Politechnika Lubelska

Wydział Elektrotechniki i Informatyki

Katedra Informatyki

Laboratorium: Programowanie aplikacji w chmurze obliczeniowej.

Temat projektu: Forum internetowe.

Autorzy:

1. Patryk Iwanowski
2. Michał Goluch

Grupa: IO 6.2

Rok: 3

Tryb studiów: stacjonarne

Lublin, 02.06.2022 r.

1. PROJEKT APLIKACJI – OPIS ORAZ FUNKCJONALNOŚCI

Tematem tworzonej aplikacji internetowej jest forum internetowe wspomagające wyszukiwanie odpowiedzi oraz wspólne dyskusje na różne tematy (np. z zakresu programowania) za pomocą tworzenia postów oraz umieszczania odpowiedzi oraz dodatkowych pytań do dyskusji w komentarzach. Aplikacja ma być prosta w użyciu, czytelna i intuicyjna a ponadto powinna spełniać większość postawionych poniżej założeń.

1.1. Wymagania funkcjonalne:

Użytkownik:

1. Rejestracja w serwisie
2. Logowanie do serwisu za pomocą loginu i hasła
3. Możliwość wylogowania z serwisu
4. Zmiana e-maila oraz nazwy użytkownika
5. Możliwość dodania nowego posta
6. Możliwość sprawdzenia utworzonych przez siebie postów za pomocą odpowiedniej zakładki
7. Możliwość usunięcia utworzonego posta
8. Możliwość przejrzenia listy wszystkich postów w serwisie.
9. Możliwość przejrzenia komentarzy i szczegółów każdego posta z osobna.
10. Możliwość zostawienia komentarza pod postem
11. Możliwość usunięcia swojego komentarza.

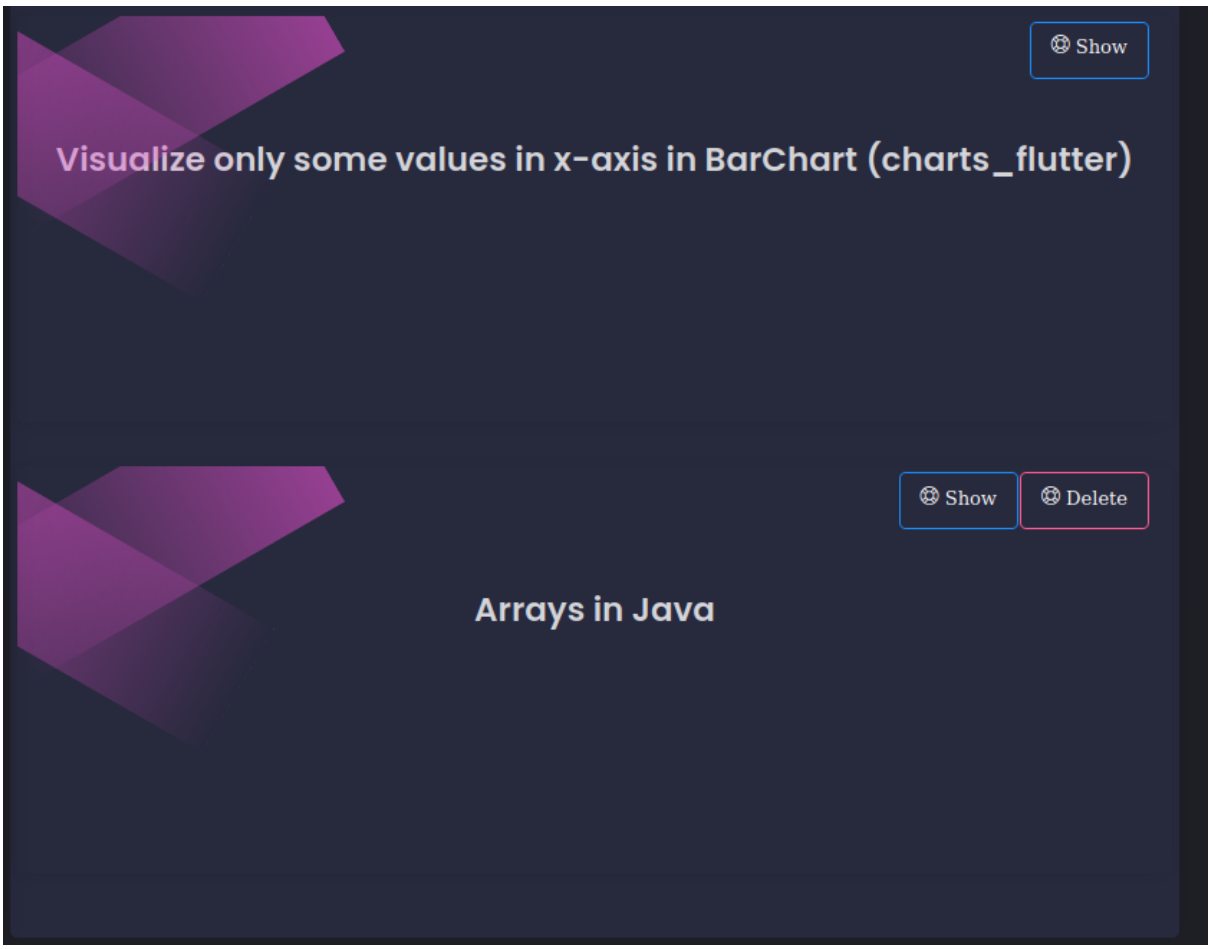
Administrator:

1. Wszystkie funkcjonalności Użytkownika
2. Możliwość usuwania postów i komentarzy bez względu kto je stworzył.

1.2. Wymagania niefunkcjonalne:

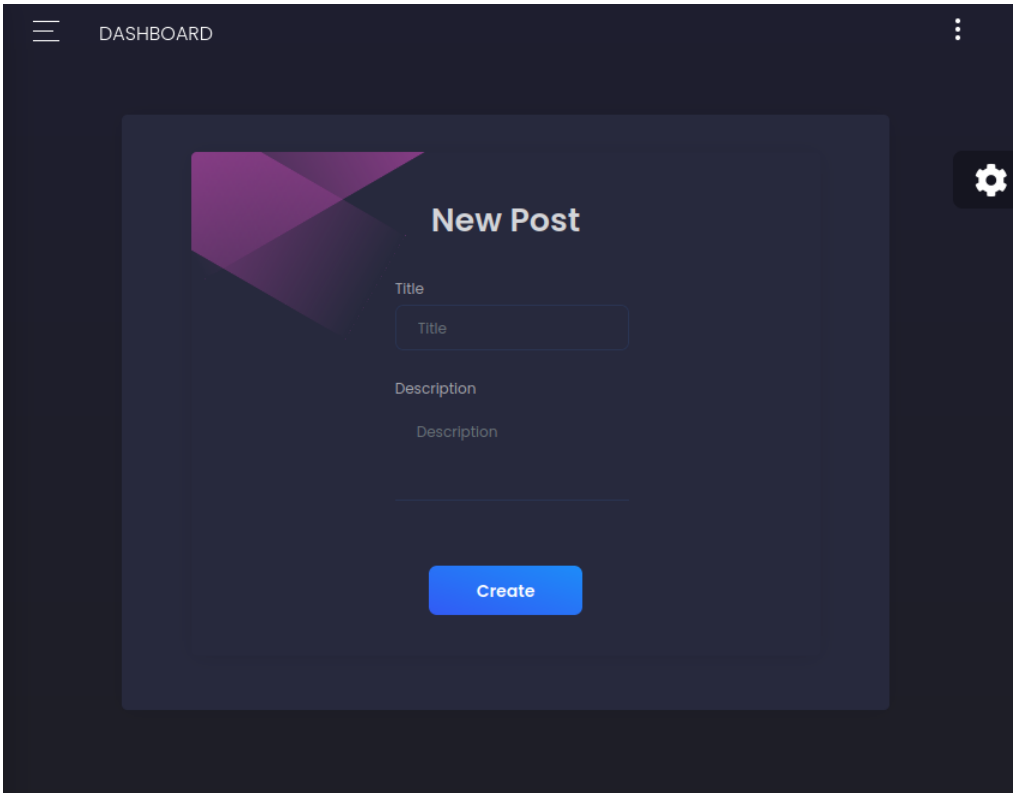
1. Dostęp do serwisu ma odbywać się za pomocą przeglądarki internetowej.
2. Interfejs musi być czytelny i intuicyjny.
3. System ma zdefiniowane role dla użytkowników
4. Interfejs powinien być responsywny.
5. System musi zawierać obsługę błędów, walidację oraz odpowiednie zabezpieczenia.
6. System musi działać na kontenerach Docker.

2. PROJEKT APLIKACJI – PRZEDSTAWIENIE INTERFEJSÓW



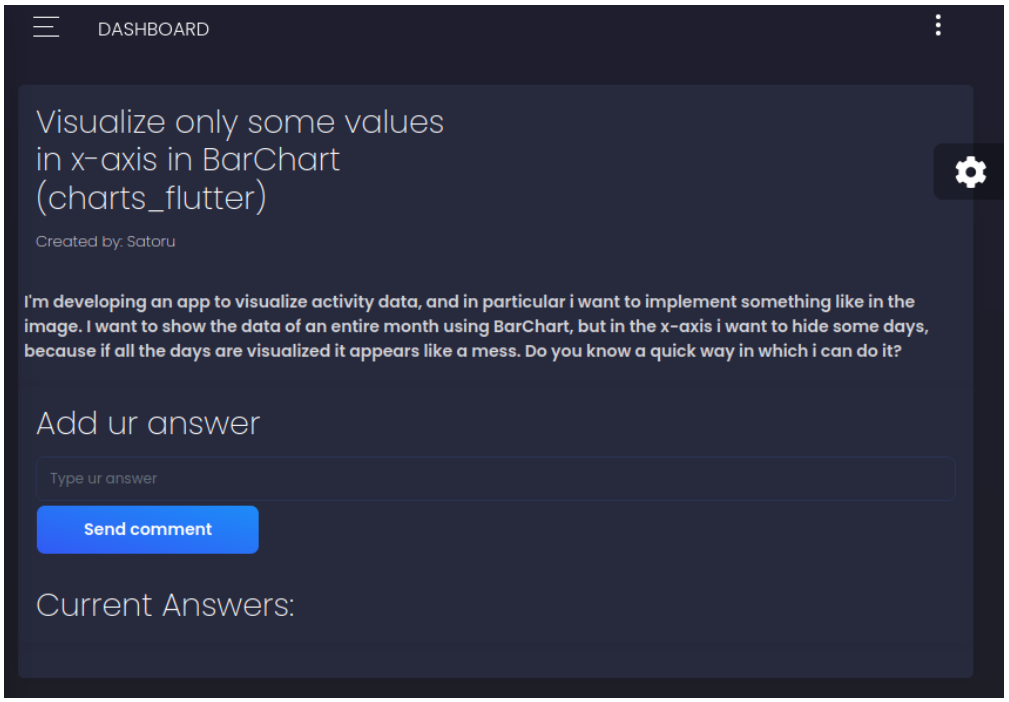
Rys.2.1. Widok menu głównego aplikacji.

Aby korzystać z aplikacji konieczne jest założenie konta oraz zalogowanie się. Po wykonaniu procedury logowania użytkownik ma możliwość przeglądania i komentowania postów innych użytkowników oraz tworzenia i usuwania własnych postów.



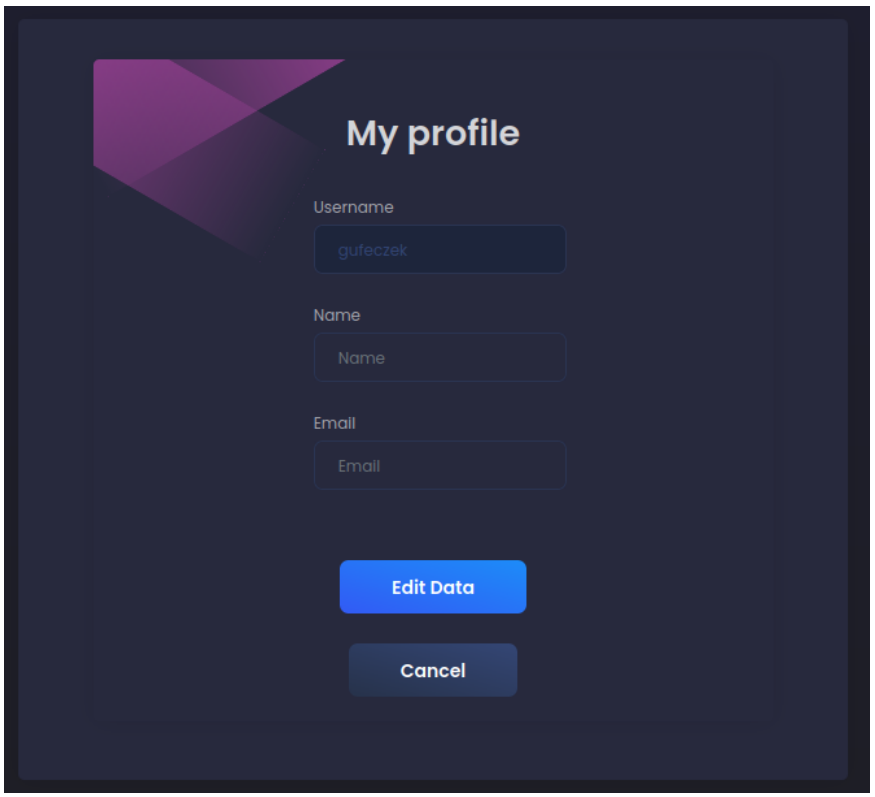
Rys.2.2. Widok tworzenia nowego postu.

Podczas tworzenia posta wymagane jest podanie jego tytułu oraz opisu problemu.



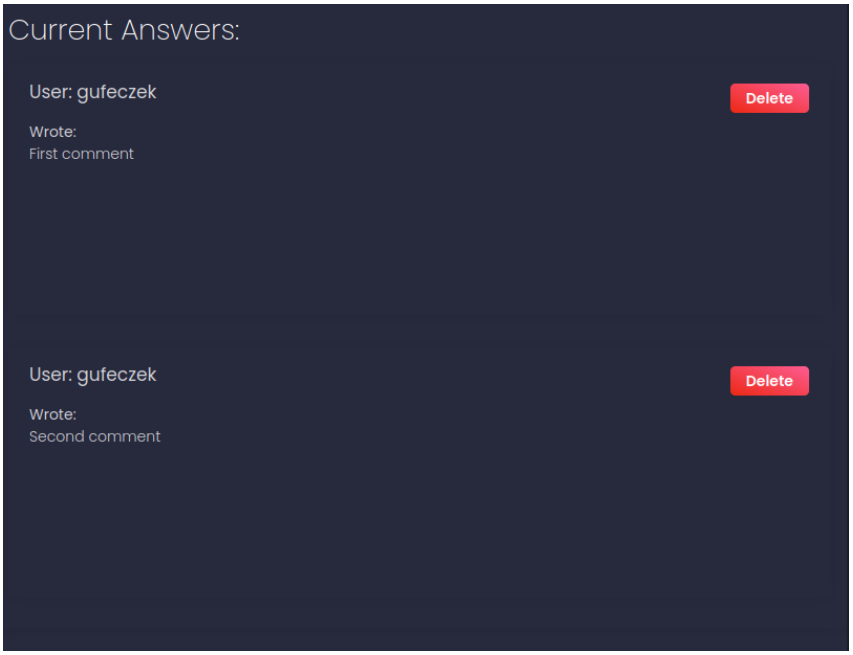
Rys.2.3. Widok podglądu posta.

Po wejściu w podgląd posta możliwe jest przeczytanie jego opisu, dodawanie własnej odpowiedzi oraz wyświetlenie odpowiedzi innych użytkowników.



Rys.2.4. Widok edycji danych konta.

Istnieje również możliwość edytowania danych konta.



Rys.2.5. Widok komentarzy posta na koncie z uprawnieniami Administratora.

Konto administratora posiada uprawnienie usuwania postów oraz komentarzy innych użytkowników.

3. WYKORZYSTANA TECHNOLOGIA

3.1. Spring Framework

Do zrealizowania aplikacji po stronie serwera zdecydowaliśmy się wykorzystać język Java z użyciem Spring Framework. Spring Framework początkowo został stworzony jako alternatywne rozwiązanie dla płatnej Javy Enterprise Edition. Umożliwia on tworzenie aplikacji webowych w technologiach MVC i REST(technologia wykorzystana w naszym projekcie). Jest rozbudowany i zawiera szereg gotowych implementacji i bibliotek ułatwiających wykorzystanie i redukujących ilość pisanego kodu.

W tym projekcie wykorzystane zostały takie elementy jak:

- Spring Boot - odpowiadający za automatyczną konfigurację aplikacji i bezproblemowe uruchomienie.
- Spring Security - odpowiadający za kwestie związane z bezpieczeństwem m.in autoryzacja i uwierzytelnienie.
- Spring Data - odpowiadająca za zarządzanie danymi poprzez definiowanie repozytoriów jako interfejsy.

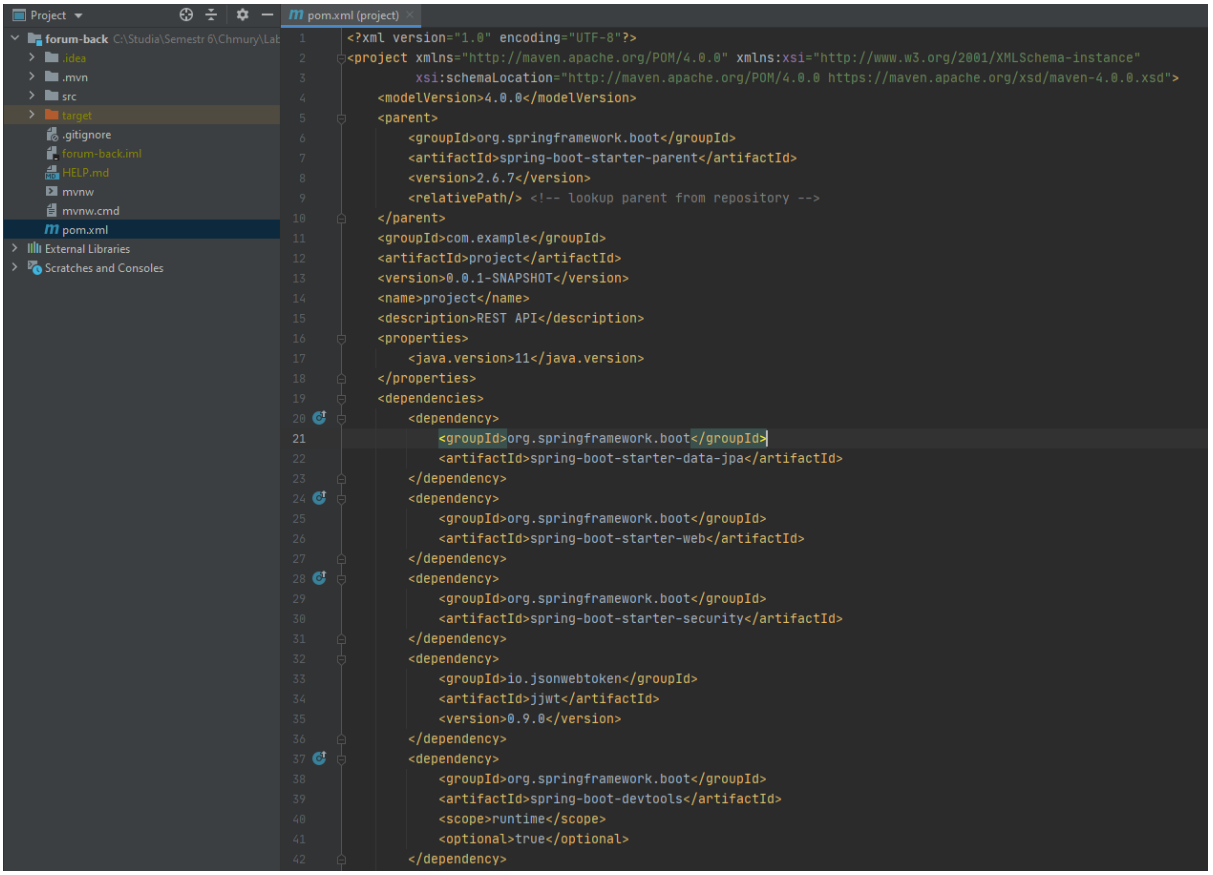
Zdecydowaliśmy się wybrać ten framework ze względu na dużą ilość możliwości oraz w dość prostą implementację funkcjonalności. Ponadto jest on z jedną z najczęściej wybieranych technologii a jej dokumentacja jest ogólnodostępna i przejrzysta.

3.2. Hibernate

Hibernate jest najczęściej wykorzystywanym frameworkiem w projektach Javy do zarządzania bazą danych. Wykorzystuje mapowanie obiektowo-relacyjne(ORM), które polega na odwzorowywaniu obiektów klas Javy na obiekty bazy danych. Wybraliśmy go, ponieważ pozwala on w łatwy i szybki sposób zbudować strukturę potrzebnej bazy danych bez bezpośredniej ingerencji w wybrane środowisko bazodanowe. Ponadto umożliwia on łatwą do konfigurację i tworzenie relacji. Przykładem takiej relacji w naszym projekcie jest relacja @ManyToOne w przypadku obiektu klasy User i Post. Oznacza to, że użytkownik może posiadać wiele postów natomiast post może należeć tylko do jednego użytkownika. Wykorzystanie technologii jaką jest Hibernate pozwoliło nam na czytelniejsze i wygodniejsze skonstruowanie niezbędnej do funkcjonowania aplikacji bazy danych.

3.3. Maven

Maven jest to narzędzie pomagające w budowaniu i zarządzaniu projektami. Konfiguracja zawarta jest w pliku pom.xml gdzie w czytelny sposób przedstawione są dodane do aplikacji zależności. W przypadku kiedy chcemy dodać potrzebną nam zależność możemy to zrobić w każdej chwili poprzez dopisanie (za pomocą odpowiedniego schematu) do pliku niezbędnych elementów a następnie wciśnięcie przycisku w prawym górnym rogu (IntelliJ) w celu przeładowania zależności, po tym Maven automatycznie wyszuka, pobierze i zaimplementuje dodane zależności Przykładowy wygląd takiego pliku został zamieszczony poniżej (Rys 2.1).



Rys.3.1. Zawartość pliku pom.xml odpowiadającego za konfigurację projektu z wykorzystaniem Maven

3.4. Lombok

Lombok jest zależnością którą dodajemy do pom.xml tak jak w przypadku Hibernate czy innych mniejszych technologii. Następnie za pomocą Mavena implementowany jest on w naszym projekcie. Lombok ułatwia i przyspiesza pisanie aplikacji oraz zwiększa czytelność kodu ze względu na jego ograniczanie. Sposobem który oferuje nam Lombok są adnotacje, dzięki którym jesteśmy w stanie automatycznie wygenerować konstruktory, gettery i settery dla naszej klasy. Jedną z ważniejszych adnotacji z której tutaj korzystamy jest @Builder, który pozwala na tworzenie instancji klasy w bardzo prosty sposób (Rys. 2.2) .

Na rysunku 2.3 przedstawione zostały przykładowe adnotacje wykorzystane w klasie “Post”:

- @Getter - automatycznie tworzy gettery dla wszystkich pól klasy
- @Setter - automatycznie tworzy settery dla wszystkich pól klasy
- @AllArgsConstructor - automatycznie tworzy konstruktor z wszystkimi parametrami
- @NoArgsConstructor - automatycznie tworzy konstruktor bezparametrowy
- @Builder - implementuje wzorzec projektowy builder, pozwalający na stworzenie obiektu klasy.

```
User u2 = User.builder()
        .username("User")
        .email("user@gmail.com")
        .password(passwordEncoder.encode( rawPassword: "user"))
        .build();
```

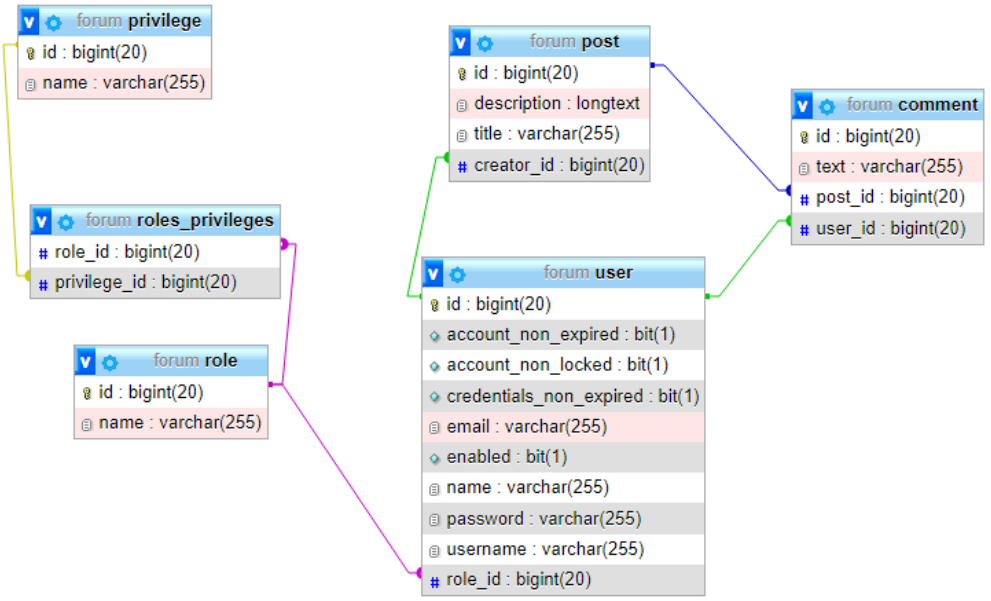
Rys.3.2. Przykład wykorzystania adnotacji @Builder.

```
@AllArgsConstructor
@NoArgsConstructor
@Builder
@Setter
@Getter
@Entity
public class Post {
```

Rys.3.3. Adnotacje klasy Post.

3.5.MySQL

MySQL to system zarządzający relacyjnymi bazami danych rozwijany jako oprogramowanie open-source. Jest on uniwersalnym systemem kompatybilnym z większością dostępnych technologii na rynku. Pozwala przechowywać duże ilości danych, łączyć je za pomocą relacji a także zapewnia swego rodzaju zabezpieczenia dotyczące ochrony danych które są dla użytkownika konfigurowalne np. tryb cascade. W aplikacji wykorzystany został tryb Cascade.REMOVE powodujący usunięcie wszystkich komentarzy należących do posta w momencie usunięcia samego posta. Schemat ERD bazy danych aplikacji przedstawiony jest na rysunku 2.4

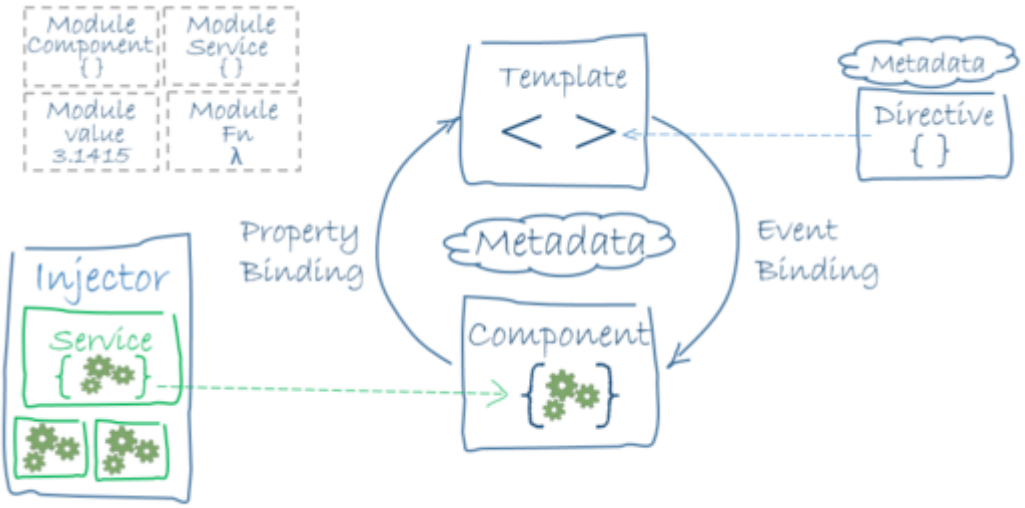


Rys.3.4. Diagram ERD bazy danych

3.6. Angular

W przypadku serwera po stronie klienta wykorzystany został framework o nazwie Angular. Jest to kompleksowy framework do projektowania oraz tworzenia wydajnych aplikacji typu SPA (Single Page Application). Pozwala na tworzenie aplikacji z wykorzystaniem języka TypeScript. Architektura Angular oparta jest na komponentach zorganizowanych w NgModules. Każda aplikacja Angular składa się z modułu głównego, umożliwiającego ładowanie początkowe oraz wielu modułów funkcji.

Poniższy diagram przedstawia w uproszczeniu schemat działania wykorzystanego frameworka:



Rys.3.5. Diagram przedstawiający uproszczony schemat działania wykorzystanego frameworka.

3.7. Bootstrap

Bootstrap to framework wykorzystujący HTML, CSS i JavaScript udostępnionych na licencji MIT. Służy on do tworzenia responsywnych stron internetowych przeznaczonych na każdego rodzaju urządzenia. Zawiera wiele gotowych implementacji pozwalajacych na łatwe i wygodne zarządzanie wyglądem HTML za pomocą nadawania tagom odpowiednich klas. Zastosowany w projekcie widok jest przerobionym widokiem darmowego szablonu Angular 12 + Bootstrap 4.

3.8. Docker

Docker jest to oprogramowanie pozwalające na stworzenie własnego środowiska kontenerowego z wykorzystaniem plików Dockerfile, obrazów z repozytorium DockerHub lub narzędzia Docker Compose. Stosowanie konteneryzacji usług pozwala użytkownikom w bardzo szybki i wygodny sposób korzystać z aplikacji składającej się z wielu usług za pomocą zaledwie dwóch poleceń. Zapobiega to problemom z ewentualnymi niekompatybilnościami wersji danego oprogramowania lub innymi zależnościami. Wykorzystane w tym projekcie zostały zarówno pliki Dockerfile, obrazy z DockerHub jak i narzędzie Docker Compose. Dzięki temu byliśmy w stanie stworzyć 3 kontenery współpracujące ze sobą które świadczą usługi oferowane przez naszą aplikację. Poniżej przedstawione są pliki Dockerfile dla backendu i frontendu a także plik docker-compose.yml składający wszystko w całość.

Na rysunku 3.6 przedstawiona została zawartość pliku Dockerfile dla aplikacji po stronie klienta. Całość kontenera opiera się na systemie node:alpine pobranym z repozytorium DockerHub. Plik znajduje się w folderze głównym aplikacji z którego kopiowane są wszystkie niezbędne pliki.

Na początku ustawiany jest katalog docelowy oraz zmienna środowiskowa konfigurująca openssl. Następnie kopiowany jest plik package.json definiujący wszystkie zależności niezbędne do uruchomienia aplikacji. W kolejnych 2 wierszach wywoływane są polecenia instalujące angulara oraz wszystkie niezbędne zależności zawarte w pliku package.json. Po zainstalowaniu wszystkich bibliotek kopiowana jest cała zawartość folderu app.
W kolejnej linii wystawiany jest port 4200 który jest domyślny portem dla aplikacji budowanych na szkielecie Angular. Kolejne polecenie powoduje nadanie odpowiednich uprawnień do projektu dla usera o nazwie node, który jest tworzony za pomocą polecenia “USER node”. Na samym końcu mamy polecenie odpowiadające za uruchomienie aplikacji Angulara.

```
FROM node:alpine

WORKDIR '/app'
ENV NODE_OPTIONS=--openssl-legacy-provider
COPY package.json .
RUN npm install -g @angular/cli
RUN npm install --legacy-peer-deps

COPY . .
EXPOSE 4200

RUN chown -R node /app/node_modules

USER node

CMD ng serve --host 0.0.0.0 -c production
```

Rys.3.6. Zawartość pliku Dockerfile dla aplikacji frontend

Na rysunku 3.7 przedstawiony został plik Dockerfile dla aplikacji po stronie serwera napisanej w Javie z wykorzystaniem Spring Framework. Tak jak w przypadku aplikacji po stronie klienta, całość opiera się na obrazie pobranym z DockerHub. W tym przypadku jest to openjdk:11 (po prostu platforma Java).

Analizując po kolei polecenia możemy zauważyć, że na początku ustawiany jest katalog docelowy oraz kopiowane są foldery mvn a także pom.xml odpowiadający za jego konfigurację. Kolejne dwa polecenia polegają na nadaniu uprawnień dla skopiowanego tam wcześniej katalogu a także pobranie do niego wszystkich bibliotek i zależności zdefiniowanych w pliku pom.xml (2 polecenie ./mvnw dependency:go-offline). Następnie kopiowane są pliki źródłowe projektu oraz uruchamiane za pomocą Maven'a oraz Spring Boota.

```
FROM openjdk:11-jdk-slim
WORKDIR /app
COPY .mvn/ .mvn
COPY mvnw pom.xml ./
RUN chmod +x ./mvnw
RUN ./mvnw dependency:go-offline

COPY src ./src

CMD ["./mvnw", "spring-boot:run"]
```

Rys.3.7. Zawartość pliku Dockerfile dla aplikacji backend

Rysunek 3.8 przedstawia zawartość pliku Docker-compose.yml. Jak można zauważyć całość usługi składa się z 3 serwisów, jest to:

- db - baza danych utworzona na bazie obrazu mysql pobranego z DockerHub
- backendserver - aplikacja serwerowa stworzone na bazie przedstawionego powyżej Dockerfile
- frontend - aplikacja po stronie klienta utworzona na bazie przedstawionego powyżej Dockerfile

Analizując nieco głębiej konfigurację serwisów widzimy, że w przypadku bazy danych ustawiane są zmienne środowiskowe odpowiadające za nadanie nazwy oraz hasła roota dla tworzonej bazy. Ponadto podpięty został wolumen dzięki któremu jesteśmy w stanie przechowywać dane w folderze na maszynie hosta. Kolejne ustawienia to wystawienie i mapowanie portów. Jak widać poniżej wystawiony został port 3306 oraz zmapowany został port 3306 na 3306 co jest podstawowym portem w przypadku baz mysql.

Przechodząc do analizy konfiguracji serwisu backendserver możemy zauważyć że posiada on konfigurację zmiennych środowiskowych dla Spring'a. Zmienna ta odpowiada za ustawienie źródła danych dla aplikacji. W tym przypadku adres został podany jako zależność od db na porcie 3306 (zamiast localhost:3306 mamy db:3306). Powoduje to, że aplikacja bez problemu jest w stanie znaleźć odpowiedni adres na którym znajduje się baza danych o nazwie forum. W kolejnych konfiguracjach widzimy mapowanie portów tj z 8080 na 8080 oraz ustawienie zależności serwisu backendserver od serwisu db.

Ostatni serwis o nazwie frontend skonfigurowany jest w bardzo prosty sposób. Posiada on zadeklarowane wolumeny, pozwalające na korzystanie z danych zawartych na maszynie hosta. Dodatkowo ma ustawione mapowanie portów typowych dla aplikacji Angular tj. 4200 na 4200.

```yaml
version: '3.7'
services:
  db:
    image: 'mysql'
    environment:
           - MYSQL_ROOT_PASSWORD=password
           - MYSQL_DATABASE=forum
    volumes:
           - ./my-db:/var/lib/mysql
    ports:
           - 3306:3306
    expose:
           - 3306
  backendserver:
    build: ./backend/
    environment:
      SPRING_DATASOURCE_URL: jdbc:mysql://db:3306/forum?createDatabaseIfNotExists=true&autoReconnect=true
    ports:
           - "8080:8080"
    links:
           - "db"
    depends_on:
           - "db"
  frontend:
    build:
      context: frontend
      dockerfile: Dockerfile
    volumes:
           - /app/node_modules
           - ./frontend:/app
    ports:
           - "4200:4200"
```

Rys.3.8. Zawartość pliku docker-compose.yml

4. SPOSOBY ROZWIĄZANIA FUNKCJONALNOŚCI

Przedstawione w pierwszej części funkcjonalności zostały stworzone z wykorzystaniem podstawowych rozwiązań stosowanych w tworzeniu aplikacji internetowych oraz aplikacji zbudowanych z wykorzystaniem Spring Framework oraz Angular. Przykładem implementacji jednej z funkcjonalności tj. logowania lub rejestracji jest generowanie tokenu JWT (Rys.3.1), przesłanie go w headerze do serwera klienta gdzie następnie jest umieszczany w session storage. Na podstawie otrzymanego tokenu stosowane są zabezpieczenia umożliwiające określenie kiedy użytkownik jest zalogowany(serwer przesłał token dla serwera klienta).

Na podstawie tokena ustawione są również przekierowania do odpowiednich stron. Reszta funkcjonalności opiera się na podobnym schemacie. Cała aplikacja została stworzona w technologii REST co oznacza, że po stronie serwera mamy wystawione endpointy zwracające odpowiednie do postawionego zadania dane(obiekty)(Rys.3.2). W momencie kiedy użytkownik wykona jakąś czynność po stronie klienta (Rys.3.6) wysłane zostaną żądania przypisane do odpowiednich elementów na stronie odpowiadające ich funkcjonalnościom. Poglądowy schemat działania aplikacji znajduje się na rysunku 3.4.

Przy każdym przesłaniu żądania w headerze umieszczany jest obecny w session storage token. W przypadku kiedy serwer otrzyma wraz z żądaniem token który nie przejdzie filtrowania (tj. np. token użytkownika o username którego nie ma w bazie) żądanie zostanie odrzucone a serwer zwróci błąd o kodzie 401 oznaczającym brak autoryzacji. Wszystko to odbywa się z wykorzystaniem Spring Security pozwalającym w szybki i łatwy sposób zabezpieczyć wszystkie endpointy z wykluczeniem tych które są użytkownikowi potrzebne bez autoryzacji tj. logowania i rejestracji (Rys.3.3).

```java
import java.util.Date;

@Slf4j
@Component
public class JwtTokenUtil {

    @Value("https://www.youtube.com/watch?v=dQw4w9WgXcQ")
    private String jwtSecret;

    @Value("heh")
    private String jwtIssuer;

    public String generateAccessToken(User user) {
        Claims claims = Jwts.claims().setSubject(user.getUsername());

        return buildToken(user.getUsername(), claims);
    }

    private String buildToken(String subject, Claims claims) {
        return Jwts.builder()
                .setSubject(subject)
                .setClaims(claims)
                .setIssuer(jwtIssuer)
                .setIssuedAt(new Date())
                .setExpiration(new Date(System.currentTimeMillis() + 7 * 24 * 60 * 60 * 1000)) // 1 week
                .signWith(SignatureAlgorithm.HS512, jwtSecret)
                .compact();
    }

    public String getUsername(String token) {
        Claims claims = Jwts.parser()
                .setSigningKey(jwtSecret)
                .parseClaimsJws(token)
                .getBody();
```

Rys.3.1. Kawałek kodu z generowania token'u JWT.

```java
@RestController
@RequestMapping("/api/v1/posts")
public class PostController {
    private final PostService postService;

    public PostController(PostService postService) { this.postService = postService; }

    @GetMapping
    public ResponseEntity<PostDTO> getUserByName(@RequestParam String name) {
        return ResponseEntity.ok(postService.getPostByName(name));
    }

    @GetMapping("/all")
    public ResponseEntity<List<PostDTO>> getAllPosts() { return ResponseEntity.ok(postService.getAllPosts()); }

    @PostMapping("/add")
    public ResponseEntity<?> createPost(@RequestBody PostDTO postDTO){
        return ResponseEntity.ok(postService.save(postDTO));
    }

    @GetMapping("/{id}")
    public ResponseEntity<PostDTO> getPostById(@PathVariable Long id){
        return ResponseEntity.ok(postService.getPostById(id));
    }

    @PutMapping("/{id}")
    public ResponseEntity<?> updatePost(@PathVariable Long id, @RequestBody PostDTO postDTO){
        return ResponseEntity.ok(postService.updatePostByDTO(id, postDTO));
    }

    @DeleteMapping("/{id}")
    public ResponseEntity<Void> deletePostById(@PathVariable Long id){
        postService.deletePostById(id);
        return new ResponseEntity<>(HttpStatus.OK);
    }

    @PostMapping("/newComment")
    public ResponseEntity<Void> addComment(@RequestBody CommentDTO commentDTO) {
        postService.addComment(commentDTO);
        return new ResponseEntity<>(HttpStatus.OK);
    }
```

Rys.3.2. Przykład wystawionych endpointów po stronie serwera dla PostController.

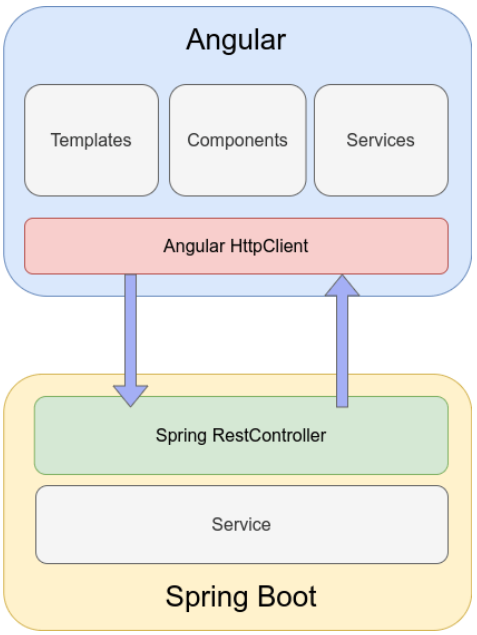
```java
@RequiredArgsConstructor
@Configuration
@EnableWebSecurity
@EnableGlobalMethodSecurity(prePostEnabled = true)
public class SecurityConfig extends WebSecurityConfigurerAdapter {

    private final JwtTokenFilter jwtTokenFilter;

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.authorizeRequests() ExpressionUrlAuthorizationConfigurer<H>.ExpressionInterceptUrlRegistry
                .antMatchers( ...antPatterns: "/api/v1/auth/login","/api/v1/auth","/api/v1/auth/new-account","/h2-console/**").permitAll()
                .anyRequest().authenticated()
                .and().httpBasic() HttpBasicConfigurer<HttpSecurity>
                .and().sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATELESS) SessionManagementConfigurer<HttpSecurity>
                .and().exceptionHandling() ExceptionHandlingConfigurer<HttpSecurity>
                .authenticationEntryPoint(
                        (request, response, ex) -> response.sendError(HttpServletResponse.SC_UNAUTHORIZED, ex.getMessage()))
                .and().headers().frameOptions().sameOrigin() HeadersConfigurer<HttpSecurity>
                .and().cors().and().csrf().disable();

        http.addFilterBefore(jwtTokenFilter, UsernamePasswordAuthenticationFilter.class);
    }
```

Rys.3.3. Konfiguracja elementów autoryzacji Spring Security.



Rys.3.4.Rysunek przedstawiający opisany w skrócie powyżej schemat

```typescript
@Injectable({
  providedIn: 'root'
})
export class PostService {
  private baseUrl = 'http://localhost:8080/api/v1';

  constructor(private httpClient: HttpClient) { }

  getPosts(): Observable<Post[]> {

    return this.httpClient.get<Post[]>( url: this.baseUrl+'/posts/all');
  }


  deleteComment(commentId: number) {
    const url = this.baseUrl + '/posts/comment/'+commentId;
    return this.httpClient.delete(url);
  }

  showPostContent(postId:number):Observable<Post>{
    return this.httpClient.get<Post>( url: this.baseUrl+'/posts/' + postId)
  }

  addPost(newPost: Post) {
    const headers = {'content-type': 'application/json'}
    const body = JSON.stringify(newPost);
    return this.httpClient.post<Post>( url: this.baseUrl + '/posts/add',body, options: {headers});
  }

  addComment(newComment: Comment){
    const headers = {'content-type': 'application/json'}
    const body = JSON.stringify(newComment);
    return this.httpClient.post<Comment>( url: this.baseUrl+'/posts/newComment',body, options: {headers});
  }
  removePost(postId: number) {
    const url = this.baseUrl+'/posts/' + postId

    return this.httpClient.delete(url);
```

Rys.3.6. Rysunek przedstawiający serwisy obsługujące wysyłanie żądań do serwera.

5. PODSUMOWANIE

Podsumowując efekt stworzonej aplikacji możemy stwierdzić, że w pełni spełniliśmy założenia wstępne i w całości zaimplementowaliśmy wszystkie przedstawione przez nas wymagania, zarówno funkcjonalne jak i niefunkcjonalne. Aplikacja jest intuicyjna i czytelna, pozwala na zmianę koloru tła a także elementów kolorystyki. Serwis pozwala na tworzenie kont użytkownika lecz ich nazwy nie mogą się powtarzać i muszą przejść odpowiednią walidację (zostało to odpowiednio zabezpieczone oraz wyświetlane są odpowiednie komunikaty).

Zalogowany użytkownik może korzystać z serwisu tj., tworzyć, usuwać oraz wyświetlać utworzone przez niego posty, dodawać oraz usuwać swoje komentarze a także zmienić niektóre elementy jego profilu.

Jeżeli używamy konta Administratora obok naszej nazwy użytkownika w kolorze pomarańczowym wyświetlany jest napis *(Admin)*. Administrator serwisu posiada wszystkie funkcjonalności użytkownika wraz z dodatkowymi odpowiadającymi jego roli tzn. możliwość usunięcia dowolnego postu lub komentarza w obrębie całego serwisu.

Aplikacja korzysta z kontenerów Docker oraz narzędzia Docker Compose, dzięki czemu bez problemu możliwe jest wdrożenie systemu na dowolnym urządzeniu z zainstalowanym oprogramowaniem Docker.