MVC.

```
@Configuration
@EnableWebMvc
@ComponentScan(basePackages = "pl.coderslab")
public class AppConfig extends WebMvcConfigurerAdapter {
}
```

Tworzenie projektu

Dodajemy inicjalizator aplikacji. Korzystamy z innej, nieco uproszczonej jego implementacji w porównaniu do tej, z której korzystaliśmy wcześniej.

```
public class AppInitializer extends
  AbstractAnnotationConfigDispatcherServletInitializer {
    @Override
    protected Class<?>[] getRootConfigClasses() { return null; }

    @Override
    protected Class<?>[] getServletConfigClasses() {
        return new Class[]{AppConfig.class};  }

    @Override
    protected String[] getServletMappings() {
        return new String[]{"/"};   }
}
```

Tworzenie projektu

Dodajemy inicjalizator aplikacji. Korzystamy z innej, nieco uproszczonej jego implementacji w porównaniu do tej, z której korzystaliśmy wcześniej.

```
public class AppInitializer extends
  AbstractAnnotationConfigDispatcherServletInitializer {
    @Override
    protected Class<?>[] getRootConfigClasses() { return null; }

    @Override
    protected Class<?>[] getServletConfigClasses() {
        return new Class[]{AppConfig.class};  }

    @Override
    protected String[] getServletMappings() {
        return new String[]{"/"};   }
}
```

Określamy naszą klasę konfiguracji.

Tworzenie projektu

Rejestrujemy filtr ustawiający odpowiednie kodowanie:

```
public class AppInitializer extends
  AbstractAnnotationConfigDispatcherServletInitializer {

    // ... kod z poprzedniego slajdu

    @Override
    protected Filter[] getServletFilters() {
        CharacterEncodingFilter characterEncodingFilter =
                    new CharacterEncodingFilter();
        characterEncodingFilter.setEncoding("UTF-8");
        return new Filter[] { characterEncodingFilter };
    }
}
```

Tworzenie projektu

W naszym projekcie moglibyśmy samodzielnie przekształcać obiekty na format **JSON**. Nie jest to jednak zbyt wygodne.

Posłużymy się w tym celu biblioteką **Jackson**, należy uzupełnić zależności w pliku **pom.xml**:

```
<dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-databind</artifactId>
    <version>2.9.4</version>
</dependency>
```

Po wykryciu biblioteki w projekcie, Spring zadba o odpowiednie przekształcenia obiektu na **JSON**.

Tworzenie projektu

Uzupełniamy zależności odpowiedzialne za bibliotekę **faker**, którą posłużymy się nią w celu wygenerowania testowych danych dla **API**.

W pliku **pom.xml** dodajemy:

```
<dependency>
    <groupId>com.github.javafaker</groupId>
    <artifactId>javafaker</artifactId>
    <version>0.14</version>
</dependency>
```

Tworzenie projektu

Uzupełnimy nasz projekt o konfigurację dla biblioteki do tworzenia automatycznej dokumentacji **Swagger**, dodając do pliku **pom.xml** następujące zależności:

```
<dependency>
    <groupId>io.springfox</groupId>
    <artifactId>springfox-swagger2</artifactId>
    <version>2.6.1</version>
</dependency>
<dependency>
    <groupId>io.springfox</groupId>
    <artifactId>springfox-swagger-ui</artifactId>
    <version>2.6.1</version>
    <scope>compile</scope>
</dependency>
```

Swagger konfiguracja

Następnie uzupełniamy klasę konfiguracji **AppConfig** o poniższą metodę:

```
@Override
public void addResourceHandlers(final ResourceHandlerRegistry registry) {

    registry.addResourceHandler("swagger-ui.html")
            .addResourceLocations("classpath:/META-INF/resources/");

    registry.addResourceHandler("/webjars/**")
            .addResourceLocations("classpath:/META-INF/resources/webjars/");
}
```

Definiujemy w ten sposób jakie zasoby jakie mają być dostępne dla wyszczególnionych ścieżek URL.

Swagger konfiguracja

Utworzymy również dodatkową klasę konfiguracji **SwaggerConfig**

```
@Configuration
@EnableSwagger2
public class SwaggerConfig {

    private ApiInfo apiInfo() {
        return new ApiInfoBuilder()
                .title("Simple with swagger")
                .description("Demo with Swagger")
                .version("1.0.0")
                .build();
    }
}
```

Swagger konfiguracja

Utworzymy również dodatkową klasę konfiguracji **SwaggerConfig**

```
@Configuration
@EnableSwagger2
public class SwaggerConfig {

    private ApiInfo apiInfo() {
        return new ApiInfoBuilder()
                .title("Simple with swagger")
                .description("Demo with Swagger")
                .version("1.0.0")
                .build();
    }
}
```

Określamy opis dla naszego **API**.

Swagger konfiguracja

Uzupełniamy klasę o kolejną metodę:

```
@Configuration
@EnableSwagger2
public class SwaggerConfig {
    //wcześniejsza metoda określająca opis.
    @Bean
    public Docket api() {
        return new Docket(DocumentationType.SWAGGER_2)
                .apiInfo(apiInfo()).select()
                .apis(RequestHandlerSelectors.basePackage("pl.coderslab.web.rest"))
                .paths(PathSelectors.any())
                .build();
    }
}
```

Swagger konfiguracja

Uzupełniamy klasę o kolejną metodę:

```
@Configuration
@EnableSwagger2
public class SwaggerConfig {
    //wcześniejsza metoda określająca opis.
    @Bean
    public Docket api() {
        return new Docket(DocumentationType.SWAGGER_2)
                .apiInfo(apiInfo()).select()
                .apis(RequestHandlerSelectors.basePackage("pl.coderslab.web.rest"))
                .paths(PathSelectors.any())
                .build();
    }
}
```

Włączamy Swagger.

Swagger konfiguracja

Uzupełniamy klasę o kolejną metodę:

```
@Configuration
@EnableSwagger2
public class SwaggerConfig {
    //wcześniejsza metoda określająca opis.
    @Bean
    public Docket api() {
        return new Docket(DocumentationType.SWAGGER_2)
                .apiInfo(apiInfo()).select()
                .apis(RequestHandlerSelectors.basePackage("pl.coderslab.web.rest"))
                .paths(PathSelectors.any())
                .build();
    }
}
```

Określamy metodę która udostępnia opis.

Swagger konfiguracja

Uzupełniamy klasę o kolejną metodę:

```
@Configuration
@EnableSwagger2
public class SwaggerConfig {
    //wcześniejsza metoda określająca opis.
    @Bean
    public Docket api() {
        return new Docket(DocumentationType.SWAGGER_2)
                .apiInfo(apiInfo()).select()
                .apis(RequestHandlerSelectors.basePackage("pl.coderslab.web.rest"))
                .paths(PathSelectors.any())
                .build();
    }
}
```

Określamy pakiet z którego metody mają być składnikami api. Nie wszystkie dostępne metody musimy udostępniać.

Swagger konfiguracja

Uzupełniamy klasę o kolejną metodę:

```
@Configuration
@EnableSwagger2
public class SwaggerConfig {
    //wcześniejsza metoda określająca opis.
    @Bean
    public Docket api() {
        return new Docket(DocumentationType.SWAGGER_2)
                .apiInfo(apiInfo()).select()
                .apis(RequestHandlerSelectors.basePackage("pl.coderslab.web.rest")
                .paths(PathSelectors.any())
                .build();
    }
}
```

Określamy ścieżki które mają nasze wchodzić w skład udostępnianego **API**, w naszym przypadku określamy że mają to być wszystkie adresy.

Zadanie

Repozytorium

- Załóż nowe repozytorium Git na GitHubie.
- Pamiętaj o robieniu commitów (również co każde zadanie).

- Stwórz plik .gitignore i dodaj do niego wszystkie podstawowe dane: (katalog z danymi twojego IDE, jeżeli istnieje, pliki *.class, itp.).

Przykład punkt dostępowy

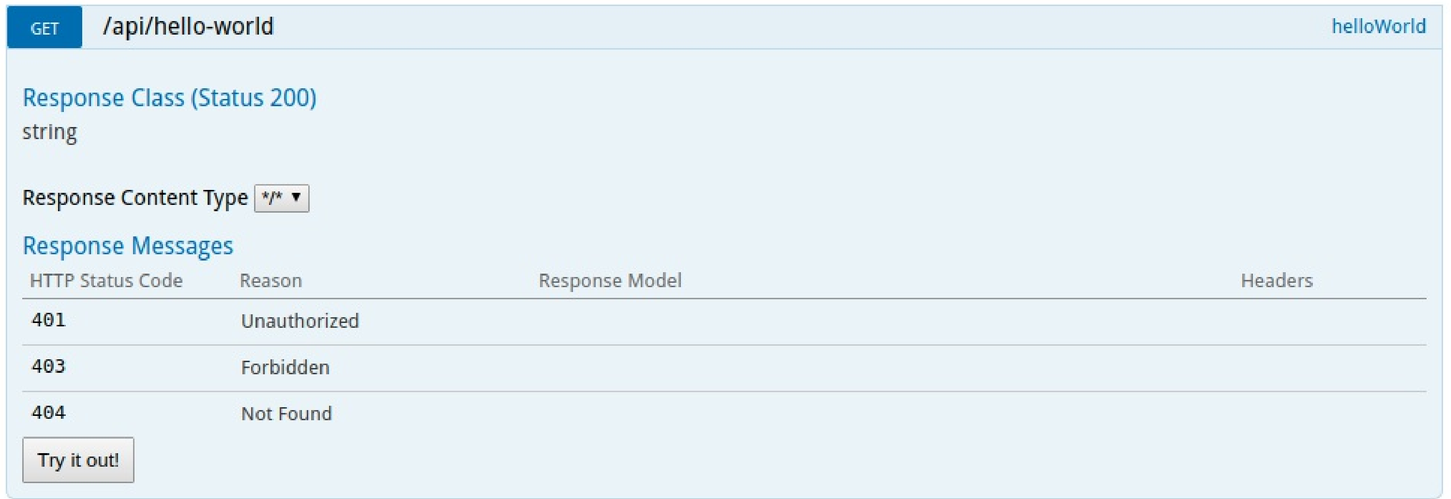
W pakiecie **`pl.coderslab.web.rest`** utworzymy klasę o nazwie **`FakerResource`** z metodą **API** zwracającą napis: **`Hello World`**.

```
@RestController
@RequestMapping("/api")
public class FakerResource {

    @GetMapping(path= "/hello-world")
    public String helloWorld() {
        return "Hello World";
    }
}
```

Przykład punkt dostępowy

Wchodząc na adres: **http://localhost:8080/swagger-ui.html** otrzymamy listę punktów dostępowych z możliwością przejścia do każdego z nich i przetestowania.



Faker

Przy pomocy biblioteki **Faker** utworzymy dodatkowe metody, udostępniające informacje na temat meczy z różnych dyscyplin oraz ich wyników.

Uzupełniamy zależność odpowiedzialną za bibliotekę **Faker**:

```
<dependency>
    <groupId>com.github.javafaker</groupId>
    <artifactId>javafaker</artifactId>
    <version>0.14</version>
</dependency>
```

Zewnętrze źródło danych

Użyjemy również biblioteki **org.json** – posłuży nam ona do generowania obiektów w formacie **JSON**:

```
<dependency>
    <groupId>org.json</groupId>
    <artifactId>json</artifactId>
    <version>20180130</version>
</dependency>
```

Faker – przykład

Faker posiada szereg przydatnych, które pozwalają generować przykładowe dane z różnych zakresów,

Przykład generowania faktów o Chacku Norrisie:

```
Chuck Norris does not use revision control software. None of his code has ever needed revision.
The programs that Chuck Norris writes don't have version numbers because he only writes them once.
If a user reports a bug or has a feature request they don't live to see the sun set.
```

Dokumentację wszystkich generatorów znajdziemy pod adresem:

http://dius.github.io/java-faker/apidocs/index.html

Faker – przykład

Przykład generowania danych:

```
Faker faker = new Faker();

String name = faker.name().fullName();
String firstName = faker.name().firstName();
String lastName = faker.name().lastName();
String streetAddress = faker.address().streetAddress();
```

Przykładowy otrzymany zestaw wartości to:

```
Hettie Christiansen
Rhoda
Ortiz
1586 Myrl Run,
```

Faker – przykład

Definiując obiekt możemy również ustalić wersję językową, np:

```
Faker faker = new Faker(new Locale("pl_PL"))

String name = faker.name().fullName();
String firstName = faker.name().firstName();
String lastName = faker.name().lastName();
String streetAddress = faker.address().streetAddress();
```

Przykładowy otrzymany zestaw wartości to:

```
Klarencjusz Markowski
Pankracy
Jędrzejczyk
ul. Strzelczyk 1170
```

Zewnętrze źródło danych

```
@Service
public class FakerService {
    private ArrayList<JSONObject> todayGames = new ArrayList<>();
    public void regenerate() throws JSONException {
        Faker faker = new Faker();
        todayGames.clear();
        for (int i = 0; i < 10; i++) {
            JSONObject oJsonInner = new JSONObject();
            oJsonInner.put("firstTeam", faker.team().name());
            oJsonInner.put("firstPoints", faker.number().randomDigitNotZero());
            oJsonInner.put("secondTeam", faker.team().name());
            oJsonInner.put("secondPoints", faker.number().randomDigitNotZero());
            oJsonInner.put("sport", faker.team().sport());
            todayGames.add(oJsonInner);
}}}
```

Zewnętrze źródło danych

```
@Service
public class FakerService {
    private ArrayList<JSONObject> todayGames = new ArrayList<>();
    public void regenerate() throws JSONException {
        Faker faker = new Faker();
        todayGames.clear();
        for (int i = 0; i < 10; i++) {
            JSONObject oJsonInner = new JSONObject();
            oJsonInner.put("firstTeam", faker.team().name());
            oJsonInner.put("firstPoints", faker.number().randomDigitNotZero());
            oJsonInner.put("secondTeam", faker.team().name());
            oJsonInner.put("secondPoints", faker.number().randomDigitNotZero());
            oJsonInner.put("sport", faker.team().sport());
            todayGames.add(oJsonInner);
}}}
```

Tworzymy serwis **FakerService**, odpowiedzialny za generowanie danych.

Zewnętrze źródło danych

```
@Service
public class FakerService {
    private ArrayList<JSONObject> todayGames = new ArrayList<>();
    public void regenerate() throws JSONException {
        Faker faker = new Faker();
        todayGames.clear();
        for (int i = 0; i < 10; i++) {
            JSONObject oJsonInner = new JSONObject();
            oJsonInner.put("firstTeam", faker.team().name());
            oJsonInner.put("firstPoints", faker.number().randomDigitNotZero());
            oJsonInner.put("secondTeam", faker.team().name());
            oJsonInner.put("secondPoints", faker.number().randomDigitNotZero());
            oJsonInner.put("sport", faker.team().sport());
            todayGames.add(oJsonInner);
}}}
```

Tworzymy listę obiektów **JSONObject** – dzięki wykorzystaniu tej klasy możemy dynamicznie kreować zestaw danych, bez tworzenia i przekształcania obiektów.

Zewnętrze źródło danych

```
@Service
public class FakerService {
    private ArrayList<JSONObject> todayGames = new ArrayList<>();
    public void regenerate() throws JSONException {
        Faker faker = new Faker();
        todayGames.clear();
        for (int i = 0; i < 10; i++) {
            JSONObject oJsonInner = new JSONObject();
            oJsonInner.put("firstTeam", faker.team().name());
            oJsonInner.put("firstPoints", faker.number().randomDigitNotZero());
            oJsonInner.put("secondTeam", faker.team().name());
            oJsonInner.put("secondPoints", faker.number().randomDigitNotZero());
            oJsonInner.put("sport", faker.team().sport());
            todayGames.add(oJsonInner);
}}}
```

Obiekt **Faker** - wykorzystamy go do generowania danych.

Zewnętrze źródło danych

```
@Service
public class FakerService {
    private ArrayList<JSONObject> todayGames = new ArrayList<>();
    public void regenerate() throws JSONException {
        Faker faker = new Faker();
        todayGames.clear();
        for (int i = 0; i < 10; i++) {
            JSONObject oJsonInner = new JSONObject();
            oJsonInner.put("firstTeam", faker.team().name());
            oJsonInner.put("firstPoints", faker.number().randomDigitNotZero());
            oJsonInner.put("secondTeam", faker.team().name());
            oJsonInner.put("secondPoints", faker.number().randomDigitNotZero());
            oJsonInner.put("sport", faker.team().sport());
            todayGames.add(oJsonInner);
}}}
```

Czyścimy listę - operacja ta pozwoli nam przeładować dane po określonym czasie.

Zewnętrze źródło danych

```
@Service
public class FakerService {
    private ArrayList<JSONObject> todayGames = new ArrayList<>();
    public void regenerate() throws JSONException {
        Faker faker = new Faker();
        todayGames.clear();
        for (int i = 0; i < 10; i++) {
            JSONObject oJsonInner = new JSONObject();
            oJsonInner.put("firstTeam", faker.team().name());
            oJsonInner.put("firstPoints", faker.number().randomDigitNotZero());
            oJsonInner.put("secondTeam", faker.team().name());
            oJsonInner.put("secondPoints", faker.number().randomDigitNotZero());
            oJsonInner.put("sport", faker.team().sport());
            todayGames.add(oJsonInner);
}}}
```

Tworzymy obiekt **`JSONObject`**

Zewnętrze źródło danych

```
@Service
public class FakerService {
    private ArrayList<JSONObject> todayGames = new ArrayList<>();
    public void regenerate() throws JSONException {
        Faker faker = new Faker();
        todayGames.clear();
        for (int i = 0; i < 10; i++) {
            JSONObject oJsonInner = new JSONObject();
            oJsonInner.put("firstTeam", faker.team().name());
            oJsonInner.put("firstPoints", faker.number().randomDigitNotZero());
            oJsonInner.put("secondTeam", faker.team().name());
            oJsonInner.put("secondPoints", faker.number().randomDigitNotZero());
            oJsonInner.put("sport", faker.team().sport());
            todayGames.add(oJsonInner);
}}}
```

Umieszczamy w obiekcie wygenerowane wartości.

Zewnętrze źródło danych

```
@Service
public class FakerService {
    private ArrayList<JSONObject> todayGames = new ArrayList<>();
    public void regenerate() throws JSONException {
        Faker faker = new Faker();
        todayGames.clear();
        for (int i = 0; i < 10; i++) {
            JSONObject oJsonInner = new JSONObject();
            oJsonInner.put("firstTeam", faker.team().name());
            oJsonInner.put("firstPoints", faker.number().randomDigitNotZero());
            oJsonInner.put("secondTeam", faker.team().name());
            oJsonInner.put("secondPoints", faker.number().randomDigitNotZero());
            oJsonInner.put("sport", faker.team().sport());
            todayGames.add(oJsonInner);
}}}
```

Dodajemy obiekt do listy.

Zewnętrze źródło danych

Uzupełniamy klasę serwisu o konstruktor wywołujący metodę generującą, oraz getter zwracający utworzoną listę:

```
FakerService() throws JSONException{
    this.regenerate();
}
public ArrayList<JSONObject> getTodayGames() {
    return todayGames;
}
```

Zewnętrze źródło danych

Aby zapewnić zmienność danych po upływie określonego czasu, uzupełnimy naszą metodę o dodatkową adnotację, która sprawi, że metoda wywoła się automatycznie po określonym czasie.

Dodajemy adnotację **`Scheduled`** do metody **`regenerate`**:

```
@Scheduled(fixedRate = 5000)
public void regenerate() throws JSONException {
    //wcześniejszy kod
    }
}
```

Atrybut **`fixedRate`** oznacza liczbę milisekund, po jakim metoda ma się automatycznie wywołać.

Dla poprawności działania musimy dodać adnotację **`@EnableScheduling`** dla klasy konfiguracji **`AppConfig`**.

Zewnętrze źródło danych

Ostatni krok to utworzenie kontrolera oraz dodanie w nim akcji:

```
@GetMapping(path= "/fake-today-games")
public String sample() {
    return fakerService.getTodayGames().toString();
}
```

Pamiętaj o poprawnym wstrzyknięciu serwisu do kontrolera.

Możemy w ten sposób wypełnić naszą bazę danych do testów.

Zewnętrze źródło danych

Przykład danych jakie zostaną zwrócone:

```
[
{
firstTeam: "Maine tigers",
firstPoints: 4,
secondTeam: "Washington lycanthropes",
secondPoints: 7,
sport: "hockey"
},
{
firstTeam: "Oregon penguins",
firstPoints: 7,
secondTeam: "Minnesota sons",
secondPoints: 6,
sport: "football"
},
...
```

Zadanie – punkty dostępowe

Zastanów się jakie punkty dostępowe powinna mieć Twoja aplikacja, a następnie je utwórz. Możesz wykorzystać opisaną bibliotekę **Faker**, utworzyć listę przechowywaną w pamięci lub tworzyć i zwracać dane bezpośrednio w akcji.

Przykładowe punkty jakie możesz utworzyć:

- lista krajów,
- lista lig,
- ligi z danego kraju,
- aktualne wyniki trwających spotkań.

Zewnętrze źródło danych

Zapoznaj się z metodami, które udostępnia serwis: https://apifootball.com– posiada on możliwość darmowego korzystania z **API**.

Dokumentację znajdziemy pod adresem:

https://apifootball.com/documentation/

Możesz w swojej aplikacji posiadać dwa lub więcej źródeł danych.

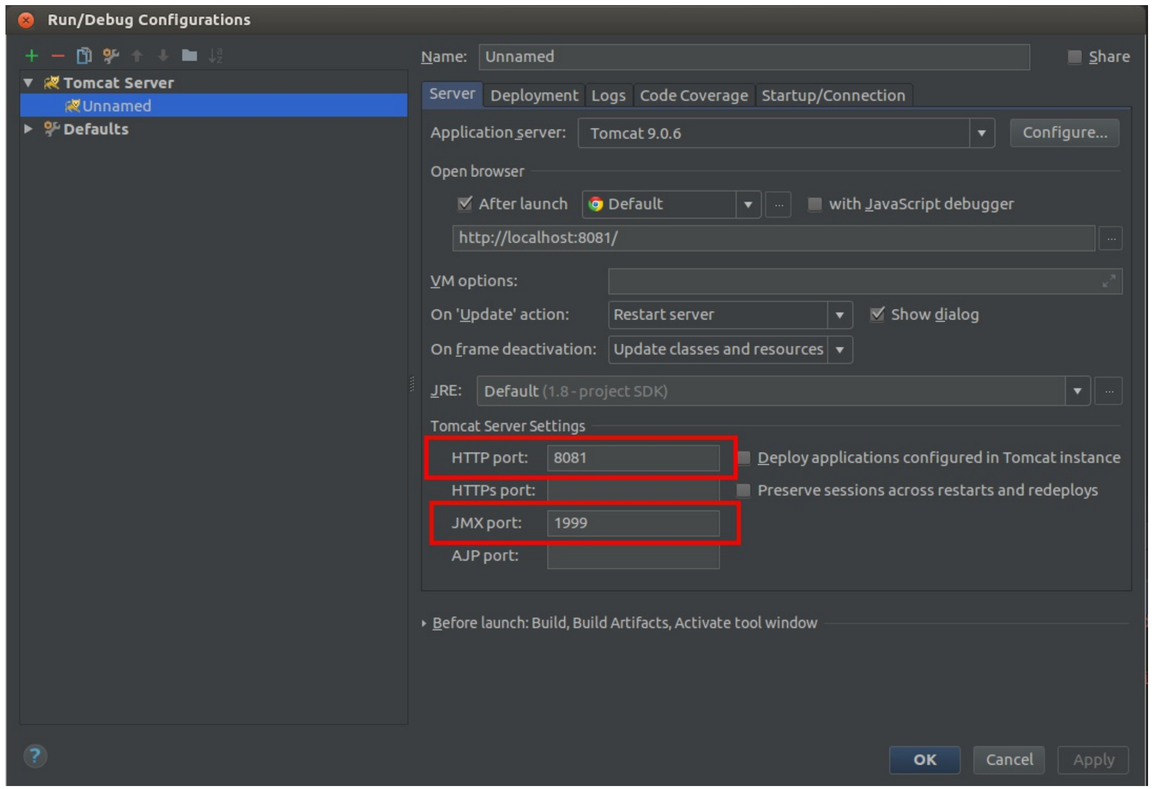
Zwróć uwagę, że różne systemy udostępniające swoje interfejsy w postaci **API** będą korzystać z różnych formatów danych.

Aplikacja korzystająca z **API**

Utwórz drugi projekt, który będzie korzystał z utworzonego wcześniej **API**.

Aby uruchomić jednocześnie dwa projekty musimy w jednym z nich zmienić porty serwera, na których jest on uruchomiony, np.:

- **HTTP port:** 8081
- **JMX port:** 1999



Aplikacja korzystająca z **API**

Aplikacja ta będzie łączyła się z **API**, pobierała i prezentowała odebrane informacje.

Dodaj zależności odpowiedzialne za **Spring MVC**.

Aplikację tę dodatkowo uzupełnimy o biblioteki do tworzenia logów oraz **`jackson-databind`**.

Tworzenie projektu

Uzupełniamy zależności odpowiedzialne za bibliotekę służącą do tworzenia logów:

```
<dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>jcl-over-slf4j</artifactId>
    <version>1.7.20</version>
</dependency>
<dependency>
    <groupId>ch.qos.logback</groupId>
    <artifactId>logback-classic</artifactId>
    <version>1.1.7</version>
</dependency>
```

Tworzenie projektu

W katalogu **resources** umieszczamy konfigurację w pliku o nazwie **logback.xml**:

```
<?xml version="1.0" encoding="UTF-8"?>
                    <configuration>
    <appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">
        <layout class="ch.qos.logback.classic.PatternLayout">
<Pattern>%d{yyyy-MM-dd HH:mm:ss} [%thread] %-5level %logger{36} - %msg%n</Pattern>
        </layout>
    </appender>
    <logger name="org.springframework" level="info" additivity="false">
        <appender-ref ref="STDOUT" /> </logger>
    <logger name="pl.coderslab" level="info" additivity="false">
        <appender-ref ref="STDOUT" /> </logger>
    <root level="debug">
        <appender-ref ref="STDOUT" /> </root>
    <root level="info">
        <appender-ref ref="STDOUT" /> </root>
</configuration>
```

Tworzenie projektu

Uzupełniamy zależności w pliku pom.xml:

```
<dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-databind</artifactId>
    <version>2.9.4</version>
</dependency>
```

Posłużymy się tą biblioteką do automatycznego przekształcenia danych w formacie **JSON** na obiekt.

Zadanie

Utwórz modele klas, które umożliwią przechowywanie informacji o:

- sportach (piłka nożna, koszykówka, hokej, itd.),
- krajach,
- ligach w danym kraju (I liga, II liga, itd.),
- zespołach w danej lidze (Arsenal, Liverpool F.C., itd.),
- rozgrywkach pomiędzy drużynami,
- wynikach rozgrywek,
- użytkownikach.

Utwórz powiązania, które powinny występować między klasami.

Przykład połączenia

Dla przykładu przedstawimy połączenie między klasami **Country** oraz **League**.

Klasa **Country**:

```
public class Country {
    private Long id;
    private String name;
    private Set<League> leagues;
// pozostałe atrybuty i metody
}
```

Przykład połączenia

Klasa **League**:

```
public class League {
    private long id;
    private String name;
    private Country country;
// pozostałe atrybuty i metody
}
```

Przykład wywołania metody **API**

Dodatkowo posłużymy się dodatkową klasą:

```
package pl.coderslab.web;
import com.fasterxml.jackson.annotation.JsonIgnoreProperties;
import com.fasterxml.jackson.annotation.JsonProperty;

@JsonIgnoreProperties(ignoreUnknown = true)
public class CountryDto {

    @JsonProperty("country_id")
    long apiCountryId;
    @JsonProperty("country_name")
    String name;
    //gettery, settery, toString
}
```

Przykład wywołania metody **API**

Dodatkowo posłużymy się dodatkową klasą:

```
package pl.coderslab.web;
import com.fasterxml.jackson.annotation.JsonIgnoreProperties;
import com.fasterxml.jackson.annotation.JsonProperty;

@JsonIgnoreProperties(ignoreUnknown = true)
public class CountryDto {

    @JsonProperty("country_id")
    long apiCountryId;
    @JsonProperty("country_name")
    String name;
    //gettery, settery, toString
}
```

Przy pomocy adnotacji określamy że elementy, które istnieją w **JSON** , ale nie istnieją w klasie, mają być ignorowane.

Przykład wywołania metody **API**

Dodatkowo posłużymy się dodatkową klasą:

```
package pl.coderslab.web;
import com.fasterxml.jackson.annotation.JsonIgnoreProperties;
import com.fasterxml.jackson.annotation.JsonProperty;

@JsonIgnoreProperties(ignoreUnknown = true)
public class CountryDto {

    @JsonProperty("country_id")
    long apiCountryId;
    @JsonProperty("country_name")
    String name;
    //gettery, settery, toString
}
```

Określamy nazwę parametru w **JSON** .

Przykład wywołania metody **API**

Tworzymy kontroler, w którym wywołamy połączenie do zewnętrznego serwisu.

W naszym przykładzie wykorzystamy metodę z apifootball:

```
@RestController
public class HelloController {
    private final Logger logger = LoggerFactory.getLogger(HelloController.class);
}
```

Przykład wywołania metody **API**

Tworzymy kontroler, w którym wywołamy połączenie do zewnętrznego serwisu.

W naszym przykładzie wykorzystamy metodę z apifootball:

```
@RestController
public class HelloController {
    private final Logger logger = LoggerFactory.getLogger(HelloController.class);
}
```

Tworzymy logger, który posłuży nam do wypisania pobranych danych na konsoli.

Przykład wywołania metody **API**

Metoda, z którą będziemy się łączyć to:

https://apifootball.com/api/?action=get_countries&APIkey=eee5028bd4f1a9645f0de3b18aa4c17c11a0eedd815aeaacf2cae4d5801e8969

Zwraca ona poniższe dane:

```
[
{
country_id: "169",
country_name: "England"
},
{
country_id: "173",
country_name: "France"
}
]
```

Przykład wywołania metody **API**

Dodajemy akcję, która wykona połączenie, pobierze i przekształci dane na nasze obiekty, a następnie wyświetli je.

```
@RequestMapping("/get-countries")
public String getCountriesAction() {
    String url = "https://apifootball.com/api/?action=get_countries&
APIkey=eee5028bd4f1a9645f0de3b18aa4c17c11a0eedd815aeaacf2cae4d5801e8969";
    RestTemplate restTemplate = new RestTemplate();
    ResponseEntity<CountryDto[]> responseCountries = restTemplate.getForEntity(
            url, CountryDto[].class);
    CountryDto[] countries = responseCountries.getBody();
    for (CountryDto country: countries) {logger.info("countries {}", country);}
    return "some result";
}
```

Przykład wywołania metody **API**

Dodajemy akcję, która wykona połączenie, pobierze i przekształci dane na nasze obiekty, a następnie wyświetli je.

```
@RequestMapping("/get-countries")
public String getCountriesAction() {
    String url = "https://apifootball.com/api/?action=get_countries&
APIkey=eee5028bd4f1a9645f0de3b18aa4c17c11a0eedd815aeaacf2cae4d5801e8969";
    RestTemplate restTemplate = new RestTemplate();
    ResponseEntity<CountryDto[]> responseCountries = restTemplate.getForEntity(
            url, CountryDto[].class);
    CountryDto[] countries = responseCountries.getBody();
    for (CountryDto country: countries) {logger.info("countries {}", country);}
    return "some result";
}
```

Adres **API** wraz z przykładowym kluczem.

Przykład wywołania metody **API**

Dodajemy akcję, która wykona połączenie, pobierze i przekształci dane na nasze obiekty, a następnie wyświetli je.

```
@RequestMapping("/get-countries")
public String getCountriesAction() {
    String url = "https://apifootball.com/api/?action=get_countries&
APIkey=eee5028bd4f1a9645f0de3b18aa4c17c11a0eedd815aeaacf2cae4d5801e8969";
    RestTemplate restTemplate = new RestTemplate();
    ResponseEntity<CountryDto[]> responseCountries = restTemplate.getForEntity(
            url, CountryDto[].class);
    CountryDto[] countries = responseCountries.getBody();
    for (CountryDto country: countries) {logger.info("countries {}", country);}
    return "some result";
}
```

Tworzymy obiekt klasy **RestTemplate**, którym posłużymy się w celu wywołania zapytania do **API**.

Przykład wywołania metody **API**

Dodajemy akcję, która wykona połączenie, pobierze i przekształci dane na nasze obiekty, a następnie wyświetli je.

```
@RequestMapping("/get-countries")
public String getCountriesAction() {
    String url = "https://apifootball.com/api/?action=get_countries&
APIkey=eee5028bd4f1a9645f0de3b18aa4c17c11a0eedd815aeaacf2cae4d5801e8969";
    RestTemplate restTemplate = new RestTemplate();
    ResponseEntity<CountryDto[]> responseCountries = restTemplate.getForEntity(
            url, CountryDto[].class);
    CountryDto[] countries = responseCountries.getBody();
    for (CountryDto country: countries) {logger.info("countries {}", country);}
    return "some result";
}
```

Odpytujemy serwis i dokonujemy automatycznej konwersji na obiekty naszego typu.

Przykład wywołania metody **API**

Dodajemy akcję, która wykona połączenie, pobierze i przekształci dane na nasze obiekty, a następnie wyświetli je.

```
@RequestMapping("/get-countries")
public String getCountriesAction() {
    String url = "https://apifootball.com/api/?action=get_countries&
APIkey=eee5028bd4f1a9645f0de3b18aa4c17c11a0eedd815aeaacf2cae4d5801e8969";
    RestTemplate restTemplate = new RestTemplate();
    ResponseEntity<CountryDto[]> responseCountries = restTemplate.getForEntity(
            url, CountryDto[].class);
    CountryDto[] countries = responseCountries.getBody();
    for (CountryDto country: countries) {logger.info("countries {}", country);}
    return "some result";
}
```

Pobieramy tablicę.

Przykład wywołania metody **API**

Dodajemy akcję, która wykona połączenie, pobierze i przekształci dane na nasze obiekty, a następnie wyświetli je.

```
@RequestMapping("/get-countries")
public String getCountriesAction() {
    String url = "https://apifootball.com/api/?action=get_countries&
APIkey=eee5028bd4f1a9645f0de3b18aa4c17c11a0eedd815aeaacf2cae4d5801e8969";
    RestTemplate restTemplate = new RestTemplate();
    ResponseEntity<CountryDto[]> responseCountries = restTemplate.getForEntity(
            url, CountryDto[].class);
    CountryDto[] countries = responseCountries.getBody();
    for (CountryDto country: countries) {logger.info("countries {}", country);}
    return "some result";
}
```

Wyświetlamy dane.

Dalsze prace

Zmodyfikuj aplikację tak, aby pobierane dane zapisywała do bazy danych.

Rozwiń według uznania model klas aplikacji.

Zapoznaj się z oficjalnym przewodnikiem jak pobierać dane **REST**:

https://spring.io/guides/gs/consuming-rest/