PATRYK KARBOWNIK
ANIELA KOSEK
ALEKSANDRA OKRUTNY
ZUZANNA SANTOROWSKA

KICKSTART DO TECHNOLOGII

Aniela Kosek

REACT

W ramach tego tutorialu zostanie utworzona prosta aplikacja React, napisana w TypeScript.

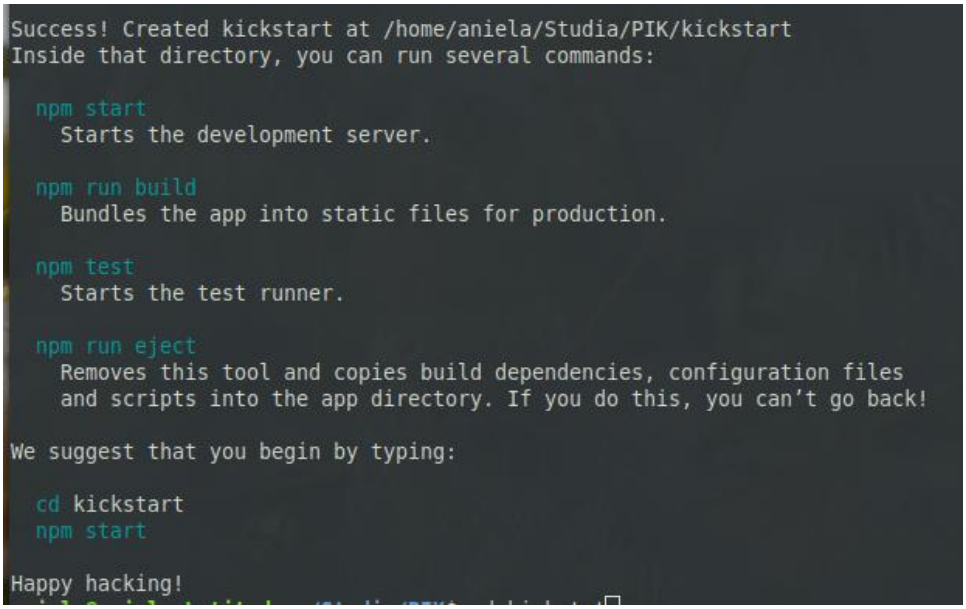
Aby utworzyć nowy projekt, należy najpierw zainstalować npm.

```
sudo apt install npm nodejs
```

Po tej operacji, aby stworzyć nową aplikację React, wystarczy użyć pakietu create-react-app, korzystając z szablonu dla aplikacji w TypeScript. Instalacja pakietów może potrwać kilka minut.

```
npx create-react-app kickstart --template typescript
```

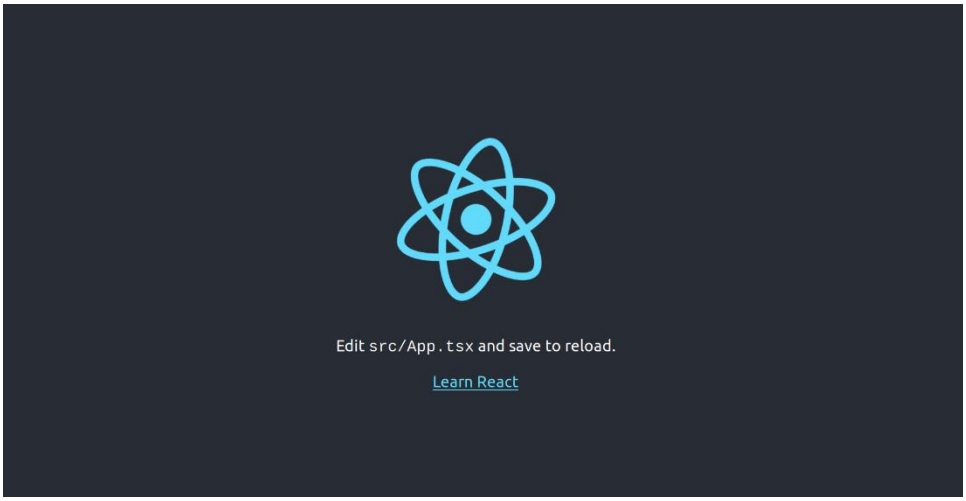
Po zakończeniu instalacji, create-react-app sam sugeruje najczęściej wykonywane operacje:



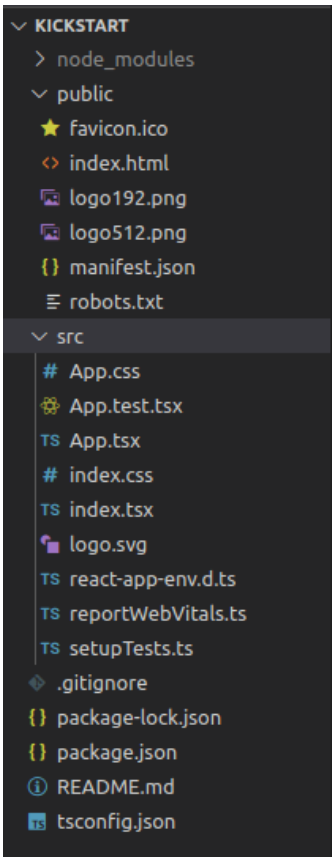
Podążając za sugestiami, uruchamiamy utworzoną aplikację.

```
cd kickstart
npm start
```

Nowoutworzona aplikacja wyświetla logo Reacta i odsyła do dokumentacji.



Po otworzeniu folderu w Visual Studio Code widzimy następującą strukturę projektu:



- Katalog node_modules zawiera wykorzystywane przez nas pakiety, zainstalowane przez npm.
- W katalogu public znajdziemy zasoby, wykorzystywane przez aplikację.
- Katalog src przechowuje nasze skrypty w języku TypeScript, a także pliki .css, definiujące style.
- Plik package-lock.json zawiera informacje o pakietach, znajdujących się w katalogu node_modules.
- Plik package.json jest miejscem, w którym dokonywana jest konfiguracja projektu.

Tutaj możemy zdefiniować nazwę i wersję tworzonej aplikacji. Parametr ”private” określa, czy aplikacja może zostać opublikowana np. w npm. Następnie definiowane są wersje pakietów, których wymaga projekt. ”scripts” określa skróty dla podstawowych operacji na projekcie, odpowiednio: uruchomienie, utworzenie pakietu do opublikowania, testowanie oraz publikacja.

Aby wykonać aplikację, najpierw usuwamy elementy wykorzystywane przez aplikację przykładową. Są to: favicon.ico, logo192.png, logo512.png.

Następnie modyfikujemy plik App.tsx, definiujący wygląd naszej aplikacji. Stworzymy prostą aplikację, która po wciśnięciu przycisku wyświetla napis “Witam serdecznie, jestem aplikacją, wyświetlającą napis, po wciśnięciu przycisku”.

```
{
  "name": "kickstart",
  "version": "0.1.0",
  "private": true,
  "dependencies": {
    "@testing-library/jest-dom": "^5.13.0",
    "@testing-library/react": "^11.2.7",
    "@testing-library/user-event": "^12.8.3",
    "@types/jest": "^26.0.23",
    "@types/node": "^12.20.15",
    "@types/react": "^17.0.11",
    "@types/react-dom": "^17.0.7",
    "react": "^17.0.2",
    "react-dom": "^17.0.2",
    "react-scripts": "4.0.3",
    "typescript": "^4.3.2",
    "web-vitals": "^1.1.2"
  },
  Debug
  "scripts": {
    "start": "react-scripts start",
    "build": "react-scripts build",
    "test": "react-scripts test",
    "eject": "react-scripts eject"
  },
  "eslintConfig": {
    "extends": [
      "react-app",
      "react-app/jest"
    ]
  },
  "browserslist": {
    "production": [
      ">0.2%",
      "not dead",
      "not op_mini all"
    ],
    "development": [
      "last 1 chrome version",
      "last 1 firefox version",
      "last 1 safari version"
    ]
  }
}
```

Ze zwracanego, przez funkcję App() JSX usuwamy zawartość i zastępujemy ją komponentem, który sami stworzymy.

```
import KickstartComponent from './kickstart.component';
import './App.css';

function App() {
  return (
    <div className="App">
      <KickstartComponent/>
    </div>
  );
}

export default App;
```

Teraz należy stworzyć wykorzystywany przez nas komponent. Do katalogu src dodajemy plik o nazwie kickstart.component.tsx I wypełniamy go następująco:

```
1   import React, { Component } from "react";
2
3
4   export default class KickstartComponent extends Component {
5
6       render() {
7           return (
8               <div>
9
10              </div>
11          );
12      }
13  }
```

Komponenty w React mogą być klasami lub funkcjami. Komponenty funkcyjne są prostsze w implementacji i wystarczają do nieskomplikowanych zadań. Klasy pozwalają między innymi na przechowywanie stanu, co wykorzystamy w tej aplikacji.

```
import React, { Component } from "react";
import Button from '@material-ui/core/Button';
import InputLabel from '@material-ui/core/InputLabel';


export default class KickstartComponent extends Component {

    render() {
        return (
            <div>
                <InputLabel></InputLabel>
                <Button> Click me! </Button>
            </div>
        );
    }
}
```

Wykorzystamy komponenty z biblioteki material-ui. Odpowiednie pakiety należy zainstalować:

`npm install @material-ui/core`

Kolejnym krokiem jest zdefiniowanie typu stanu i jego pierwotnej wartości. W tym przypadku stan ma tylko jedno pole, ale w bardziej rozbudowanych aplikacjach może być bardziej złożony.

```
 5   type State = {
 6       labelText: string;
 7   }
 8   let initialState: State = {
 9       labelText: ""
10   }
```

Teraz przyszła pora, by do KickstartComponent dodać pole state oraz konstruktor, który ustawi stan.

```
export default class KickstartComponent extends Component {

    state: State;

    constructor(props: any) {
        super(props);
        this.state = initialState;
    }

    render() {
        return (
            <div>
                <Button> Click me! </Button>
                <InputLabel>{this.state.labelText}</InputLabel>
            </div>
        );
    }
}
```

Nasza aplikacja na razie nic nie robi. Aby po wciśnięciu przycisku wykonywała się akcja, trzeba dodać funkcję handle.

```
handleClick = () => {
    this.setState({labelText: "Witam serdecznie! Jestem aplikacją wyświetlającą napis po wciśnięciu przycisku"});
}
```

I powiązać ją z przyciskiem.

```
<Button onClick={this.handleClick}> Click me! </Button>
```

Ostatecznie kod komponentu wygląda następująco:

```
import { Component } from "react";
import Button from '@material-ui/core/Button';
import InputLabel from '@material-ui/core/InputLabel';

type State = {
    labelText: string;
}
let initialState: State = {
    labelText: ""
}

export default class KickstartComponent extends Component {

    state: State;

    constructor(props: any) {
        super(props);
        this.state = initialState;
    }

    handleClick = () => {
        this.setState({labelText: "Witam serdecznie! Jestem aplikacją wyświetlającą napis po wciśnięciu przycisku"});
    }

    render() {
        return (
            <div>
                <Button onClick={this.handleClick}> Click me! </Button>
                <InputLabel>{this.state.labelText}</InputLabel>
            </div>
        );
    }
}
```

Po kolejnym użyciu komend:

```
npm install
```

```
npm start
```

W przegladarce otwiera się działająca aplikacja.

Po wciśnięciu przycisku:

Zuzanna Santorowska

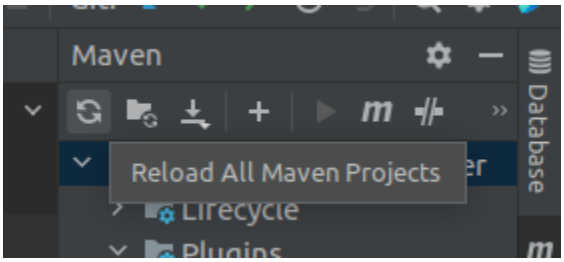
JACOCO

Jacoco to technologia pozwalająca na zmierzenie pokrycia kodu testami. Omówię jak należy skonfigurować mavena aby dało się sprawdzać pokrycie kodu testami.

Dodajemy plugin Jacoco w pliku pom.xml w <build><plugins>

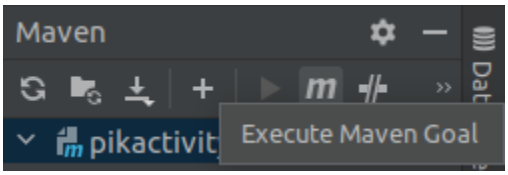
```
<plugin>
    <groupId>org.jacoco</groupId>
    <artifactId>jacoco-maven-plugin</artifactId>
    <version>0.8.6</version>

    <executions>
        <execution>
            <goals>
                <goal>prepare-agent</goal>
            </goals>
        </execution>
        <execution>
            <id>report</id>
            <phase>prepare-package</phase>
            <goals>
                <goal>report</goal>
            </goals>
        </execution>
    </executions>
</plugin>
```

Musimy przeładować Mavena jeśli zmeniamy plik pom.xml za pomocą pierwszej opcji

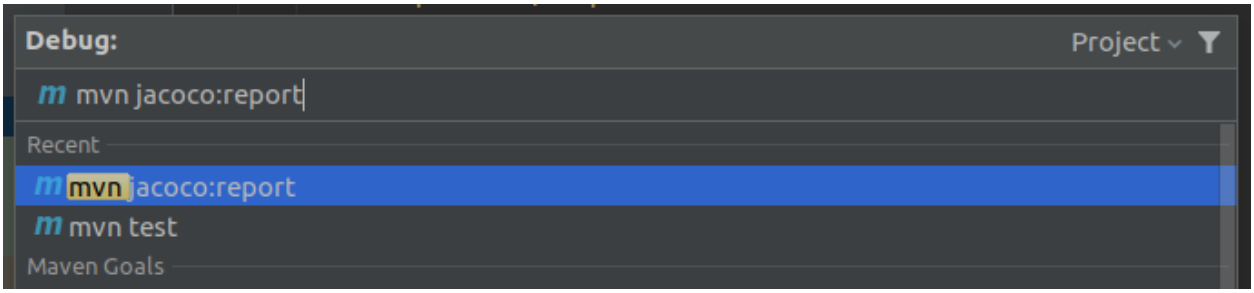


Następnie Odpalamy testy aby wygenerować plik jacoco.exec

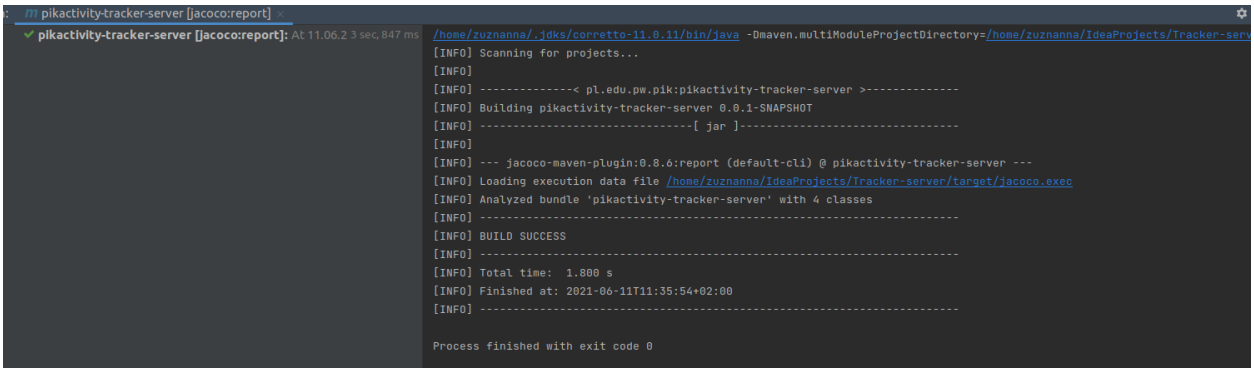
Następnie musimy wygenerować reprezentację graficzną testów, w tym celu wywołujemy mavena



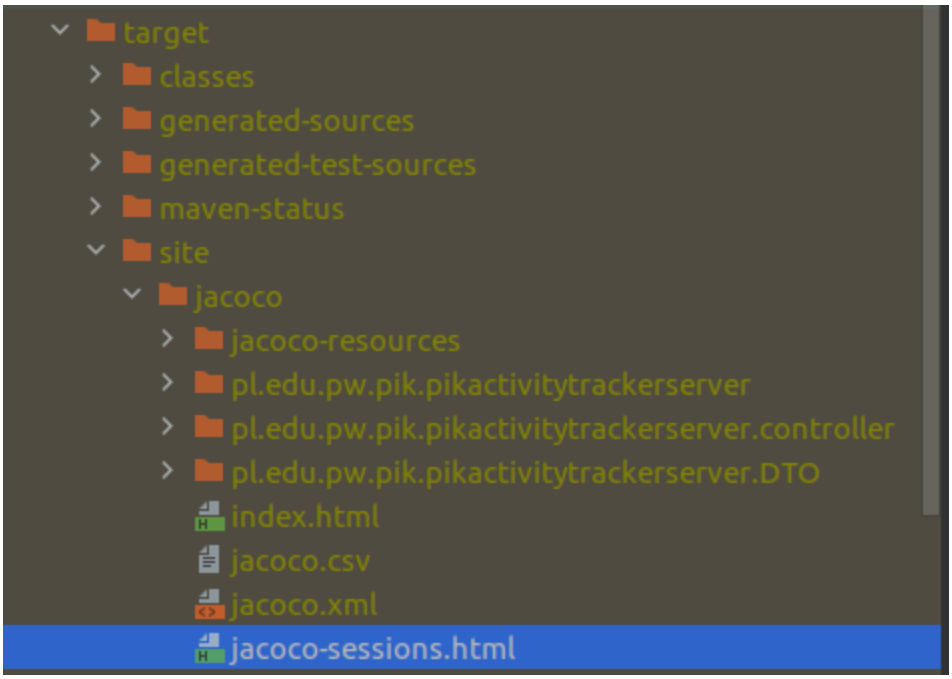
I wybieramy opcję mvn jacoco:report



Gdy otrzymamy komunikat że wszystko przebiegło zgodnie z planem



W target/site pojawią się wygenerowane raporty



Gdy odpalimy przykładowo plik jacoco-sessions.html pojawi nam się wygenerowana strona zawierająca jaka część kodu została pokryta testami.

Aleksandra Okrutny

Konfiguracja Jenkinsa w Dockerze, CI/CD dla aplikacji Java – Springboot

Konfigurację rozpoczynamy od utworzenia pliku Dockerfile, który posłuży do utworzenia obrazu Jenkinsa. Obraz ten zostanie uruchomiony w kontenerze dockerowym. Poniżej przedstawiono podstawową konfigurację pliku Dockerfile, która jest również dostępna na oficjalnej stronie Jenkinsa, pod adresem: https://www.jenkins.io/doc/book/installing/docker/

```
FROM jenkins/jenkins:2.263.4-lts
USER root
RUN apt-get update && apt-get install -y apt-transport-https \
      ca-certificates curl gnupg2 \
      software-properties-common

RUN curl -fsSL https://download.docker.com/linux/debian/gpg | apt-key add -
RUN apt-key fingerprint 0EBFCD88
RUN add-apt-repository \
      "deb [arch=amd64] https://download.docker.com/linux/debian \
      $(lsb_release -cs) stable"
RUN apt-get update && apt-get install -y docker-ce-cli
RUN curl https://cli-assets.heroku.com/install-ubuntu.sh | sh
USER jenkins
# Install Blue Ocean plugin since it doesn't seem to be available from the plugin manager
RUN jenkins-plugin-cli --plugins blueocean:1.24.3
```

Aby ułatwić zarządzanie Jenkinsem w kontenerze możemy skorzystać z mechanizmu docker-compose. W tym celu należy utworzyć plik docker-compose.yaml, który zawiera konfigurację obrazu Jenkinsa uruchamianego w kontenerze. Przykładowy plik docker-compose.yaml przedstawiono poniżej:

```
version: '2'

services:
  docker:
    image: docker:20.10.0-rc2-dind
```

```
    privileged: true
    ports:
      - "2376:2376"
    networks:
      - jenkins
    volumes:
      - jenkins-docker-certs:/certs/client
      - jenkins-data-real:/var/jenkins_home
    environment:
      DOCKER_TLS_CERTDIR: "/certs"
  jenkins:
    build:
      context: .
    restart: unless-stopped
    networks:
      - jenkins
    ports:
      - "1234:8080"
      - "50001:50000"
    volumes:
      - jenkins-docker-certs:/certs/client:ro
      - jenkins-data-real:/var/jenkins_home
      - /var/run/docker.sock:/var/run/docker.sock
    environment:
      DOCKER_HOST: "tcp://docker:2376"
      DOCKER_CERT_PATH: "/certs/client"
      DOCKER_TLS_VERIFY: "1"
```

Najważniejsze elementy pliku (pogrubione):

- image – wskazuje na repozytorium oraz wersję obrazu, która ma zostać pobrana do uruchomienia w kontenerze
- ports – pierwsza wartość wskazuje port na hoście, druga wartość jego odpowiednika w kontenerze
- volumes - pozwala nam na mapowanie katalogów pomiędzy kontenerem, a maszyną, chroniąc przed utratą wrażliwych danych w przypadku restartu kontenera
- build oraz context - wskazują na miejsce, gdzie umieszczony jest Dockerfile, który służy do zbudowania obrazu aplikacji

Aby uruchomić Jenkinsa w kontenerze należy wykonać następującą komendę:

```
docker-compose up –d
```

W przypadku gdy chcemy zbudować wyłącznie obraz dockerowy należy wykonać komendę:

```
docker build –t <nazwa_repozytorium>/<nazwa_obrazu>:<numer_wersji>
        -f <sciezka_do_dockerfile>
```

Do skonfigurowania pipeline'u CI/CD w Jenkinsie należy utworzyć plik Jenkinsfile i umieścić go w systemie wersji kontroli w root'cie projektu. Przykładowy plik Jenkinsfile został przedstawiony poniżej:

```
pipeline {
  agent any

  tools {
    maven "M3"
    jdk "openjdk-11"
  }
  environment {
    NEXUS_VERSION = "nexus3"
    NEXUS_PROTOCOL = "http"
    NEXUS_URL = "nexus:8081"
    NEXUS_REPOSITORY = "maven-snapshots"
    NEXUS_CREDENTIAL_ID = "nexus-user-credentials"
    HEROKU_CREDENTIALS_EMAIL = "heroku-user"
    HEROKU_CREDENTIALS_PASSWORD = "heroku-password"
  }

  stages {
    stage("Build")
    {
      steps
      {
        sh "mvn clean compile"
      }
    }
    stage("Test")
```

```
        {
            steps{
                sh "mvn test"
            }
        }
        stage("Package")
        {
            steps {
                sh "mvn package"
            }
        }
        stage('Deploy') {
            steps {
                sh "heroku container:push web -a activity-tracker-server"
                sh "heroku container:release web -a activity-tracker-server"
            }
        }
    }
}
```

Najważniejsze elementy pliku (pogrubione):

- Tools – sekcja odpowiadająca za dodanie narzędzi
- Environment – zmienne środowiskowe, które są dostępne globalnie w kontenerze
- Stages – kolejne etapy wykonywane podczas CI/CD
    - stage("Build") - odpowiada za skompilowanie projektu
    - stage("Test") - wykonuje testy
    - stage("Package") - tworzy plik wynikowy .jar
    - stage('Deploy') - łączy się i wypycha aktualną wersję projektu na serwer

Ostatnim krokiem jest utworzenie joba w panelu administratora, gdzie wystarczy wskazać adres pod jakim znajduje się repozytorium oraz branch, z którego chcemy wdrożyć aplikację.

Patryk Karbownik

KONFIGURACJA POŁĄCZENIA BACKENDU W SPRINGBOOT Z BAZAMI DANYCH

W opisie konfiguracji założono, że programista korzysta z narzędzia Maven do budowania projektu oraz, że wcześniej poprawnie skonfigurował bazy danych.

a) PostgreSQL

Na początku należy dodać odpowiednie zależności do pliku pom.xml

```xml
<dependency>
    <groupId>org.springframework.data</groupId>
    <artifactId>spring-data-jpa</artifactId>
    <version>2.1.8.RELEASE</version>
</dependency>
```

```xml
<dependency>
    <groupId>org.postgresql</groupId>
    <artifactId>postgresql</artifactId>
</dependency>
```

Pierwsza z nich dostarcza Java Persistence API, które bardzo upraszcza tworzenie części projektu odpowiedzialnej za komunikację z bazami danych. Z kolei druga zależność zawiera sterownik (driver) do bazy danych, bez którego jest niemożliwe wykonywanie zapytań oraz innych modyfikacji.

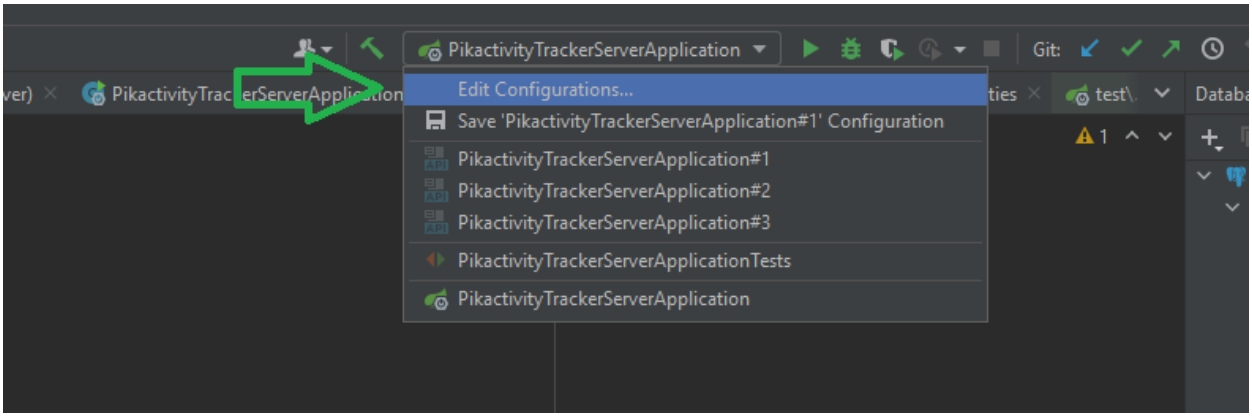
Mając zapewnione zależności możemy przystąpić do dostarczenia danych dostępowych do naszej bazy. Umieścimy je w pliku *application.properties.*

```
sql.datasource.jdbc-url=${POSTGRESQL_HOST}
sql.datasource.username=${POSTGRESQL_USER}
sql.datasource.password=${POSTGRESQL_PASSW}
sql.datasource.pool-size=30
```

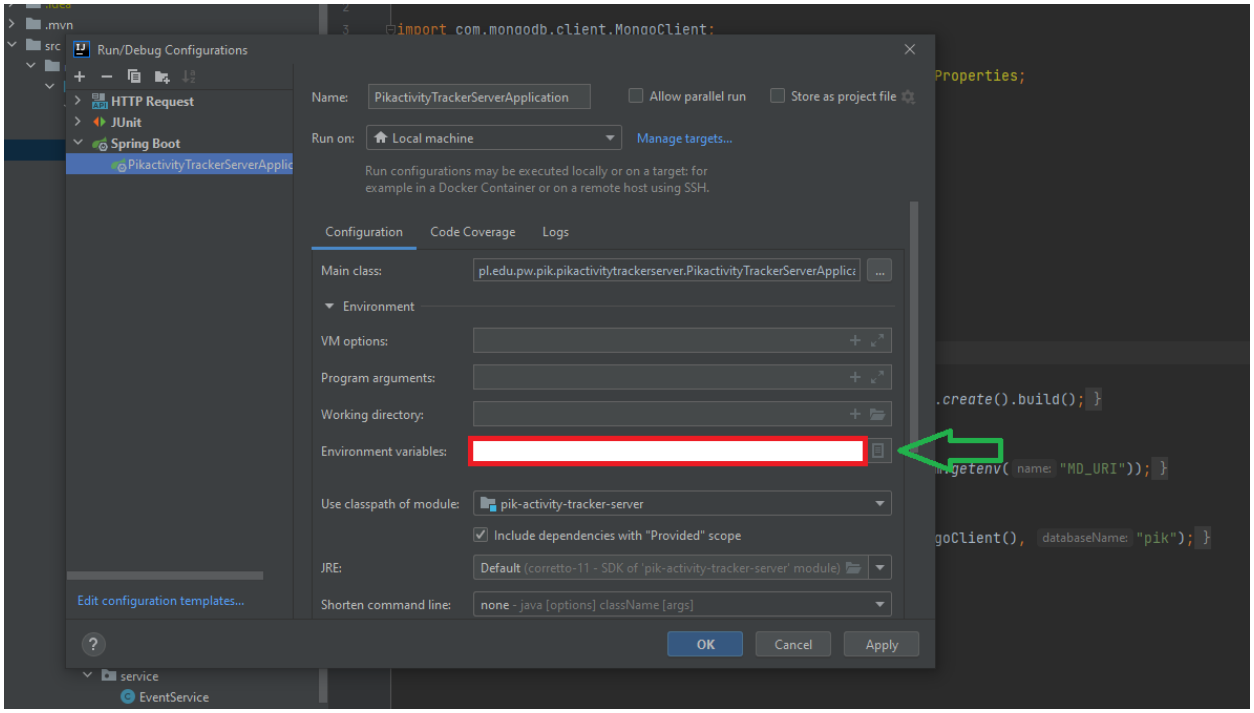
Nazwy jednoznacznie sugerują, co powinno być do nich przypisane. W tym przypadku skorzystaliśmy ze zmiennych środowiskowych. Jest to zalecane podejście, szczególnie gdy

nasz kod jest publicznie dostępny, ponieważ ogranicza to możliwość, że niepożądane osoby otrzymają dostęp do *credentials* naszej bazy.

Zmienne środowiskowe można bardzo łatwo ustawić w IDE. W przypadku IntelliJ należy:

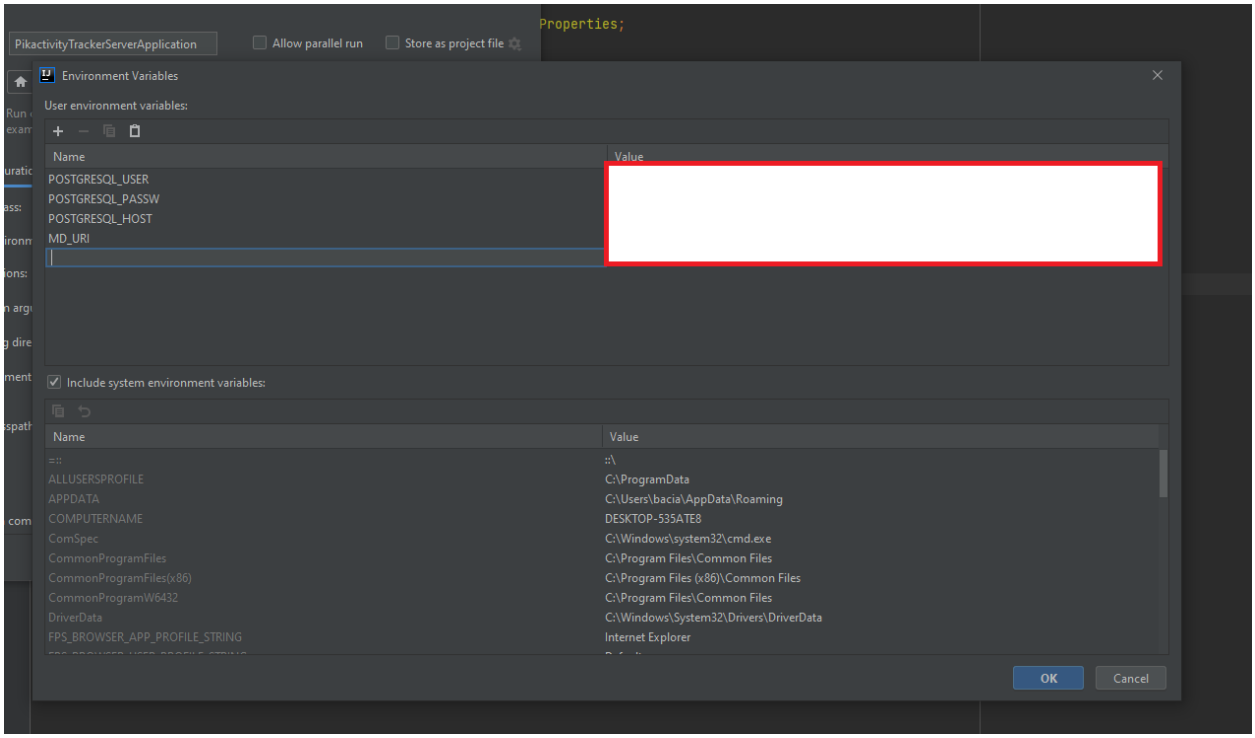


1. Rozwinąć pokazaną na obrazku listę i wybrać opcję *Edit Configurations…*



2. Wybrać opcję pokazaną na obrazku

3. Naszym oczom ukaże się lista zmiennych środowiskowych



Do której za pomocą przycisku + możemy dowolnie dodawać nowe pozycje.

Mając tak przygotowane zmienne oraz plik application.properties można przejść do następnego etapu konfiguracji, w którym utworzymy klasę oznaczoną adnotacją @Configuration.

Należy do wykonać tak, jak na załączonym zrzucie ekranu:

```java
package pl.edu.pw.pik.pikactivitytrackerserver.configurations;

import com.mongodb.client.MongoClient;
import com.mongodb.client.MongoClients;
import org.springframework.boot.context.properties.ConfigurationProperties;
import org.springframework.boot.jdbc.DataSourceBuilder;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Primary;
import org.springframework.data.mongodb.core.MongoTemplate;

import javax.sql.DataSource;

@Configuration(proxyBeanMethods = true)
public class SqlDataSourceConfig {
    @Bean
    @Primary
    @ConfigurationProperties("sql.datasource")
    public DataSource getDataSource() { return DataSourceBuilder.create().build(); }

    public @Bean
    MongoClient mongoClient() { return MongoClients.create(System.getenv( name: "MD_URI")); }

    public @Bean
    MongoTemplate mongoTemplate() { return new MongoTemplate(mongoClient(), databaseName: "pik"); }

}
```

Tutaj bardzo ważne jest, aby wartość w adnotacji @ConfigurationProperties pokrywała się z przedrostkiem, którego używaliśmy do definiowania *credentials*. Dzięki czemu SpringBoot będzie wiedział, które z nich wykorzystać, aby stworzyć obiekt do komunikacji z bazą

b) MongoDB

W tej części przedstawię, jak ustawić połączenie z bazą typu MongoDB korzystając z *MongoTemplate*, co jest ziemlich konieczne, gdy chcemy dodawać własne kolekcje (w porównanie do wykorzystania MongoRepository).

W tym wypadku nie będziemy korzystać z ustawiania *credentials* w pliku *application.properties*.

Należy stworzyć zmienną środowiskową zawierającą całe URI do bazy, tak jak to pokazano dla bazy typu PostgreSQL.

Następnie w klasie konfiguracyjnej należy utworzyć dwie metody oznaczone adnotacją @Bean:

```
public @Bean
MongoClient mongoClient() { return MongoClients.create(System.getenv("MD_URI")); }

public @Bean
MongoTemplate mongoTemplate() { return new MongoTemplate(mongoClient(), "pik"); }
```

Pierwsza tworzy klienta Mongo, a druga służy do wykonywania zapytań do bazy. Bardzo ważne jest to, że w konstruktorze MongoTemplate należy podać nazwę bazy na serwerze. Z kolei w metodzie create klasy MongoClients musimy podać kompletną ścieżkę do bazy