КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ІМЕНІ ТАРАСА ШЕВЧЕНКА
Факультет комп'ютерних наук та кібернетики
Кафедра теоретичної кібернетики

Кваліфікаційна робота
на здобуття ступеня бакалавра
за спеціальністю 122 Комп'ютерні науки
на тему:

Розробка веб додатку відео зв'язку на реактивному підході

Виконав студент 4-го курсу
Баранець Артемій Геннадійович _____
(підпис)

Науковий керівник:
доцент каф-ри ТТП _____Волохов В.М
(підпис)

Засвідчую, що в цій дипломної роботі немає запозичень з праць інших авторів без відповідних посилань.
Студент _____
(підпис)

Роботу розглянуто й допущено до захисту на засіданні кафедри теоретичної кібернетики
«____» _____ 2021 р., протокол №___

Завідувач кафедри теоретичної кібернетики
докт. фіз.-матю наук, професор _____ Ю. В. Крак

Київ – 2021

РЕФЕРАТ

Обсяг роботи 74 сторінки, 10 джерел посилань.

Назва роботи: Розробка веб додатку відео зв’язку на реактивному підході.

Це дослидження для вивчення і розуміння, що таке реактивне програмування і як воно працює, навіщо воно потрібно, в моему випадкі для написання реактивно-асинхронного сервера відео дзвінків з підтримкою відправки повідомлень\файлів. Також підтримкою анонімних дзвінків, які працюють на новому реактивному протоколі. Цей продукт можна використовувати як бізнес проект, як сервіс на заміну «Zoom» чи «Skype». Він буде швидше та вимагати менше продуктивності.

Мета роботи: зрозуміти як працює реактивне програмування, навчитися писати реактивні сервіси\контролери\вебсокет, розробити програму, яка буде максимально реактивна та асинхронна.

План роботи: розібратися що таке реактивне програмування на Java, створити реактивний захист, асинхронність обробки запитів. Розібратися що таке реактивний вебсокет, та за допомогою його створити обробку команд. Розібратися в новому протоколі SSE, який буде використовуватись для анонімних кімнат.

Результат роботи: робочий сервер для сервісу відео дзвінків. В якій можна створювати чати, запрошувати туди інших користувачів, обмінювати повідомленнями та файлами розмовляти по відео зв’язку. Програма вміє створювати анонімні кімнати, які не потребують реєстрації, де можна також розмовляти по відео зв’язку.

Зміст

ВСТУП

Реактивне програмування - один из самых актуальных трендов современности. це асинхронність, поєднана з потокогивої обробкою даних. Тобто якщо в асинхронної обробки немає блокувань потоків, але дані обробляються однаково порціями, то реактивність додає можливість обробляти дані потоком. Загалом реактивне програмування можна використовувати для написання скидних високо напружених проектів, реалізації складної бізнес логіки.

Зараз настав такий час коли складно представити життя без відео дзвінків та чатів. Більшість популярних додатків написані на застарілих підходах та технологіях. Це мене наштовхнуло написати свою якісну реалізацію такого сервісу.

Можливими сферами застосовування мого додатку є:

- Дистанційні уроки у школі\університетах;
- Дистанційні бізнес зустрічі;
- Звичайне чатове спілкування.

РОЗДІЛ 1. ЩО ТАКЕ РЕАКТИВНЕ ПРОГРАМУВАННЯ НА JAVA

1.1 Реактивне програмування

Приклад, коли начальник доручає завдання Васі, той повинен передати результат Дімі, а Діма повернути начальнику? Але у нас завдання - це якась порція, і поки вона не буде зроблена, далі передати її не можна. Такий підхід дійсно розвантажує начальника, але Петя і Вася періодично простоюють, адже Петя треба дочекатися результатів роботи Васі, а Васі - дочекатися нового завдання.

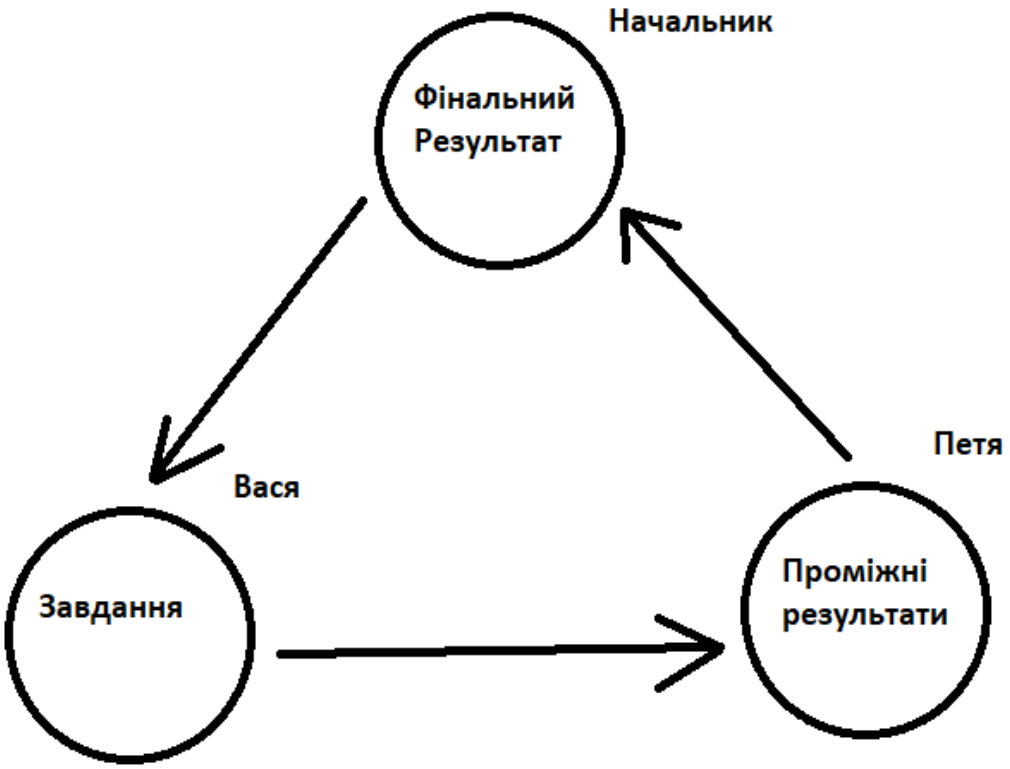

Рисунок 1.1 – Приклад взаємодії

А тепер уявіть, що завдання розбили на безліч підзадач. І тепер вони пливуть безперервним потоком:

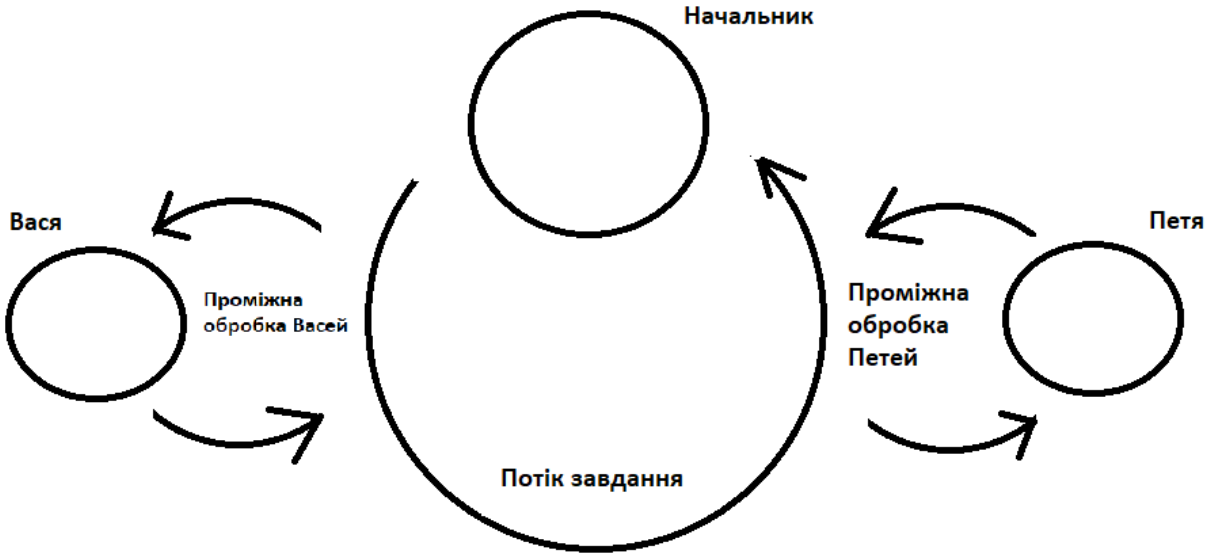

Рисунок 1.2 – Приклад реактивної взаємодії

Кажуть, коли Генрі Форд придумав свій конвеєр, він підвищив продуктивність праці в чотири рази, завдяки чому йому вдалося зробити автомобілі доступними. Тут ми бачимо те ж саме: у нас невеликі порції даних, а конвеєр з потоком даних, і кожен обробник пропускає через себе ці дані, якимось чином їх перетворюючи. Як Васі і Діми у нас виступають потоки виконання (threads), забезпечуючи, таким чином, многопоточную обробку даних.

Java 9/11	Reactive Streams
Java 8	Completable Future
Java 7	Fork\Join framework
Java 5	Executor framework
Java 1	Threads

Таблиця 1.1 – реактивність у різних версіях

На цій таблиці показані різні технології розпаралелювання, що додавалися в Java в різних версіях. Як ми бачимо, специфікація Reactive Streams на вершині - вона не замінює всього, що було до неї, але додає найвищий рівень абстракції, а значить її використання просто і ефективно.

Ідея реактивності побудована на патерні проектування Observer:

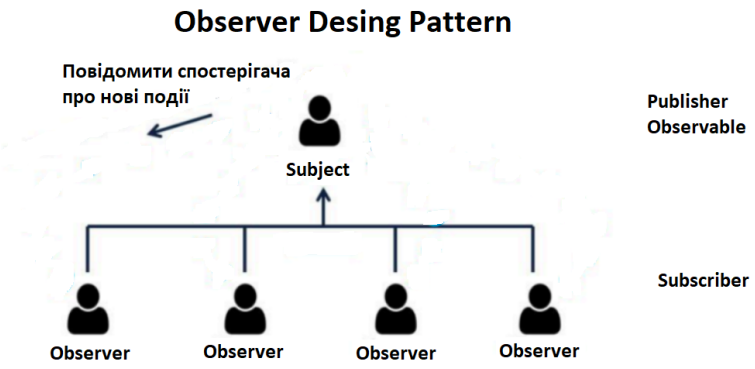

Рисунок 1.3 – Патерн Observer

У нас є підписники і ті, на що підписуються. Підписатися на якусь подію, а потім отримувати оновлення. Після підписки, як тільки з'являється нове повідомлення, всім підписникам приходить notify, тобто повідомлення. В даній схемі є:

- Publisher - той, хто публікує нові повідомлення;
- Observer - той, хто на них підписаний. У реактивних потоках підписник зазвичай називається Subscriber. Терміни різні, але по суті це одне і те ж. У більшості спільнот більш звичні терміни Publisher / Subscriber.

Один з життєвих прикладів реактивності - система оповіщення при пожежі. Припустимо, нам треба зробити систему, що включає тривогу в разі перевищення задимленості і температури.

1.2 Приклади застосування реактивного програмування

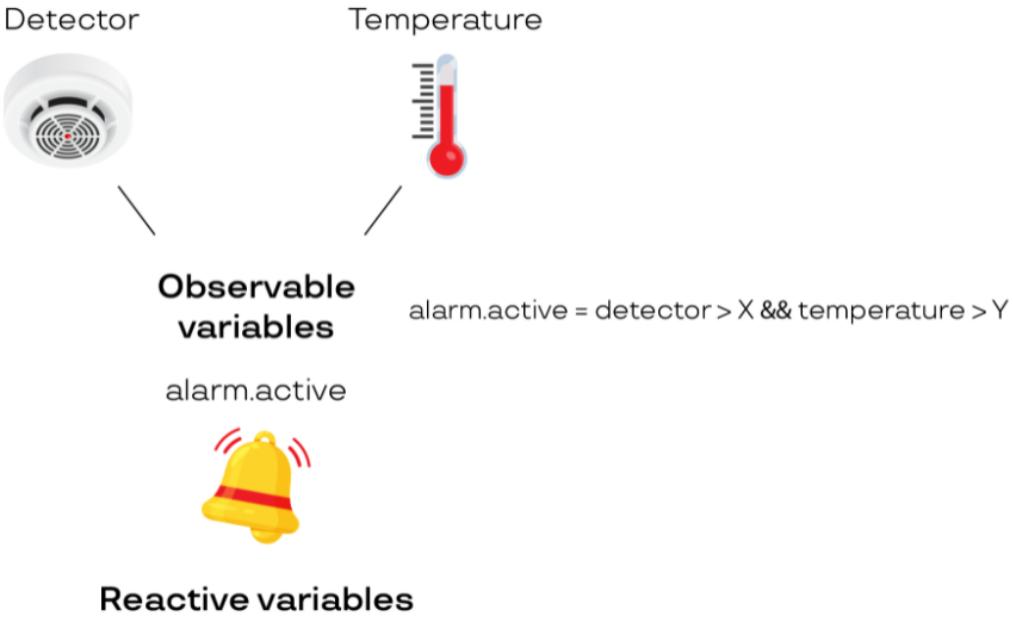

Рисунок 1.4 – Приклад з датчиком

У нас є датчик диму і градусник. Коли диму стає багато або температура зростає, на відповідних датчиках збільшується значення. Коли значення і температура на датчику диму виявляються вищими граничних, включається дзвіночок і оповіщає про тривогу.

Якби у нас був традиційний, а не реактивний підхід, ми б писали код, який кожні п'ять хвилин опитує детектор диму і датчик температури, і включає або вимикає дзвіночок. Однак в реактивному підході за нас це робить реактивний фреймворк, а ми тільки прописуємо умови: дзвіночок активний, коли детектор більше X, а температура більше Y. Це відбувається кожного разу, коли приходить нова подія.

Від детектора диму йде потік даних: наприклад, значення 10, потім 12, і т.д. Температура теж змінюється, це інший потік даних - 20, 25, 15. Кожен раз, коли з'являється нове значення, результат перераховується, що призводить до включення або виключення системи оповіщення. Нам досить сформулювати умову, за якої дзвіночок повинен включитися.

Якщо повернутися до паттерну Observer, у нас детектор диму і термометр - це публікатори повідомлень, тобто джерела даних (Publisher), а дзвіночок на них підписаний, тобто він Subscriber, або спостерігач (Observer).

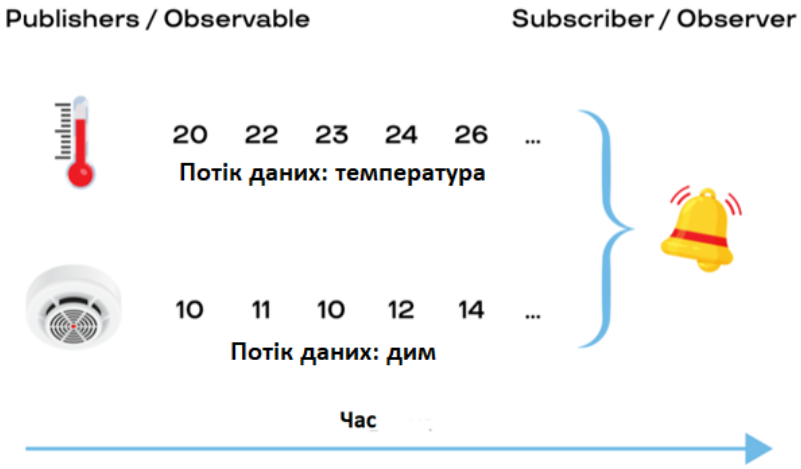

Рисунок 1.5 – Приклад з реактивним датчиком

Трохи розібравшись з ідеєю реактивності, давайте заглибимось в реактивний підхід. Ми поговоримо про операторів реактивного програмування. Оператори дозволяють будь-яким чином трансформувати потоки даних, змінюючи дані і створюючи нові потоки. Для прикладу розглянемо оператор distinctUntilChanged. Він прибирає однакові значення, що йдуть один за одним. Дійсно, якщо значення на детекторі диму не змінилося - навіщо нам на нього реагувати і щось там перераховувати:

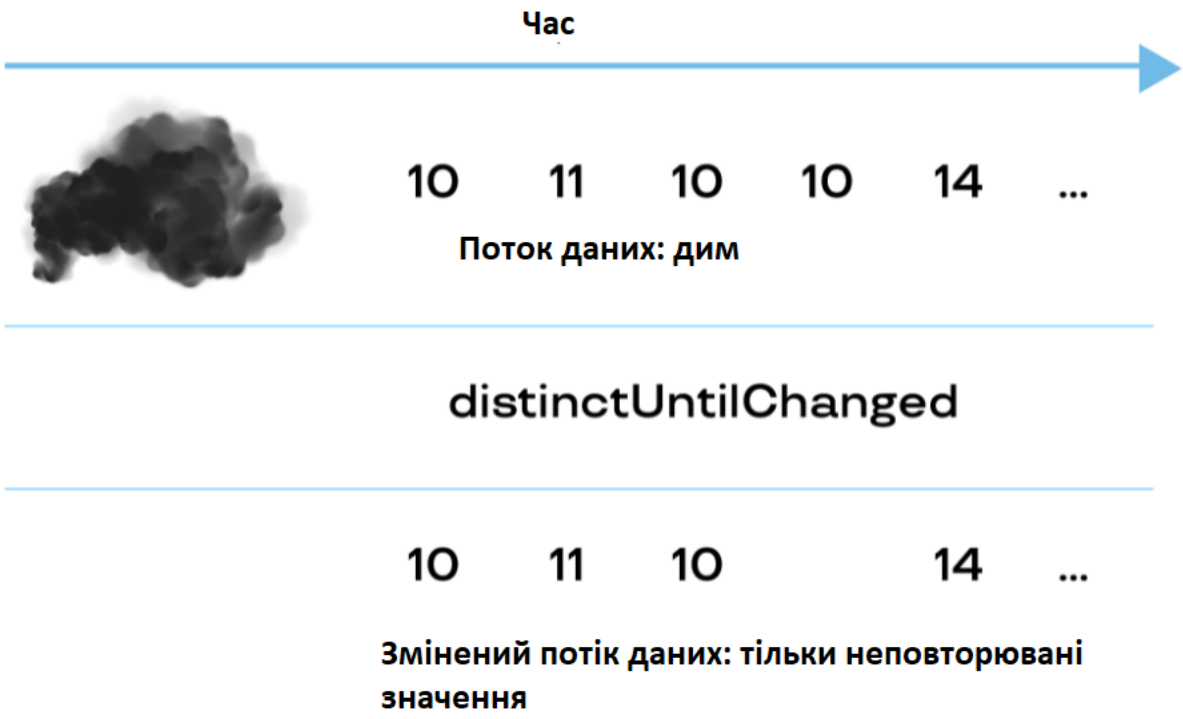

Рисунок 1.6 – Другий приклад з реактивним датчиком

РОЗДІЛ 2. СТВОРЕННЯ ПРОЕКТУ НА РЕАКТИВНОМУ ПІДХОДІ

2.1 Створення проекту

Я буду використовувати Java Spring Boot Framework. Для простого створення проекту достатньо зайти на сайт https://start.spring.io/ та вибрати всі необхідні бібліотеки:

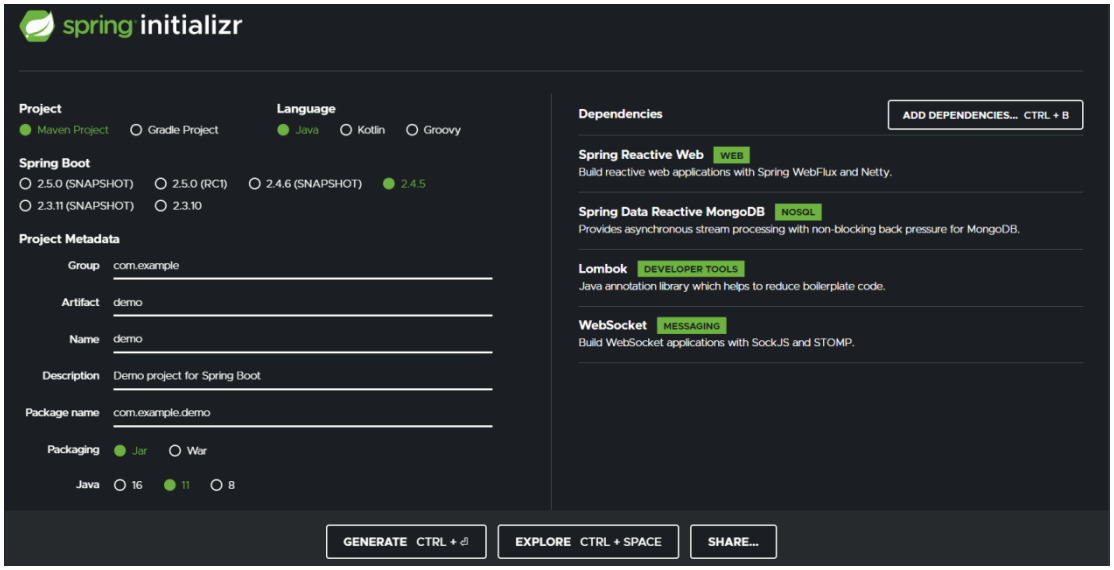

Рисунок 2.1 – Зразок Spring initializr

Версія Java – 11

В якості збірника проекту буду Maven.

2.2 Налаштування мережі Netty Boot Reactor

Перш ніж ми почнемо, давайте розглянемо, що таке Reactor Netty і як він пов’язаний із Spring Boot.

Reactor Netty - це асинхронний механізм мережевих додатків, керований подіями. Він надає клієнтам та серверам TCP, HTTP та UDP, що не блокують та підтримують протитиск. Як випливає з назви, він базується на структурі Netty.

По-перше, ми додамо необхідну залежність Maven.

Щоб використовувати сервер Reactor Netty, ми додамо spring-boot-starter-webflux як залежність у наш файл pom:

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

Рисунок 2.2 – Конфігурація Maven

Це також залучить spring-boot-starter-reactor-netty як транзитивну залежність від нашого проекту.

Можна налаштувати сервер Netty через properties files(файли власностей). Spring Boot пропонує деякі загальні конфігурації сервера у файлах своїх додатків:

Визначити порт сервера в application.properties:

```
server.port=${PORT:8080}
```

Рисунок 2.3 – Конфігурація порта

Будемо використовувати клас SslServerCustomizer, який є ще однією реалізацією NettyServerCustomi:

```
@Component
public class NettyWebServerFactorySslCustomizer
        implements WebServerFactoryCustomizer<NettyReactiveWebServerFactory>{

    @Override
    public void customize(NettyReactiveWebServerFactory serverFactory) {
        Ssl ssl = new Ssl();
        ssl.setEnabled(true);
        ssl.setKeyStore("classpath:sample.jks");
        ssl.setKeyAlias("alias");
        ssl.setKeyPassword("password");
        ssl.setKeyStorePassword("secret");
        Http2 http2 = new Http2();
        http2.setEnabled(false);
        serverFactory.addServerCustomizers(new SslServerCustomizer(ssl,
http2, null));
        serverFactory.setPort(8443);
    }
}
```

Рисунок 2.4 – клас NettyServerCustomi

Тут ми визначили наші властивості, пов’язані з магазином ключів, вимкнули HTTP / 2 і встановили порт на 8080

Тепер ми розглянемо, як ми можемо налаштувати ведення журналу доступу за допомогою Logback.

Spring Boot дозволяє налаштувати реєстрацію доступу у файлі властивостей програми для Tomcat, Jetty та Undertow. Однак у Netty ще немає цієї підтримки.

Щоб увімкнути журналювання доступу до Netty, нам слід встановити -Dreactor.netty.http.server.accessLogEnabled = true під час запуску додатку:

mvn spring-boot:run -Dreactor.netty.http.server.accessLogEnabled=true

2.3 Spring Security

Діаграма показує, як ми впроваджуємо процедуру реєстрації користувачів, реєстрації користувачів та авторизації:

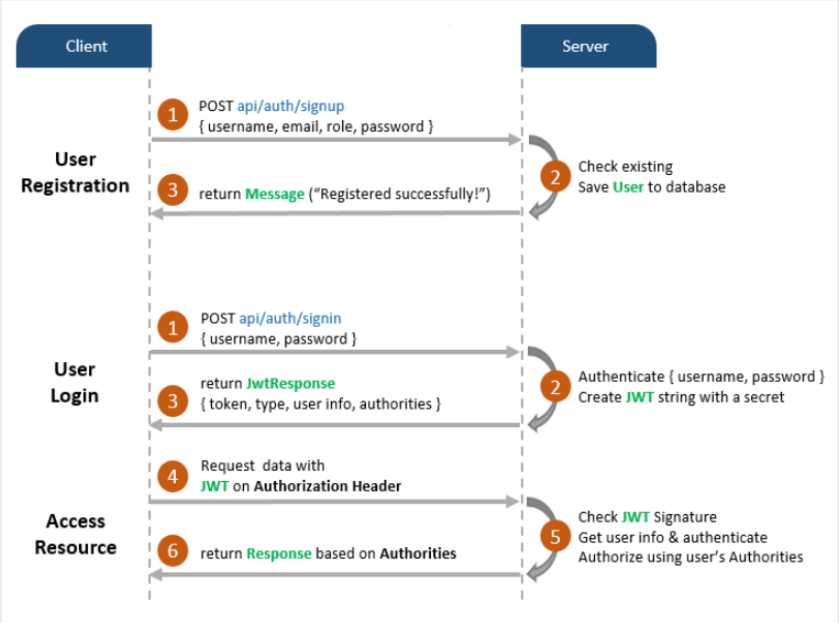

Рисунок 2.5 – Діаграма класів

JWT повинен бути доданий до загальної авторизації HTTP, якщо клієнт повертається до захищених ресурсів.

Отримати огляд Spring Boot Server можна за діаграмою нижче:

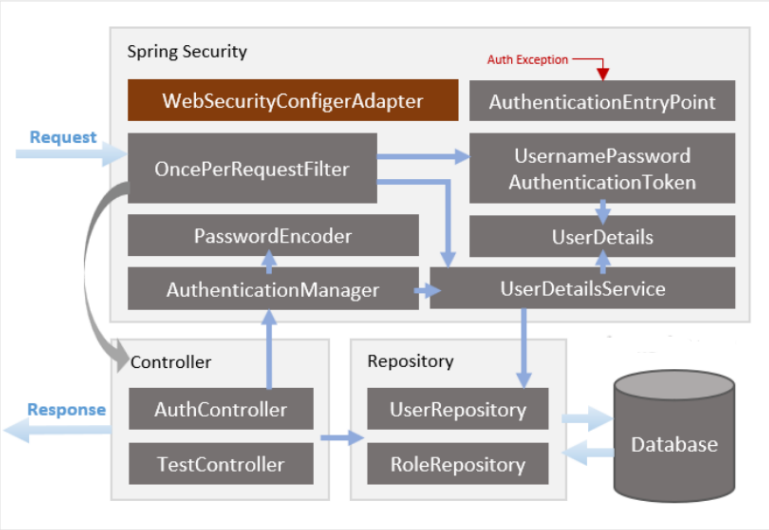

Рисунок 2.6 – Друга діаграма класів

Дозвольте коротко пояснити.

Spring Security

- WebSecurityConfigurerAdapter - суть реалізації безпеки. Він надає конфігурації HttpSecurity для налаштування cors, csrf, управління сесіями, правил для захищених ресурсів. Також можемо розширити та налаштувати конфігурацію за замовчуванням, яка містить елементи нижче.
- Інтерфейс UserDetailsService має метод завантаження користувача за іменем користувача та повертає об’єкт UserDetails, який Spring Security може використовувати для автентифікації та перевірки.
- UserDetails містить необхідну інформацію (таку як: ім'я користувача, пароль, повноваження) для побудови об'єкта аутентифікації.
- UsernamePasswordAuthenticationToken отримує {ім'я користувача, пароль} із Запиту на вхід, AuthenticationManager використовуватиме його для автентифікації облікового запису для входу.
- AuthenticationManager має DaoAuthenticationProvider (за допомогою UserDetailsService & PasswordEncoder) для перевірки об’єкта UsernamePasswordAuthenticationToken. У разі успіху AuthenticationManager повертає повністю заповнений об’єкт автентифікації (включаючи надані повноваження).
- OncePerRequestFilter робить одне виконання для кожного запиту до нашого API. Він надає метод doFilterInternal (), за допомогою якого ми реалізуємо синтаксичний аналіз та перевірку JWT, завантаження даних користувача (за допомогою UserDetailsService), перевірку авторизації (за допомогою UsernamePasswordAuthenticationToken).
- AuthenticationEntryPoint виявить помилку автентифікації.

Поки що трібно створити одну колекцію в базі даних. В якості бази данних буду використовувати – MongoDB. Чому саме вона?

MongoDB - це база даних документів із масштабною зручністю та гнучкістю, яка мені потрібна, із запрошеннями та індексированням. Зберігає дані у гнучких, схожих на JSON документах, щоб можна було заповнити

поля з документа до документа, а структура даних може змінюватися протягом часу. Та головне: MongoDB можна використовувати безкоштовно. Версії, випущені до 16 жовтня 2018 р., Публікуються на умовах AGPL. Усі версії, випущені після 16 жовтня 2018 р., включаючи виправлення для попередніх версій, публікуються в рамках загальнодоступної серверної ліцензії (SSPL) v1.

Для підключення бази даних MongoDB, достатньо добавити у файл pom:

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

Рисунок 2.7 – Залежність Maven

Та в application.properties посилання на неї:

```
spring.data.mongodb.uri=mongodb+srv://zipliSo
```

Рисунок 2.8 – Посилання на базу даних

Анотація @Document – зв`язує об'єкт з базою данних:

```
@Getter
@Setter
@ToString
@Document
@NoArgsConstructor
public class User {

    @Id
    private String id;

    @Indexed(unique = true)
    private String email;
    private String password;

    @Indexed(unique = true)
    private String userName;
    private String nickName;
    private boolean isConfirm;
    private List<String> chatsId;
    private String avatar;

    public User(String email, String password, String userName, String
nickName) {
        this.email = email;
        this.password = password;
        this.userName = userName;
        this.nickName = nickName;
        this.isConfirm = false;
        chatsId = new ArrayList<>();
    }
```

Рисунок 2.9 – Клас User

Lombok дає можливість не писати базові штуки:

- Де @Getter @Setter - гетери\сетери.
- @ToString – не перевизначать toString

- @NoArgsConstructor – створює конструктор без параметрів

В User будемо зберігати:

- Id – унікальний індефікатор користувача
- Email – поштова скринька(унікальна)
- Password - пароль
- UserName – ім'я користувача(унікальне)
- NickName – псевдонім
- IsConfirm – підтвердження акаунту
- ChatsId – список чатів в яких буде перебувати користувач
- Avatar – аватар, яких буду зберігатися у текстовому вигляді.

Тепер для моделі, представленої вище, потрібно створити репозиторій для зберігання даних і доступу до них:

```
@Component
public class UserRepository {

    private final MongoTemplate mongoTemplate;

    public UserRepository(MongoTemplate mongoTemplate) {
        this.mongoTemplate = mongoTemplate;
    }

    public User save(User user) {
        return mongoTemplate.save(user);
    }

    public void saveAll(List<User> users) {
        for (User user : users) {
            mongoTemplate.save(user);
        }
    }

    public User getUserByEmail(String email) {
        Query query = new Query()
                .addCriteria(where("email").is(email));
        return mongoTemplate.findOne(query, User.class);
    }
```

```
    public User getUserByUserName(String userName) {
        Query query = new Query()
                .addCriteria(where("userName").is(userName));
        return mongoTemplate.findOne(query, User.class);
    }

    public User getUserById(String id) {
        Query query = new Query()
                .addCriteria(where("id").is(id));
        return mongoTemplate.findOne(query, User.class);
    }

    public List<User> findUsersByIdIn(Collection<String> id) {
        Query query = new Query()
                .addCriteria(where("id").in(id));
        return mongoTemplate.find(query, User.class);
    }

    public List<User> findAllByIsConfirm(boolean confirm) {
        Query query = new Query()
                .addCriteria(where("isConfirm").is(confirm));
        return mongoTemplate.find(query, User.class);
    }

    public User findUserByEmail(String email) {
        Query query = new Query()
                .addCriteria(where("email").is(email));
        return mongoTemplate.findOne(query, User.class);
    }

    public void deleteById(String id) {
        Query query = new Query()
                .addCriteria(Criteria.where("id").is(id));
        mongoTemplate.remove(query, User.class);
    }

    public User updateOrDeleteAvatar(String userId, String avatar) {
        Query query = new Query()
                .addCriteria(where("id").is(userId));
        Update update = new Update()
                .set("avatar", avatar);
        return mongoTemplate.findAndModify(query, update, new
FindAndModifyOptions().returnNew(true), User.class);
```

```
    }

    public User updateNickName(String userId, String nickName) {
        Query query = new Query()
                .addCriteria(where("id").is(userId));
        Update update = new Update()
                .set("nickName", nickName);
        return mongoTemplate.findAndModify(query, update, new
FindAndModifyOptions().returnNew(true), User.class);
    }

    public User updateEmail(String userId, String email) {
        Query query = new Query()
                .addCriteria(where("id").is(userId));
        Update update = new Update()
                .set("email", email)
                .set("isConfirm", false);
        return mongoTemplate.findAndModify(query, update, new
FindAndModifyOptions().returnNew(true), User.class);
    }

    public User updatePassword(String userId, String password) {
        Query query = new Query()
                .addCriteria(where("id").is(userId));
        Update update = new Update()
                .set("password", password);
        return mongoTemplate.findAndModify(query, update, new
FindAndModifyOptions().returnNew(true), User.class);
    }

    public List<User> findUsersBySearchParam(String param) {
        Criteria criteria = new Criteria();
        Query query = new Query(criteria);

        criteria.orOperator(
                Criteria.where("userName").regex("^" + param),
                Criteria.where("nickName").regex("^" + param),
                Criteria.where("email").regex("^" + param));

        return mongoTemplate.find(query, User.class);
    }


    public void confirmAccountInUsersModel(String userName) {
```

```
        Query query = new Query();
        query.addCriteria(where("userName").is(userName));
        Update update = new Update();
        update.set("isConfirm", true);
        mongoTemplate.updateFirst(query, update, User.class);
    }

    public void updatePasswordInUsersModel(String userName, String
codedPassword) {
        Query query = new Query();
        query.addCriteria(where("userName").is(userName));
        Update update = new Update();
        update.set("password", codedPassword);
        mongoTemplate.updateFirst(query, update, User.class);
    }
}
```

Рисунок 2.10 – Клас UserRepository

Хоч на MongoTemplate потрібно писати більше коду, але він дає більше гнучкості в запитах. Вище написано всі запити які знадобляться при розробці.

Методи класу:

- save(User user) – зберігає об'єкт у базу даних
- saveAll(List<User> users) – зберігає список об'єктів у базу даних
- getUserByEmail(String email) – віддає об'єк користувача по почті
- getUserByUserName(String userName) – віддає об'єк користувача по імені
- getUserById(String id) – віддає об'єк користувача по Id
- findUsersByIdIn(Collection<String> id) – віддає список об'єків користувачів по їх id
- findAllByIsConfirm(boolean confirm) – віддає список об'єків користувачів по їх підтвердженням
- deleteById(String id) – видаляє користувача по id
- updateOrDeleteAvatar(String userId, String avatar) – змінює або добавляє аватар користувача

- updateNickName(String userId, String nickName) - змінює або добавляє псевдонім користувача
- updateEmail(String userId, String email) - змінює почту користувача
- updatePassword(String userId, String password) - змінює пароль користувача
- findUsersBySearchParam(String param) – шукаю список користувачів по якомусь параметру
- confirmAccountInUsersModel(String userName) – підтверджує користувача

У безпеки створив WebSecurityConfig класс, який буде вирішувати на які посиланням можна заходити без авторизації:

```
@Configuration
@EnableWebFluxSecurity
public class WebSecurityConfig {
   private final AuthTokenManager authTokenManager;
   private final SecurityContextRepository securityContextRepository;
   @Value("${cors.urls}")
   private List<String> corsUrls;
   @Value("${cors.path}")
   private String corsPath;

   public WebSecurityConfig(AuthTokenManager authTokenManager,
SecurityContextRepository securityContextRepository) {
      this.authTokenManager = authTokenManager;
      this.securityContextRepository = securityContextRepository;
   }

   @Bean
   public SecurityWebFilterChain securityWebFilterChain(ServerHttpSecurity
http) {
      return http
            .exceptionHandling()
            .authenticationEntryPoint(
                  (swe, e) -> Mono.fromRunnable(
                        () ->
swe.getResponse().setStatusCode(HttpStatus.UNAUTHORIZED)
                  )
            )
            .and()
```

```
            .csrf().disable()
            .authenticationManager(authTokenManager)
            .securityContextRepository(securityContextRepository)
            .authorizeExchange()
            .pathMatchers("/zipli/auth/signup",
                  "/zipli/auth/signin",
                  "/zipli/auth/forgot_password",
                  "/zipli/room/**",
                  "/v2/api-docs",
                  "/configuration/ui",
                  "/swagger-resources/**",
                  "/configuration/security",
                  "/swagger-ui/**",
                  "/webjars/**").permitAll()
            .anyExchange().authenticated()
//            .and().oauth2Client()
            .and().build();
   }

   @Bean
   CorsConfigurationSource corsConfiguration() {
      CorsConfiguration corsConfig = new CorsConfiguration();
      corsConfig.applyPermitDefaultValues();
      corsConfig.addAllowedMethod(String.valueOf(HttpMethod.PUT));
      corsConfig.addAllowedMethod(String.valueOf(HttpMethod.DELETE));
      corsConfig.setAllowedOrigins(corsUrls);

      UrlBasedCorsConfigurationSource source = new
UrlBasedCorsConfigurationSource();
      source.registerCorsConfiguration(corsPath, corsConfig);
      return source;
   }

   @Bean
   public PasswordEncoder passwordEncoder() {
      return new BCryptPasswordEncoder();
   }

}
```

Рисунок 2.11 – Клас WebSecurityConfig

@EnableWebFluxSecurity – додає підтримку Spring Security WebFlux, працює тільки в реактивній Java. Дозволяє Spring знаходити і автоматично застосовувати клас до глобальної веб-безпеки.

Ми перевизначає securityWebFilterChain (ServerHttpSecurity http) метод з інтерфейса WebSecurityConfigurerAdapter. Він повідомляє Spring Security, як ми налаштовуємо CORS і CSRF, коли ми хочемо вимагати аутентифікації всіх користувачів чи ні, який filter (AuthTokenFilter) і коли ми хочемо, щоб він працював (фільтрує перед іншими фільтрами), який обробник виключень обраний. Нам також потрібен PasswordEncoder для шифрування пароля.

Spring Security завантажує дані користувача для виконання аутентифікації та авторизації. Ітак, у нього є інтерфейс UserDetailsService, який нам потрібно реалізувати, Я хочу отримати більше даних (ідентифікатор, адресу електронної пошти…), тому створю реалізацію цього інтерфейсу UserDetailsImpl:

```
public class UserDetailsImpl implements UserDetails {

   private static final long serialVersionUID = 1L;
   private final User user;

   private Collection<? extends GrantedAuthority> authorities;

   public UserDetailsImpl(User user) {
      this.user = user;
   }


   @Override
   public Collection<? extends GrantedAuthority> getAuthorities() {
      return authorities;
   }

   @Override
   public String getPassword() {
      return user.getPassword();
   }
```

```
    @Override
    public String getUsername() {
        return user.getUserName();
    }

    @Override
    public int hashCode() {
        return Objects.hash(
                user.getUserName(),
                user.getEmail(),
                user.getPassword(),
                user.getNickName(),
                user.isConfirm());
    }

    @Override
    public boolean equals(Object o) {
        if (this == o)
            return true;
        if (o == null || getClass() != o.getClass())
            return false;
        UserDetailsImpl user = (UserDetailsImpl) o;
        return Objects.equals(this.user.getUserName(), user.getUsername())
                && Objects.equals(this.user.getPassword(), user.getPassword());
    }
}
```

Рисунок 2.12 – Клас UserDetailsImpl

Ми можемо використовувати AuthenticationManagerResolver всюди, де нам потрібно динамічно вибирати AuthenticationManager, але в цьому керівництві ми зацікавлені в його використанні у вбудованих потоках аутентифікації.

Спочатку давайте налаштуємо AuthenticationManagerResolver, а потім будемо використовувати його для аутентифікації Basic і OAuth2.

Почнемо зі створення класу для настройки безпеки. Ми повинні розширити WebSecurityConfigurerAdapter.

```
@Configuration
public class CustomWebSecurityConfigurer extends
WebSecurityConfigurerAdapter {
    // ...
}
```

Рисунок 2.13 – Клас CustomWebSecurityConfigurer

Потім давайте додамо метод, який повертає AuthenticationManager для клієнтів:

```
AuthenticationManager customersAuthenticationManager() {
   return authentication -> {
      if (isCustomer(authentication)) {
         return new UsernamePasswordAuthenticationToken(/*credentials*/);
      }
      throw new UsernameNotFoundException(/*principal name*/);
   };
}
```

Рисунок 2.14 – Метод customersAuthenticationManager

Нарешті, давайте додамо AuthenticationManagerResolver, який дозволяється відповідно до URL-адресою запиту:

```
AuthenticationManagerResolver<HttpServletRequest> resolver() {
   return request -> {
      if (request.getPathInfo().startsWith("/employee")) {
         return employeesAuthenticationManager();
      }
      return customersAuthenticationManager();
   };
}
```

Рисунок 2.15 – Метод resolver

Для реактивного веб-додатки все ще можемо отримати вигоду з концепції дозволу AuthenticationManager відповідно до контексту.

```
@Slf4j
@Component
public class AuthTokenManager implements ReactiveAuthenticationManager {
   private final JwtUtils jwtUtils;
   private final UserDetailsService userDetailsService;

   public AuthTokenManager(JwtUtils jwtUtils,
@Qualifier("userDetailsServiceImpl") UserDetailsService userDetailsService) {
      this.jwtUtils = jwtUtils;
      this.userDetailsService = userDetailsService;
   }
```

```
        @Override
    public Mono<Authentication> authenticate(Authentication authentication) {
        String authToken = authentication.getCredentials().toString();

        String username;

        if (authToken != null && jwtUtils.validateJwtToken(authToken)) {
            username = jwtUtils.getUserNameFromJwtToken(authToken);

            UserDetails userDetails =
userDetailsService.loadUserByUsername(username);
            UsernamePasswordAuthenticationToken authenticationToken = new
UsernamePasswordAuthenticationToken(
                    userDetails, null, userDetails.getAuthorities());

            log.info("User authentication");
            return Mono.just(authenticationToken);
        } else {
            log.info("Cannot set user authentication: no valid Jwt token");
            throw new AuthenticationException();
        }
    }
}
```

Рисунок 2.16 – Клас AuthTokenManager

Вона повертає Mono з ReactiveAuthenticationManager. ReactiveAuthenticationManager є реактивним еквівалентом AuthenticationManager, тому його метод аутентифікації повертає Mono.

РОЗДІЛ 3. АУТЕНТИФІКАЦІЯ

3.1. Аутентифікація JWT

Потрібно створити класс JWTUtils. У цього класу є 3 функції:

- згенерувати JWTіз імені користувача, дати, терміну дії, секрету
- отримати ім'я(або EMAIL) користувача від JWT
- підтвердити JWT

```
public class JwtUtils {
    public static final String EMAIL = "email";
    @Value("${app.jwtSecret}")
    private String jwtSecret;
    @Value("${app.jwtExpirationMs}")
    private int jwtExpirationMs;
    public String generateJwtToken(UserDetails userPrincipal, String email) {
        return Jwts.builder()
                .setSubject((userPrincipal.getUsername()))
                .claim(EMAIL, email)
                .setIssuedAt(new Date())
                .setExpiration(new Date((new Date()).getTime() + jwtExpirationMs))
                .signWith(SignatureAlgorithm.HS512, jwtSecret)
                .compact();
    }
    public String getUserNameFromJwtToken(String token) {
        return Jwts.parser()
                .setSigningKey(jwtSecret)
                .parseClaimsJws(token)
                .getBody()
                .getSubject();
    }
    public boolean validateJwtToken(String authToken) {
        try {
            Jwts.parser().setSigningKey(jwtSecret).parseClaimsJws(authToken);
            return true;
        } catch (SignatureException e) {
            log.error("Invalid JWT signature: {}", e.getMessage());
        }
        return false;
    }
}
```

Рисунок 3.1 – Клас JwtUtils

String jwtSecret та int jwtExpirationMs, прописуємо в client.properties:

```
app.jwtSecret=ialythiireapernanazemle
app.jwtExpirationMs=86400000
```

Рисунок 3.2 – Приклад файлу типу properties

86400000 – 24 години в милисекундах

3.2. Створення Rest контролерів для авторизації

REST швидко став стандартом для створення веб-сервісів в мережі, тому що їх легко створювати і легко використовувати. Включає в себе принципи Інтернету, включаючи його архітектуру, переваги і все інше. Це не дивно, враховуючи, що його автор, Рой Філдінг, брав участь, ймовірно, в десятці специфікацій, які регулюють роботу мережі.

Які переваги? Інтернет і його основний протокол HTTP надають набір функцій:

- Відповідні дії (GET, POST, PUT, DELETE, ...)
- Кешування
- Перенаправлення і пересилання
- Безпека (шифрування і аутентифікація)

Все це критичні фактори для створення відмовостійких сервісів. Але це ще не все. Інтернет побудований з безлічі крихітних специфікацій, тому він може легко розвиватися, не грузнучи в «війнах стандартів».

Щоб обернути репозиторій веб-шаром, потрібно звернутися до Spring MVC. Завдяки Spring Boot в інфраструктурі практично немає коду. Замість цього ми можемо зосередитися на діях.

@RestController вказує, що дані, які повертаються кожним методом, будуть записані прямо в тіло відповіді замість візуалізації шаблону.

Є маршрути для кожної операції (@GetMapping, @PostMapping, @ PutMappingi @DeleteMapping, що відповідає HTTP GET, POST, PUTi DELETE дзвінки).

Тепер почнемо створювати запити.

3.3. Запит регістрації

```
@PostMapping("/signup")
public ResponseEntity<?> addUser(@Valid @RequestBody SignupRequest
signupRequest) {
   User user = new User(signupRequest.getEmail(),
         signupRequest.getPassword(),
         signupRequest.getUserName(),
         signupRequest.getNickName());
   try {
      authService.registration(user);
   } catch (AuthException e) {
      log.error("Failed registration by email {}, userName {}, nickName {}, reason
{}", signupRequest.getEmail(),
            signupRequest.getUserName(), signupRequest.getNickName(),
e.getErrorStatusCode().getMessage());
      return ResponseEntity
            .badRequest()
            .body(new ErrorResponse(e.getErrorStatusCode()));
   }
   return ResponseEntity.ok("User registered successfully!");
}
```

Рисунок 3.3 – Метод addUser

Приймає в себе SignupRequest де за допомогою javax.validation.constraints.* перевіряється валідація:

```
@Getter
@Setter
public class SignupRequest {
   @NotBlank(message = "Email can't be empty")
   @NotNull(message = "Email can't be null")
   @Email
   @Size(max = 50)
   @Pattern(regexp = "^\\w+([\\.-]?\\w+)*@\\w+([\\.-]?\\w+)*\\.\\w{2,}$")
   private String email;

   @NotBlank(message = "Password can't be empty")
   @NotNull(message = "Password can't be null")
   @Size(min = 8, max = 16)
   @Pattern(regexp = "^(?=.*[0-9])(?=.*[a-z])(?=.*[A-Z])(?=\\S+$).{8,16}$")
   private String password;

   @NotBlank(message = "User name can't be empty")
   @NotNull
```

```
    @Pattern(regexp = "^[a-zA-Z_-]{2,16}$")
    private String userName;

    @NotBlank(message = "Nickname can't be empty")
    @Pattern(regexp = "^.{1,16}$")
    private String nickName;
```

Рисунок 3.4 – Клас SignupRequest

Анотації:

- @NotBlank - встановлює що поле не повинно бути порожнім
- @NotNull - встановлює що поле не повинно бути Нулла
- @Email - встановлює що поле повинно поштою
- @Size - встановлює розмір допустимого поля
- @Pattern - встановлює патерн с помощью регулярного виразу

Як ми показано по коду вище, вся основна логіка виконується в AuthService. Метод registration:

```
@Override
public void registration(User user) {
    User existingUser = userRepository.getUserByEmail(user.getEmail());
    if (existingUser != null) {
        throw new AuthException(ErrorStatusCode.EMAIL_ALREADY_EXISTS);
    } else {
        user.setPassword(passwordEncoder.encode(user.getPassword()));
        userRepository.save(user);
        UserDetails userDetails = new UserDetailsImpl(user);
        String token = jwtUtils.generateJwtToken(userDetails, user.getEmail());
        emailConfirmationService.sendEmail(user.getEmail(), token);
    }
}
```

Рисунок 3.5 – Метод registration

Приймає в себе об об’єкт User. Перевіряє щоб такого юзера не було у базі даних, якщо такий є одтаємо помилку у відповідь. Якщо такого нема – додаємо до бази даних та шифруємо пароль. Також відправляємо на почту

повідомлення для підтвердження акаунту, за це відповідає метод sendEmail в EmailConfirmationService:

```
@Async
public void sendEmail(String email, String token) {
    MimeMessage message = javaMailSender.createMimeMessage();
    try {
        MimeMessageHelper helper = new MimeMessageHelper(message, true, "utf-
8");
        helper.setTo(email);
        helper.setSubject("Complete Registration!");
        helper.setFrom("zipli.socknet@gmail.com");
        String htmlMsg = "<h3>Confirm your mail</h3>"
                + "<p style=\"font-size:18px;\">To confirm your user, please click "
                + "<strong>"
                + "<a href=\"" + deploy + "/confirm-mail?token=" + token + "\"
target=\"_blank\">here</a>"
                + "</strong>"
                + "</p>"
                + "<br/>"
                + "<br/>"
                + "<br/>";

        message.setContent(htmlMsg, "text/html");

            new Thread(() -> {
           try {
              javaMailSender.send(message);
           } catch (Exception e) {
              log.error("Error send message to email {} message {} class {}", email,
e.getMessage(), e.getClass().getSimpleName());
           }

        }).start();
    } catch (MessagingException e) {
        log.error("Error to create message to email {} message {} class {}", email,
e.getMessage(), e.getClass().getSimpleName());
    }
}
```

Рисунок 3.6 – Метод sendEmail

Тут створюється простий лист, який має посилання на підтвердження.

3.4. Запит підтвердження облікового запису

Як я писав раніше, для того щоб створити обліковий запит потрібно підтвердити його через пошту. За це відповідає Rest контролер /confirm-mail:

```
@PostMapping("/confirm-mail")
public ResponseEntity<?> emailConfirm(@Valid @RequestBody
ConfirmMailRequest confirmMailRequest) {
   try {
      emailConfirmationService.confirmAccount(confirmMailRequest.getToken());
   } catch (NotConfirmAccountException e) {
      log.error("Failed confirm email tokenIsNull {}, reason {}",
Objects.isNull(confirmMailRequest.getToken()),
e.getErrorStatusCode().getMessage());
      return ResponseEntity
            .badRequest()
            .body(new ErrorResponse(e.getErrorStatusCode()));
   }
   return ResponseEntity.ok("Account verified");
}
```

Рисунок 3.7 – Метод emailConfirm

Приймає в себе ConfirmMailRequest де зберігається токен:

```
@Getter
@Setter
@NoArgsConstructor
@AllArgsConstructor
public class ConfirmMailRequest {
   private String token;
}
```

Рисунок 3.8 – Клас ConfirmMailRequest

Логіка виконання знаходиться в EmailConfirmationService, а саме метод confirmAccount:

```
public String confirmAccount(String token) {
   if (token != null) {
      String userName = jwtUtils.getUserNameFromJwtToken(token);
      userRepository.confirmAccountInUsersModel(userName);
      return "Account verified";
   } else {
      throw new
```

```
NotConfirmAccountException(ErrorStatusCode.TOKEN_INVALID_OR_BROKEN
);
    }
}
```

Рисунок 3.9 – Метод confirmAccount

В ньому ходимо в базу та у полі confirm ставимо True, якщо токен не існуючий повертаємо помилку.

3.5. Запит входу до облікового запису

За це відповідає запит /signin:

```
@PostMapping("/signin")
public ResponseEntity<?> authenticateUser(@Valid @RequestBody
LoginRequest loginRequest) {
    LoginResponse loginResponse;
    try {
        loginResponse = authService.login(loginRequest.getLogin(),
loginRequest.getPassword());
    } catch (AuthException e) {
        log.error("Failed authenticate user by login {}, passwordIsNull {}, reason
{}",
                loginRequest.getLogin(), Objects.isNull(loginRequest.getPassword()),
e.getErrorStatusCode().getMessage());
        return ResponseEntity
                .badRequest()
                .body(new ErrorResponse(e.getErrorStatusCode()));
    }
    return ResponseEntity.ok(loginResponse);
}
```

Рисунок 3.10 – Метод authenticateUser

Приймає в себе LoginRequest яких також валідується за допомогою анотацій:

```
@Getter
@Setter
@Validated
@NoArgsConstructor
@AllArgsConstructor
public class LoginRequest {
    @NotBlank(message = "Login can't be empty")
    @NotNull
    @Pattern(regexp = "^\\w+([\\.-]?\\w+)*@\\w+([\\.-]?\\w+)*\\.\\w{2,}$|^[a-zA-
Z_-]{2,16}$")
    private String login;
```

```
    @NotBlank(message = "Password can't be empty")
    @NotNull
    @Size(min = 8, max = 16)
    @Pattern(regexp = "^(?=.*[0-9])(?=.*[a-z])(?=.*[A-Z])(?=\\S+$).{8,16}$")
    private String password;
}
```

Рисунок 3.11 – Клас LoginRequest

Також потрібно обговорити як працює авторизація. Якщо такий обліковий запис знайдено в базі та він підтверджений, за допомогою класу JWTUtils генеруємо токен та передаємо його на фронт. За це відповідає метод login с класу AuthService:

```
@Override
public LoginResponse login(String emailOrUsername, String password) {
    User result;
    if (emailOrUsername.contains("@")) {
        result = userRepository.getUserByEmail(emailOrUsername);
    } else {
        result = userRepository.getUserByUserName(emailOrUsername);
    }
    if (result == null) {
        throw new AuthException(ErrorStatusCode.USER_DOES_NOT_EXIST);
    } else {
        if (!passwordEncoder.matches(password, result.getPassword())) {
            throw new AuthException(ErrorStatusCode.PASSWORD_INCORRECT);
        } else if (!result.isConfirm()) {
            throw new
AuthException(ErrorStatusCode.USER_DOES_NOT_PASS_EMAIL_CONFIRM);
        } else {
            String token = jwtUtils.generateJwtToken(new UserDetailsImpl(result),
result.getEmail());
            return new LoginResponse(result.getId(), token, token);
        }
    }
}
```

Рисунок 3.12 – Метод login

При всіх виняткових ситуаціях повертаємо помилку, про які ми поговоримо чуть пізніше.

3.6. Запит на відновлення паролю

Якщо користувач забув пароль він може його скинути. За це відповідає запит /forgot_password:

```
@PostMapping("/forgot_password")
public ResponseEntity<?> processForgotPassword(@Valid @RequestBody
ForgotPasswordRequest forgotPasswordRequest) {
   try {

resetPasswordService.generateResetPasswordToken(forgotPasswordRequest.getE
mail());
   } catch (UserNotFoundException e) {
      log.error("Failed restore password by email {}, reason {}",
forgotPasswordRequest.getEmail(), e.getErrorStatusCode().getMessage());
      return ResponseEntity
            .badRequest()
            .body(new ErrorResponse(e.getErrorStatusCode()));
   }

resetPasswordService.sendEmailForChangingPassword(forgotPasswordRequest.ge
tEmail());
   return ResponseEntity.ok("Password can be changed");
}
```

Рисунок 3.13 – Метод processForgotPassword

Для створення посилання посилення для відновлення паролю потрібно створити токен, за це відповідає метод generateResetPasswordToken класу ResetPasswordService:

```
public String generateResetPasswordToken(String email) {
   if (email == null) {
      throw new
UserNotFoundException(ErrorStatusCode.EMAIL_DOES_NOT_CORRECT);
   } else {
      User user = userRepository.getUserByEmail(email);
      if (user != null) {
         String userName = user.getUserName();
         UserDetails userDetails =
userDetailsService.loadUserByUsername(userName);
         String token = jwtUtils.generateJwtToken(userDetails, email);
         return token;
      }
   }
```

Рисунок 3.14 – Метод generateResetPasswordToken

За відправку відповідає метод sendEmailForChangingPassword сервісу ResetPasswordService:

```
public void sendEmailForChangingPassword(String email) {

    String token = generateResetPasswordToken(email);
    SimpleMailMessage mailMessage = new SimpleMailMessage();
    mailMessage.setTo(email);
    mailMessage.setSubject("Here's the link to reset your password");
    mailMessage.setFrom("zipli.socknet@gmail.com");
    mailMessage.setText("<p>Hello,</p>"
            + "<p>You have requested to reset your password.</p>"
            + "<p>Click the link below to change your password:</p>"
            + deploy + "/zipli/auth/reset_password?token=" + token + "<p>Change my
password</p>"
            + "<br>" + "<p>Ignore this email if you do remember your password, "
            + "or you have not made the request.</p>");
    javaMailSender.send(mailMessage);
}
```

Рисунок 3.15 – Метод sendEmailForChangingPassword

Тепер потрібно створити запит на який буде приймати посилання що відправилось на почту та змінювати пароль. Це буде:

```
@PostMapping("/reset_password")
public ResponseEntity<?> processResetPassword(@Valid @RequestBody
ResetPasswordRequest resetPasswordRequest) {
    try {
        resetPasswordService.resetPassword(resetPasswordRequest.getToken(),
resetPasswordRequest.getPassword());
    } catch (UserNotFoundException e) {
        log.error("Failed update password tokenIsNull {}, reason {}",
                Objects.isNull(resetPasswordRequest.getToken()),
e.getErrorStatusCode().getMessage());
        return ResponseEntity
                .badRequest()
                .body(new ErrorResponse(e.getErrorStatusCode()));
    }
    return ResponseEntity.ok("Password successfully changed");
}
```

Рисунок 3.16 – Метод processResetPassword

Приймає в себе ResetPasswordRequest, де перевіряється валідація:

```
@Getter
@Setter
@Validated
@NoArgsConstructor
@AllArgsConstructor
public class ResetPasswordRequest {
    private String token;

    @NotBlank(message = "Password can't be empty")
    @NotNull(message = "Password can't be null")
    @Size(min = 8, max = 16)
    @Pattern(regexp = "^(?=.*[0-9])(?=.*[a-z])(?=.*[A-Z])(?=\\S+$).{8,16}$")
    private String password;
}
```

Рисунок 3.17 – Клас ResetPasswordRequest

А метод класу resetPassword, змінює пароль у базі:

```
@Transactional
public String resetPassword(String token, String newPassword) {

    if (token == null) {
        throw new
UserNotFoundException(ErrorStatusCode.USER_DOES_NOT_EXIST);
    } else if (newPassword != null) {

        String userName = jwtUtils.getUserNameFromJwtToken(token);
        String codedPassword = passwordEncoder.encode(newPassword);
        userRepository.updatePasswordInUsersModel(userName, codedPassword);

        return "Password successfully changed";
    } else {
        throw new
UserNotFoundException(ErrorStatusCode.PASSWORD_IS_NULL);
    }
}
```

Рисунок 3.18 – Метод resetPassword

РОЗДІЛ 4. СТВОРЕННЯ REST КОНТРОЛЕРІВ ДЛЯ ОБЛІКОВОГО ЗАПИСУ

4.1 Створення REST контролера

За змінення даних облікового запису потрібно створити Rest контролери. На ці запити можливо постукатися тільки з авторизацією, а саме якщо в хедері передається токен і він валідний. За зміну даних облікового запису відповідає AccountController:

```
@RestController
@Slf4j
@RequestMapping("/zipli/myAccount")
public class AccountController {
    private final UserService userService;
    public AccountController(UserService userService) {
        this.userService = userService;
    }
    @GetMapping("/getUser/{userId}")
    public ResponseEntity<?> getUser(@PathVariable String userId) {
        try {
            return ResponseEntity.ok(new
UserInfoResponse(userService.findUser(userId)));
        } catch (GetUserException e) {
            log.error("Failed get user by userId {}, reason {}", userId,
e.getErrorStatusCode().getMessage());
            return ResponseEntity
                    .badRequest()
                    .body(new ErrorResponse(e.getErrorStatusCode()));
        }
    }
    @DeleteMapping("/delete-avatar/{userId}")
    public ResponseEntity<?> deleteAvatar(@PathVariable String userId) {
        try {
            return ResponseEntity.ok(new
UserInfoResponse(userService.deleteAvatar(userId)));
        } catch (DeleteAvatarException e) {
            log.error("Failed to delete avatar by userId {}, reason {}", userId,
e.getErrorStatusCode().getMessage());
            return ResponseEntity
                    .badRequest()
                    .body(new ErrorResponse(e.getErrorStatusCode()));
```

```
        }
    }
    @PutMapping("/update-avatar")
    public ResponseEntity<?> updateAvatar(@RequestBody @Valid
AvatarRequest data) {
        try {
            return ResponseEntity.ok(new
UserInfoResponse(userService.updateAvatar(data)));
        } catch (UpdateAvatarException e) {
            log.error("Failed update avatar by userId {}, avatarIsNull {}, reason {}",
                    data.getUserId(), Objects.isNull(data.getAvatar()),
e.getErrorStatusCode().getMessage());
            return ResponseEntity
                    .badRequest()
                    .body(new ErrorResponse(e.getErrorStatusCode()));
        }
    }
    @PutMapping("/update-nickName")
    public ResponseEntity<?> updateNickName(@RequestBody @Valid
NickNameRequest data) {
        try {
            return ResponseEntity.ok(new
UserInfoResponse(userService.updateNickName(data)));
        } catch (UpdateNickNameException e) {
            log.error("Failed update nickName by userId {}, nickName {}, reason {}",
                    data.getUserId(), data.getNickName(),
e.getErrorStatusCode().getMessage());
            return ResponseEntity
                    .badRequest()
                    .body(new ErrorResponse(e.getErrorStatusCode()));
        }
    }
    @PutMapping("/update-email")
    public ResponseEntity<?> updateEmail(@RequestBody @Valid EmailRequest
data) {
            return ResponseEntity.ok(new
UserInfoResponse(userService.updateEmail(data)));
    }
    @PutMapping("/update-password")
    public ResponseEntity<?> updatePassword(@RequestBody @Valid
PasswordRequest data) {
        try {
            return ResponseEntity.ok(new
UserInfoResponse(userService.updatePassword(data)));
```

```
        } catch (UpdatePasswordException e) {
            log.error("Failed update password by userId {}, passwordIsNull {}, reason
{}",
                    data.getUserId(), Objects.isNull(data.getPassword()),
e.getErrorStatusCode().getMessage());
            return ResponseEntity
                    .badRequest()
                    .body(new ErrorResponse(e.getErrorStatusCode()));
        }
    }
    @DeleteMapping("/delete-account/{userId}")
    public ResponseEntity<?> deleteAccount(@PathVariable String userId) {
        try {
            return ResponseEntity.ok(userService.deleteAccount(userId));
        } catch (DeleteAccountException e) {
            log.error("Failed delete user by userId {}, reason {}", userId,
e.getErrorStatusCode().getMessage());
            return ResponseEntity
                    .badRequest()
                    .body(new ErrorResponse(e.getErrorStatusCode()));
        }
    }
}
```

Рисунок 4.1 – Клас AccountController

@GetMapping("/getUser/{userId}") - повертає всі дані користувача по id

@DeleteMapping("/delete-avatar/{userId}") - видаляє автар користувача по id

@PutMapping("/update-avatar") - обновлює аватар користовуча по id та текстовому представленню аватара

@PutMapping("/update-nickName") - змінює нікнейм по id

@PutMapping("/update-email") - змінює пошту, потрібно підтвердити

@PutMapping("/update-password") - змінює пароль користувача

@DeleteMapping("/delete-account/{userId}") - видаляє особистий запис користувача

4.2 Обробка REST запитів для облікового запису

За логіку відповідає сервіс UserService:

```
@Service
public class UserService implements IUserService {

    private final UserRepository userRepository;
    private final EmailConfirmationService emailConfirmationService;
    private final JwtUtils jwtUtils;
    private final PasswordEncoder passwordEncoder;

    public UserService(UserRepository userRepository, EmailConfirmationService
emailConfirmationService, JwtUtils jwtUtils, PasswordEncoder passwordEncoder)
{
        this.userRepository = userRepository;
        this.emailConfirmationService = emailConfirmationService;
        this.jwtUtils = jwtUtils;
        this.passwordEncoder = passwordEncoder;
    }

    @Override
    @Transactional
    public User findUser(String userId) throws GetUserException {
        if (userId == null) {
            throw new GetUserException(ErrorStatusCode.USER_ID_NULL);
        }
        User user = userRepository.getUserById(userId);
        if (user == null) {
            throw new
GetUserException(ErrorStatusCode.USER_ID_DOES_NOT_CORRECT);
        }
        return user;
    }

    @Override
    @Transactional
    public User deleteAvatar(String userId) throws DeleteAvatarException {
        if (userId == null) {
            throw new DeleteAvatarException(ErrorStatusCode.USER_ID_NULL);
        }
        User user = userRepository.updateOrDeleteAvatar(userId, null);
        if (user == null) {
            throw new
DeleteAvatarException(ErrorStatusCode.USER_ID_DOES_NOT_CORRECT);
```

```
        }
        return user;
    }

    @Override
    @Transactional
    public User updateAvatar(AvatarRequest data) throws UpdateAvatarException
{
        if (data.getUserId() == null || data.getAvatar() == null) {
            throw new UpdateAvatarException(ErrorStatusCode.DATA_IS_NULL);
        }
        User user = userRepository.updateOrDeleteAvatar(data.getUserId(),
data.getAvatar());
        if (user == null) {
            throw new
UpdateAvatarException(ErrorStatusCode.USER_ID_DOES_NOT_CORRECT);
        }
        return user;
    }

    @Override
    @Transactional
    public User updateNickName(NickNameRequest data) throws
UpdateNickNameException {
        if (data.getUserId() == null || data.getNickName() == null) {
            throw new
UpdateNickNameException(ErrorStatusCode.DATA_IS_NULL);
        }
        User user = userRepository.updateNickName(data.getUserId(),
data.getNickName());
        if (user == null) {
            throw new
UpdateNickNameException(ErrorStatusCode.USER_ID_DOES_NOT_CORRECT)
;
        }
        return user;
    }

    @Override
    @Transactional
    public User updateEmail(EmailRequest data) throws UpdateEmailException {
        if (data.getUserId() == null || data.getEmail() == null) {
            throw new UpdateEmailException(ErrorStatusCode.DATA_IS_NULL);
        }
```

```
        if (!(data.getEmail().contains("@"))) {
            throw new
UpdateEmailException(ErrorStatusCode.EMAIL_DOES_NOT_CORRECT);
        }
        User existingUser = userRepository.getUserByEmail(data.getEmail());
        if (existingUser != null) {
            throw new
UpdateEmailException(ErrorStatusCode.EMAIL_ALREADY_EXISTS);
        }
        User user = userRepository.updateEmail(data.getUserId(), data.getEmail());
        if (user == null) {
            throw new
UpdateEmailException(ErrorStatusCode.USER_ID_DOES_NOT_CORRECT);
        }
        UserDetails userDetails = new UserDetailsImpl(user);
        String token = jwtUtils.generateJwtToken(userDetails, data.getEmail());
        emailConfirmationService.sendEmail(data.getEmail(), token);
        return user;
    }

    @Override
    @Transactional
    public User updatePassword(PasswordRequest data) throws
UpdatePasswordException {
        if (data.getUserId() == null || data.getPassword() == null) {
            throw new
UpdatePasswordException(ErrorStatusCode.DATA_IS_NULL);
        }
        User user = userRepository.updatePassword(data.getUserId(),
data.getPassword());
        if (user == null) {
            throw new
UpdatePasswordException(ErrorStatusCode.USER_ID_DOES_NOT_CORRECT);
        }
        return user;
    }

    @Override
    @Transactional
    public String deleteAccount(String userId) throws DeleteAccountException {
        if (userId == null) {
            throw new DeleteAccountException(ErrorStatusCode.USER_ID_NULL);
        }
        userRepository.deleteById(userId);
```

```
        return "Account deleted(";
    }

    @Override
    @Transactional
    public List<User> getUsersBySearchParam(String param) throws
SearchByParamsException {
        if (param == null) {
            throw new
SearchByParamsException(ErrorStatusCode.PARAM_IS_NULL);
        }
        String searchParam = param.trim();

        if (searchParam.length() < 3) {
            throw new
SearchByParamsException(ErrorStatusCode.PARAM_TOO_SHORT);
        }
        List<User> users = userRepository.findUsersBySearchParam(searchParam);
        if (users.isEmpty()) {
            throw new
SearchByParamsException(ErrorStatusCode.USERS_DOES_NOT_EXIST_BY_PA
RAM);
        }
        return users;
    }
}
```

Рисунок 4.2 – Клас UserService

РОЗДІЛ 5. РЕАКТИВНИЙ WEBSOCKET

5.1 Створення реактивного websocket

Збираюся використовувати нове API Spring 5 WebSockets разом з реактивними функціями, що забезпечують Spring WebFlux.

WebSocket - це добре відомий протокол, який забезпечує полнодуплексную зв'язок між клієнтом і сервером, зазвичай використовується у веб-додатках, де клієнтам і серверам необхідно змінювати події з високою частотою і з низькою заробкою.

Spring Framework 5 модернізував підтримку WebSockets у фреймворці, додавши реактивні можливості до цього каналу зв'язку.

Використовуємо залежності spring-boot -starters для spring-boot-integration та spring-boot-starter-webflux, які зараз доступні у репозиторіях Spring Milestone.

Використовуючи останню доступну версію 2.1.1, але завжди потрібно отримувати останню версію, доступну в репозиторіях Maven:

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

Рисунок 5.1 – Залежність Maven

Добавимо WebSocketHandler для обробки сеансу сокета у нашому додатку Spring WebSocket.

```
private final EventHandler eventHandler;
private final IEmitterService emitterService;

public WebSocketSessionHandler(IEmitterService emitterService, EventHandler
eventHandler) {
   this.emitterService = emitterService;
   this.eventHandler = eventHandler;
}
```

Рисунок 5.2 – Клас WebSocketSessionHandler

Крім того, створений HandlerMapping анотований метод, який буде відповідати для відображення між запрошеннями та обробниками об'єктів:

```
@Bean
public HandlerMapping             () {
   Map<String, WebSocketHandler> map = new HashMap<>();
   map.put("/ws", webSocketHandler);

   SimpleUrlHandlerMapping handlerMapping = new
SimpleUrlHandlerMapping();
   handlerMapping.setOrder(1);
   handlerMapping.setUrlMap(map);
   return handlerMapping;
}
```

Рисунок 5.2 – Метод webSocketHandlerMapping

URL-адрес, к которому мы можем подключиться, будет: *ws: // localhost: <port> / ws*

Наш клас ReactiveWebSocketHandler відповість за управління сеансом WebSocket на стороні сервера.

Він реалізує інтерфейс WebSocketHandler, тому ми можемо перевизначити дескриптор методу, який буде використовуватися для відправки повідомлень клієнтам WebSocket::

```
@Slf4j
@Component
public class WebSocketSessionHandler implements WebSocketHandler {
   private final EventHandler eventHandler;
   private final IEmitterService emitterService;

   @Override
   public Mono<Void> handle(WebSocketSession webSocketSession) {
      String token =
webSocketSession.getHandshakeInfo().getUri().getQuery().replace("token=", "");
      Sinks.Many<String> publisher =
Sinks.many().multicast().directAllOrNothing();
      final String userId;
      try {
         userId = emitterService.addMessageEmitterByToken(token, publisher);
      } catch (CreateSocketException e) {
         String response = JsonUtils.jsonWriteHandle(new
WsMessageResponse(ERROR_CREATE_CONNECT, e.getMessage()));
         return
webSocketSession.send(Mono.just(webSocketSession.textMessage(response)));
```

```
        }

        return Mono.zip(
                webSocketSession.receive()
                        .doOnNext(message -> {
                            try {
                                eventHandler.process(publisher,
JsonUtils.json.readValue(message.getPayloadAsText(), WsMessage.class));
                            } catch (Throwable e) {
                                log.error("Error processing message {} error class {} error
message {} ", message, e.getClass().getSimpleName(), e.getMessage());
                            }
                        })
                        .doOnComplete(() -> {
                            try {
                                emitterService.deleteMessageEmitterByUserId(userId,
publisher);
                            } catch (DeleteSessionException e) {
                                log.error("Error delete session {} for userId {} error class {}
error message {} ",
                                        webSocketSession, userId, e.getClass().getSimpleName(),
e.getMessage());
                            }
                        }).then(),

                webSocketSession.send(
                        publisher.asFlux()
                                .map(webSocketSession::textMessage)
                )).then();
    }
```

Рисунок 5.3 – Клас WebSocketSessionHandler

При логине в вебсокет, буду зберігати подписки у паралельну мапу еміторів:

```
private final Map<String, List<Sinks.Many<String>>> messageEmitterByUserId
= new ConcurrentHashMap<>();
```

Рисунок 5.4 – MessageEmitterByUserId

За це відповідає метод addMessageEmitterByToken() сервісу EmitterService:

```
@Override
public String addMessageEmitterByToken(String token, Sinks.Many<String>
emitter) throws CreateSocketException {
    try {
        String username = jwtUtils.getUserNameFromJwtToken(token);
        User user = userRepository.getUserByUserName(username);
```

```
        String userId = user.getId();
        messageEmitterByUserId.computeIfAbsent(userId, e -> new
CopyOnWriteArrayList<>()).add(emitter);
        return userId;
    } catch (Exception e) {
        throw new CreateSocketException("Can't create connect to user, Exception
cause: " + e.getMessage() + " on class " + e.getClass().getSimpleName());
    }
}
```

Рисунок 5.5 – Метод addMessageEmitterByToken

Якщо конекшен с клієнтом пропадає, потрібно видаляти її з мапи еміторів, це робить метод deleteMessageEmitterByUserId():

```
@Override
public void deleteMessageEmitterByUserId(String userId, Sinks.Many<String>
emitter) throws DeleteSessionException {
    try {
        messageEmitterByUserId.getOrDefault(userId, new
CopyOnWriteArrayList<>()).remove(emitter);
    } catch (Exception e) {
        throw new DeleteSessionException("Can't delete message emitter");
    }
}
```

Рисунок 5.6 – Метод deleteMessageEmitterByUserId

Якщо конекшен є, та приходить меседж на бекенд. Передаємо його в метод:

```
eventHandler.process(publisher, JsonUtils.json.readValue(
        message.getPayloadAsText(), WsMessage.class)
        );
```

Про який будемо говорити чуть пізніше.

5.2 Команди Websocket та їх обробка

За обробку команд вебСокету відповідає клас EventHandler, який відправляє команди на сервіси та помилки.

Сервіси:

IMessageService messageService – відповідає за логіку відправки\зміненя\ видалення повідомлень у чаті

IFileService fileService – відповідає за логіку відправки\видалення файлов(відео\картинки) у чаті

IChatService chatService – відповідає за логіку створення\зміни\видалення чатів

IVideoService videoService – відповідає за логіку відеодзвіноків

Клас EventHandler и його єдиний метод process():

```
public void process(Sinks.Many<String> emitter, WsMessage message) {
   Command eventCommand = message.getCommand();

   switch (eventCommand) {
      case CHAT_CREATE:
         FullChatData chatData = (FullChatData) message.getData();
         try {
            chatService.createChat(chatData);
         } catch (CreateChatException e) {
            log.error(commandFail, eventCommand, chatData,
e.getErrorStatusCode().getMessage());
            emitter.tryEmitNext(JsonUtils.jsonWriteHandle(
                  new WsMessageResponse(eventCommand,
                     new ErrorResponse(e.getErrorStatusCode())))
            );
         } catch (UserNotFoundException e) {
            log.error(commandFail, eventCommand, chatData,
e.getErrorStatusCode().getMessage());
            emitter.tryEmitNext(JsonUtils.jsonWriteHandle(
                  new WsMessageResponse(eventCommand,
                     new ErrorResponse(e.getErrorStatusCode())))
            );
         }
         break;

                  //.....і так далі для кожного сервісу та команди

          }
```

Рисунок 5.7 – Метод process

Усі команди можна побачити у класі типу перерахування:

```
public enum Command {
   CHAT_CREATE,
   CHAT_UPDATE,
   CHAT_DELETE,
```

```
    CHAT_LEAVE,
    CHAT_USER_ADD,
    CHATS_GET_BY_USER_ID,
    MESSAGE_SEND,
    MESSAGE_READ,
    MESSAGE_UPDATE,
    MESSAGE_DELETE,
    MESSAGES_GET_BY_CHAT_ID,
    ERROR_CREATE_CONNECT,
    FILE_SEND,
    FILE_DELETE,
    VIDEO_CALL_START,
    VIDEO_CALL_JOIN,
    VIDEO_CALL_EXIT,
    USERS_GET_BY_SEARCH_PARAM
}
```

Рисунок 5.8 – Клас Command

5.3 Сервіс повідомлень

Для початку треба створити метод який буде повертати список всіх повідомлень по чату. Перевіряємо чи існує такий чат та повертаємо всі повідомлення по ньому:

```
@Override
public List<Message> getMessages(ChatData data) throws GetMessageException
{
    Chat chat = chatRepository.findChatById(data.getChatId());
    if (chat != null) {
        List<String> listIdMessages = chat.getMessagesId();
        List<Message> messages = new ArrayList<>();
        for (String idMessage : listIdMessages) {
            messages.add(messageRepository.getMessageById(idMessage));
        }
        log.info("Get messages {} In chat:{} ", data.getUserId(), data.getChatId());
        return messages;
    } else {
        throw new GetMessageException(ErrorStatusCode.CHAT_NOT_EXISTS);
    }
}
```

Рисунок 5.9 – Метод getMessages

Тепер потрібно створити метод який буде відправляти повідомлення у чат та розсилати усім користувачам які є в тому чаті та онлайн:

```
@Override
public Message sendMessage(MessageData data) throws MessageSendException,
ChatNotFoundException {
   Message message = new Message(data.getUserId(), data.getChatId(),
data.getTimestamp(), data.getTextMessage());
   final Message finalMessage = messageRepository.save(message);
   Chat chat = chatRepository.findChatById(data.getChatId());
   if (chat != null) {
      chat.getMessagesId().add(message.getId());

      chat.getUsersId().parallelStream()
            .forEach(userId -> emitterService.sendMessageToUser(userId,
                  new WsMessageResponse(Command.MESSAGE_SEND,
                        new MessageData(data.getUserId(),
                              chat.getId(),
                              finalMessage.getId(),
                              finalMessage.getTextMessage(),
                              finalMessage.getDate()
                        )
                  ))
            );
      chatRepository.save(chat);
      return message;
   } else {
      throw new ChatNotFoundException(ErrorStatusCode.CHAT_NOT_EXISTS);
   }
}
```

Рисунок 5.10 – Метод sendMessage

Для відправки Ws повідомлення всім хто є в чаті та онлайн, потрібно дістати всі Id в чаті порівняти з тими хто онлайн та вислати їм повідомлення:

```
chat.getUsersId().parallelStream()
      .forEach(userId -> emitterService.sendMessageToUser(userId,
            new WsMessageResponse(Command.MESSAGE_SEND,
                  new MessageData("повідомлення")
      );
```

Нове повідомлення зберігається в базу:

```
chatRepository.save(chat);
```

Змінення та видалення повідомлення в чаті та видалення не сильно відрізняються. Так само ходимо в базу, але змінюємо\видаляємо повідомлення та розсилаємо івент.

Метод видалення повідомлення:

```
@Override
public void deleteMessage(MessageData data) throws MessageDeleteException,
UpdateChatException {

   Message message =
messageRepository.getMessageByIdAndAuthorId(data.getMessageId(),
data.getUserId());

   if (message != null) {
      Chat chat = chatRepository.findChatById(data.getChatId());
      if (chat != null) {
         chat.getMessagesId().remove(message.getId());
         final Chat finalChat = chatRepository.save(chat);
         finalChat.getUsersId().parallelStream()
               .forEach(userId -> emitterService.sendMessageToUser(userId,
                     new WsMessageResponse(Command.MESSAGE_DELETE,
                           new MessageData(data.getUserId(),
                                 finalChat.getId(),
                                 message.getId(),
                                 message.getTextMessage(),
                                 message.getDate()
                           )
                     ))
               );
      } else {
         throw new
ChatNotFoundException(ErrorStatusCode.CHAT_NOT_EXISTS);
      }
      messageRepository.delete(message);
   } else {
      throw new
MessageDeleteException(ErrorStatusCode.MESSAGE_ACCESS_ERROR);
   }
}
```

Рисунок 5.11 – Метод deleteMessage

Метод обновлення повідомлення:

```
@Override
public Message updateMessage(MessageData data) throws
MessageUpdateException, ChatNotFoundException {
   final Chat finalChat = chatRepository.findChatById(data.getChatId());
   if (finalChat != null) {
      final Message finalMessage = messageRepository.updateMessage(data);
      if (finalMessage != null) {
         log.info("UpdateMessage with userId {}  to chat {} with author {} new
message {} ", data.getUserId(), finalMessage.getChatId(),
finalMessage.getAuthorId(), finalMessage.getId());

         finalChat.getUsersId().parallelStream()
               .forEach(userId -> emitterService.sendMessageToUser(userId,
                     new WsMessageResponse(Command.MESSAGE_UPDATE,
                           new MessageData(data.getUserId(),
                                 finalChat.getId(),
                                 finalMessage.getId(),
                                 finalMessage.getTextMessage(),
                                 finalMessage.getDate()
                           )
                     ))
               );
         return finalMessage;
      } else {
         throw new
MessageUpdateException(ErrorStatusCode.CHAT_ACCESS_ERROR);
      }
   } else {
      throw new ChatNotFoundException(ErrorStatusCode.CHAT_NOT_EXISTS);
   }
}
```

Рисунок 5.12 – Метод updateMessage

5.4 Сервіс чатів

Сервіс чатів реалізовує усі методи інтерфейсу IChatService:

```
public interface IChatService {

   Chat createChat(FullChatData data) throws CreateChatException,
UserNotFoundException;

   Chat updateChat(FullChatData data) throws UpdateChatException;

   void deleteChat(ChatData data) throws DeleteChatException;
```

```
    List<Chat> showChatsByUser(BaseData data);

    Chat leaveChat(ChatData data) throws LeaveChatException;

    Chat joinChat(ChatData data) throws JoinChatException;

}
```

Рисунок 5.13 – Клас IChatService

Методи які реалізовані в ChatService:

- Chat createChat(FullChatData data) – створюємо чат який та розсилаємо запрошення користувачам яких запросили. Повертаємо модель чату.
- Chat updateChat(FullChatData data) – змінює настройки чата, також розсилаємо нові дані всім онлайн користовучам
- void deleteChat(ChatData data) – видаляє чат та всі повідомлення в ньому
- List<Chat> showChatsByUser(BaseData data) – повертає користувачу всі чати в яких він знаходиться
- Chat leaveChat(ChatData data) – приходить id користувача та id чату. Користувач покидає чат, усі хто знаходяться в чаті отримують повідомлення
- Chat joinChat(ChatData data) – користувач приймає запрошення до чату

5.5 Сервіс відео дзвінків

Для дзвінків буду використовувати технологію WEBRTC, а що це?

Коли двом браузерам необхідно обмінюватися даними, їм зазвичай потрібно проміжний сервер, щоб координувати обмін даними і передавати повідомлення між ними. Але наявність сервера посередині призводить до затримки зв'язку між браузерами.

WebRTC, проект з відкритим вихідним кодом, який дозволяє браузерам і мобільних додатків безпосередньо взаємодіяти один з одним в режимі реального часу.

Подивимося, як два браузера спілкуються в типовому сценарії без WebRTC. Припустимо, у нас є два браузера, і Браузер 1 повинен відправити повідомлення Браузеру 2. Браузер 1 спочатку відправляє його на Сервер:

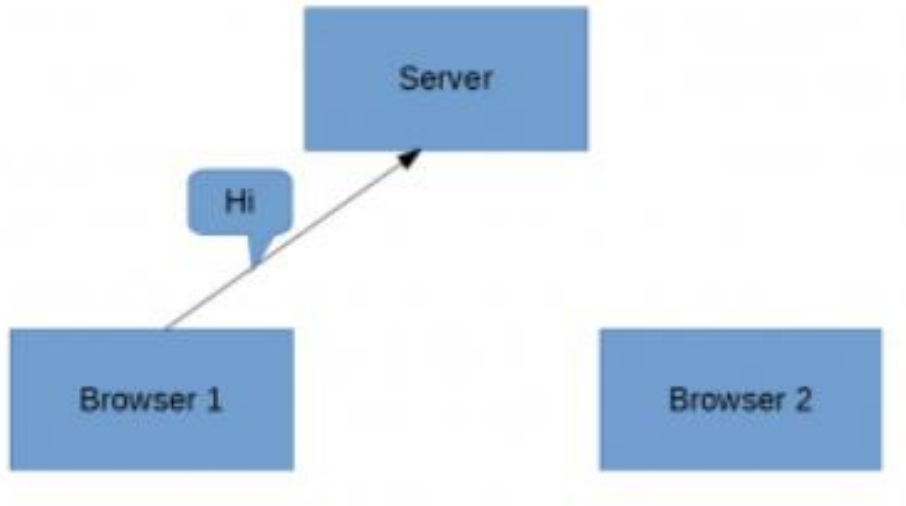

Рисунок 5.14 – Перша схема браузерів

Після того, як Сервер отримує повідомлення, він обробляє його, знаходить Браузер 2 і відправляє йому повідомлення:

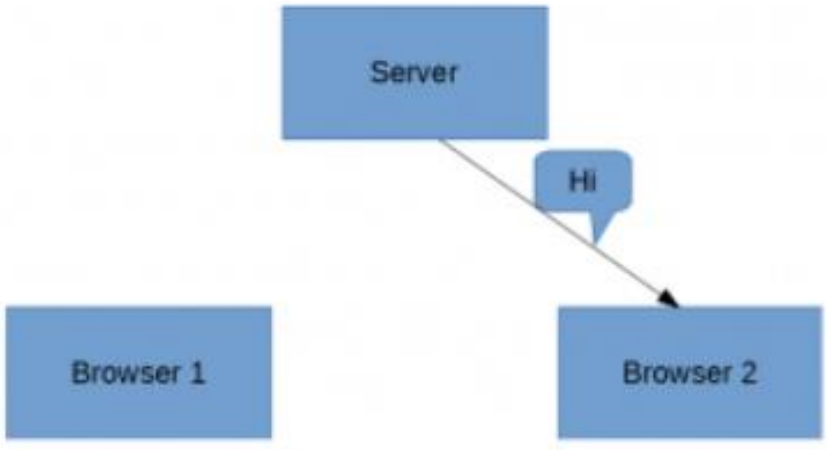

Рисунок 5.14 – Друга схема браузерів

Оскільки сервер повинен обробити повідомлення перед його відправкою в браузер 2, зв'язок відбувається майже в реальному часі. Звичайно, хотілося б, щоб це відбувалося в режимі реального часу.

WebRTC вирішує цю проблему, створюючи прямий канал між двома браузерами, усуваючи необхідність в сервері:

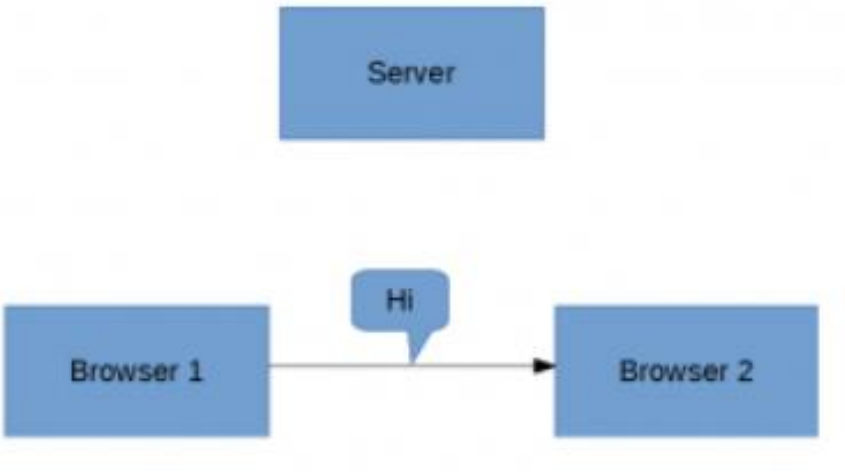

Рисунок 5.15 – Третя схема браузерів

В результаті час, необхідне для передачі повідомлень від одного браузера до іншого, значно скорочується, оскільки тепер повідомлення маршрутизируются безпосередньо від відправника до одержувача. Це також позбавляє сервери від важкої роботи і пропускної спроможності і забезпечує їх спільне використання між задіяними клієнтами. Однак все одно браузерам треба знати де знаходяться інші браузери для цього і потрібен буде сервер.

WebRTC підтримується основними браузерами, такими як Chrome, Firefox, Opera і Microsoft Edge, а також такими платформами, як Android і iOS.WebRTC не вимагає установки яких-небудь зовнішніх плагінів в нашому браузері, оскільки рішення поставляється в комплекті з браузером.

Більш того, в типовому додатку реального часу, що включає передачу відео і звуку, ми повинні сильно залежати від бібліотек C ++ і вирішувати безліч проблем, в тому числі:

- Маскування втрати пакетів
- Ехоподавлення
- Адаптивність смуги пропускання
- Динамічна буферизация джиттера
- Автоматичне регулювання посилення

- Зниження і придушення шуму
- «Очищення» зображення

Але WebRTC вирішує всі ці проблеми під капотом, спрощуючи обмін даними між клієнтами в режимі реального часу.

На відміну від взаємодії клієнт-сервер, коли існує відомий адресу сервера, і клієнт вже знає адресу сервера для зв'язку, в P2P (вибрали тимчасовий) з'єднанні жоден з вузлів не має прямого адреси. іншого партнера.

Щоб встановити однорангове з'єднання, необхідно виконати кілька кроків, що дозволяють клієнтам:

- зробити себе доступними для спілкування
- ідентифікують один одного і обмінюються мережевий інформацією
- поділитися і узгодити формат використовуваних даних, режим і протоколи
- ділитися даними

WebRTC визначає набір API і методологій для виконання цих кроків.

Щоб клієнти могли виявляти один одного, обмінюватися даними про мережі, а потім обмінюватися форматом даних, WebRTC використовує механізм, званий сигналізацією.

Під сигналізацією розуміються процеси, пов'язані з виявленням мережі, створенням сеансу, управлінням сеансом і обміном метаданими мультимедійних можливостей.

Це важливо, оскільки клієнти повинні знати один одного заздалегідь, щоб почати спілкування.

Щоб досягти всього цього, WebRTC не визначає стандарт сигналізації і залишає це на розсуд розробника. Таким чином, це дає нам можливість використовувати WebRTC на різних пристроях з будь-якою технологією і підтримує протоколом.

Для сигналізації використовуємо класс VideoService а саме метод startVideoCall(VideoData videoData):

```
public VideoData startVideoCall(VideoData videoData) {
   VideoCallState videoCallState = new VideoCallState(videoData.getUserId(),
new CopyOnWriteArrayList<>(), new CopyOnWriteArrayList<>());
   videoCallState.getUsersInCallId().add(videoData.getUserId());
   videoCallStorage.put(videoData.getChatId(), videoCallState);

   Chat chat = chatRepository.findChatById(videoData.getChatId());
   if (chat == null) {
      throw new ChatNotFoundException(ErrorStatusCode.CHAT_NOT_EXISTS);
   }
   chat.getUsersId().parallelStream()
         .forEach(userId -> emitterService.sendMessageToUser(userId,
               new WsMessageResponse(Command.VIDEO_CALL_START,
videoData)));

   List<String> usersWhoIsNotOnline =
videoCallStorage.get(videoData.getChatId()).getUsersWhoIsNotOnlineId();
   usersWhoIsNotOnline.addAll(chat.getUsersId());

usersWhoIsNotOnline.removeAll(emitterService.getMessageEmitter().keySet());

   log.info("Start VideoCall in Chat {} by user {}.", videoData.getChatName(),
videoData.getUserName());
   log.debug(videoCallStorage.get(videoData.getChatId()).toString());

   return videoData;
}
```

Рисунок 5.16 – Метод startVideoCall

З клієнта приходить VideoData, в якій є той сигнал що потрібен для створення конекшену між браузерами. Ми його висилаємо усім хто онлайн з чату:

```
chat.getUsersId().parallelStream()
      .forEach(userId -> emitterService.sendMessageToUser(userId,
            new WsMessageResponse(Command.VIDEO_CALL_START,
videoData)));
```

Також коли хтось підключається до конекшену ми повинні, розіслати повідомлення що він підключився:

```
public VideoData joinVideoCall(VideoData videoData) {
   VideoCallState videoCallState = videoCallStorage.get(videoData.getChatId());
   if (videoCallState == null) {
```

```
        throw new
VideoCallException(ErrorStatusCode.VIDEO_CALL_EXCEPTION);
    }
    videoCallState.getUsersInCallId()
            .add(videoData.getUserId());
    videoCallState.getUsersWhoIsNotOnlineId().remove(videoData.getUserId());
    Chat chat = chatRepository.findChatById(videoData.getChatId());
    if (chat == null) {
        throw new ChatNotFoundException(ErrorStatusCode.CHAT_NOT_EXISTS);
    }

    chat.getUsersId().parallelStream()
            .forEach(userId -> emitterService.sendMessageToUser(userId,
                    new WsMessageResponse(Command.VIDEO_CALL_JOIN,
videoData)));

    log.info("User {} successfully joined videoCall in Chat {}.",
videoData.getUserName(), videoData.getChatName());
    log.debug(videoCallState.toString());

    return videoData;
}
```

Рисунок 5.16 – Метод joinVideoCall

Та коли хтось покидає його:

```
public ChatData exitFromVideoCall(ChatData chatData) {
    VideoCallState videoCallState = videoCallStorage.get(chatData.getChatId());
    videoCallState.getUsersInCallId()
            .remove(chatData.getUserId());
    log.debug(videoCallState.toString());
    log.info("User {} leaved from videoCall in chat {}", chatData.getUserId(),
chatData.getChatId());

    if (videoCallStorage
            .get(chatData.getChatId())
            .getUsersInCallId()
            .size() < 2) {
        videoCallStorage.remove(chatData.getChatId());
    }
    Chat chat = chatRepository.findChatById(chatData.getChatId());
    if (chat == null) {
        throw new ChatNotFoundException(ErrorStatusCode.CHAT_NOT_EXISTS);
    }
```

```
    chat.getUsersId().parallelStream()
            .forEach(userId -> emitterService.sendMessageToUser(userId,
                    new WsMessageResponse(Command.VIDEO_CALL_EXIT,
chatData)));
    return chatData;
}
```

Рисунок 5.17 – Метод exitFromVideoCall

РОЗДІЛ 6. АНОНІМНІ КІМНАТИ ДЛЯ ВІДЕО ДЗВІНКІВ НА SSE

6.1 Що таке SSE

Для анонімних відео дзвінків буду використовувати реактивний протокол SSE(server-sent-events).

Простіше кажучи, Server-Sent-Events або скорочено SSE - це стандарт HTTP, який дозволяє веб-додатку обробляти односпрямований потік подій і отримувати оновлення кожного разу, коли сервер передає дані.

Для досягнення цієї мети можемо використовувати реалізації, такі як Flux клас, представлений Reactor бібліотеку, або потенційно ServerSentEvent суті, яка дає нам контроль над метаданими подій.

Але для початку почнемо створювати потрібні нам реактивні контролери:

```
@Bean
public RouterFunction<ServerResponse> route(RoomHandler roomHandler) {
   return RouterFunctions
         .route(RequestPredicates
                     .GET(PATH + "/getRoom/{roomId}")
.and(RequestPredicates.accept(MediaType.APPLICATION_JSON)),
               roomHandler::getRoom)
         .andRoute(RequestPredicates
                     .POST(PATH + "/leaveRoom/{roomId}")
.and(RequestPredicates.accept(MediaType.APPLICATION_JSON)),
               roomHandler::leaveRoom)
         .andRoute(RequestPredicates
                     .POST(PATH + "/createRoom")
.and(RequestPredicates.accept(MediaType.APPLICATION_JSON)),
               roomHandler::createRoom)
         .andRoute(RequestPredicates
                     .POST(PATH + "/joinRoom/{roomId}")
.and(RequestPredicates.accept(MediaType.APPLICATION_JSON)),
               roomHandler::joinRoom)
         .andRoute(RequestPredicates
                     .GET(PATH + "/getRooms")
.and(RequestPredicates.accept(MediaType.APPLICATION_JSON)),
               roomHandler::getRooms)
         .andRoute(RequestPredicates
                     .POST(PATH + "/deleteRoom/{roomId}")
.and(RequestPredicates.accept(MediaType.APPLICATION_JSON)),
```

```
                roomHandler::deleteRoom)
        .andRoute(RequestPredicates
                .POST(PATH + "/newMessage/{roomId}")
.and(RequestPredicates.accept(MediaType.APPLICATION_JSON)),
                roomHandler::saveMessage);
```

Рисунок 6.1 – Метод route

6.2 Обробка SSE запитів

Клас для обробки RoomHandler в якому реалізовані усі методи інтерфейсу:

```
public interface IRoomHandler {
   Mono<ServerResponse> getRoom(ServerRequest request);
   Mono<ServerResponse> getRooms(ServerRequest request);
   Mono<ServerResponse> joinRoom(ServerRequest request);
   Mono<ServerResponse> leaveRoom(ServerRequest request);
   Mono<ServerResponse> deleteRoom(ServerRequest request);
   Mono<ServerResponse> createRoom(ServerRequest request);
   Mono<ServerResponse> saveMessage(ServerRequest request);
   Mono<ServerResponse> subscribeMessage(ServerRequest request);
   Mono<ServerResponse> getMessagesByRoom(ServerRequest request);
}
```

Рисунок 6.2 – Клас IRoomHandler

GetRoom() – відає кімнату по її id.

GetRooms() – відає всі існуючі кімнати.

SubscribeMessage() – підписка по SSE на повідомлення кімнати:

```
@Override
public Flux<ServerSentEvent<BaseEventResponse>> subscribeMessage(String
idRoom) {
   return messageEmitterByRoomId.get(idRoom).asFlux();
}
```

Рисунок 6.3 – Метод subscribeMessage

Для підписки добавляємо idRoom в:

```
private final Map<String, Sinks.Many<ServerSentEvent<BaseEventResponse>>>
messageEmitterByRoomId = new ConcurrentHashMap<>();
```

Рисунок 6.4 – MessageEmitterByRoomId

Тепер при JoinRoom(), користувачу почнуть приходити повідомлення:

```
public Room joinRoom(String roomId, UserInfoByRoomRequest
userInfoByRoomRequest) throws JoinRoomException {
   Optional<Room> roomOptional = roomRepository.findById(roomId);
```

```
    if (roomOptional.isPresent()) {
        Room room = roomOptional.get();
        room.getUsersInfo().add(userInfoByRoomRequest);
        room = roomRepository.save(room);
messageEmitterByRoomId.get(roomId).tryEmitNext(ServerSentEvent.<BaseEvent
Response>builder()
.id(String.valueOf(eventIdGeneration.get(room.getId()).getAndIncrement()))
                .event(EventCommandRoom.JOIN_ROOM_EVENT.name())
                .data(new
RoomEventResponse(userInfoByRoomRequest.getUserName(),
                        userInfoByRoomRequest.getSignal()))
                .build());
        log.info("Join Room successful: Room - {}, user - {}, number of users - {}",
                room.getId(),
                userInfoByRoomRequest.getUserName(),
                room.getUsersInfo().size()
        );
        return room;
    } else {
        throw new JoinRoomException(ErrorStatusCode.ROOM_NOT_EXIT);
    }
```

Рисунок 6.5 – Метод joinRoom

За відправку повідомлення по SSE відповідає:

```
messageEmitterByRoomId.get(roomId).tryEmitNext(ServerSentEvent.<BaseEvent
Response>builder()
        .id(String.valueOf(eventIdGeneration.get(room.getId()).getAndIncrement()))
        .event(EventCommandRoom.JOIN_ROOM_EVENT.name())
        .data(new RoomEventResponse(userInfoByRoomRequest.getUserName(),
                userInfoByRoomRequest.getSignal()))
        .build());
```

Рисунок 6.6 – Зідправка SSE повідомлення

Знаходимо туку підписку та відправляємо повідомлення усім хто підписна на цю кімнату.

LeaveRoom(ServerRequest request) – відписка від кімнати.

DeleteRoom(ServerRequest request) – видалення кімнати та відписка всіх хто в ній є.

SaveMessage(ServerRequest request) – відправка повідомлення усім підписникам кімнати.

GetMessagesByRoom(ServerRequest request) – отримати всі повідомлення в кімнаті.

ВИСНОВОК

Реактивні системи володіють певними характеристиками, які роблять їх ідеальними для робочих навантажень із низькою задержкою та високою пропускною здатністю. Project Reactor та Spring працюють разом, щоб дозволити розробникам створювати реактивні системи корпоративного рівня, які будуть чуйними, відмовостійкими, еластичними і керованими повідомленнями.

Реактивне програмування зараз дуже популярне та дає великі можливості для оптимізації\зменшення нагрузки на сервер. Сворив реактивний сервер який повністю асінхронний та реативний:

- реактивні контролери
- реактивний захист
- реактивний вебСокет
- реактивний протокол SSE
- реактивний SpringBoot

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

- **https://www.baeldung.com/spring-server-sent-events**
- **https://docs.spring.io/spring-framework/docs/current/reference/html/web-reactive.html**
- **https://www.baeldung.com/spring-webflux**
- **https://medium.com/@kirill.sereda/reactive-programming-reactor-%D0%B8-spring-webflux-3f779953ed45**
- **https://bezkoder.com/spring-boot-jwt-auth-mongodb/**
- **https://docs.spring.io/spring-security/site/docs/current/reference/html5/**
- **https://www.baeldung.com/spring-boot-reactor-netty**
- **https://start.spring.io/**
- **https://habr.com/ru/post/278411/**
- **https://www.baeldung.com/spring-5-reactive-websockets**

ДОДАТОК А

Приклади успішних WS команд

WS REQUEST	WS RESPONSE

{
"command":"CHAT_CREATE",
"data":{
"userId":"6006c07a9da711035eec5958",
"chatName":"SuperPublicTESTChat",
"chatParticipants":[
"6006c1039da711035eec5959"
],
"isPrivate":false
}
}
	{
	"command":"CHAT_CREATE",
	"data":{
	"userId":"6006c07a9da711035eec5958",
	"chatId":"6007fe99416f8f740825a651",
	"chatName":"SuperPublicTESTChat",
	"chatParticipants":[
	"6006c1039da711035eec5959",
	"6006c07a9da711035eec5958"
	]
	}
	}

{
"command":"CHAT_USER_ADD",
"data":{
"userId":"60081199f4170d1c216c2244", - кого нужно добавить
"chatId":"6007fe99416f8f740825a651"
}
}
	{
	"command":"CHAT_USER_ADD",
	"data":{
	"userId":"60081199f4170d1c216c2244",
	"chatId":"6007fe99416f8f740825a651",
	"chatName":"SuperPublicTESTChat",
	"chatParticipants":[
	"6006c1039da711035eec5959",
	"6006c07a9da711035eec5958",
	"60081199f4170d1c216c2244"]
	}
	}

{
"command":"CHAT_LEAVE", - для CHAT_DELETE такая же дата
"data":{
"userId":"6006c1039da711035eec5959",
"chatId":"6007fe99416f8f740825a651"
}
}
	{
	"command":"CHAT_LEAVE",
	"data":{
	"userId":"6006c1039da711035eec5959",
	"chatId":"6007fe99416f8f740825a651",
	"chatName":"SuperPublicTESTChat",
	"chatParticipants":[
	"6006c07a9da711035eec5958",
	"60081199f4170d1c216c2244"
	]
	}
	}

<table>
<tr>
<td>

```
{
  "command":"CHATS_GET_BY_USER_ID",
  "data":{
    "userId":"6006c07a9da711035eec5958"
  }
}
```

</td>
<td>

```
{
  "command":"CHATS_GET_BY_USER_ID",
  "data":[
    {
      "id":"6006c3249da711035eec595a",
      "chatName":"GroupTestChatName",
      "idMessages":[
        "6006c92372d26e181616aa68",
        "6006c96572d26e181616aa69",
        "6006f76872d26e181616aa6a",
        "6006f76d72d26e181616aa6b",
        "6006f76e72d26e181616aa6c"
      ],
      "idUsers":[
        "6006c1039da711035eec5959",
        "6006c07a9da711035eec5958"
      ],

"creatorUserId":"6006c07a9da711035eec5958"
    },
    {
      "id":"6007fe99416f8f740825a651",
      "chatName":"OtherName",
      "idUsers":[
        "6006c07a9da711035eec5958",
        "60081199f4170d1c216c2244"
      ],

"creatorUserId":"6006c07a9da711035eec5958"
    }
  ]
}
```

</td>
</tr>
<tr>
<td>

```
{
  "command":"CHAT_UPDATE",
  "data":{
    "userId":"6006c07a9da711035eec5958",
    "chatId":"6007fe99416f8f740825a651",
    "chatName":"OtherName"
  }
}
```

</td>
<td>

```
{
  "command":"CHAT_UPDATE",
  "data":{
    "userId":"6006c07a9da711035eec5958",
    "chatId":"6007fe99416f8f740825a651",
    "chatName":"OtherName",
    "chatParticipants":[
      "6006c07a9da711035eec5958",
      "60081199f4170d1c216c2244"
    ]
  }
}
```

</td>
</tr>
</table>

{
"command":"MESSAGE_SEND",
"data":{
"userId":"6006c07a9da711035eec5958",
"chatId":"6007fe99416f8f740825a651",
"textMessage":"Hello World!",
"timestamp":"123456786"
}
}
	{
	"command":"MESSAGE_SEND",
	"data":{
	"userId":"6006c07a9da711035eec5958",
	"chatId":"6007fe99416f8f740825a651",
	"messageId":"600814d6f4170d1c216c2245",
	"textMessage":"Hello World!",
	"timestamp":123456786
	}
	}

{
"command":"MESSAGE_DELETE",
"data":{
"userId":"6006c07a9da711035eec5958",
"chatId":"6007fe99416f8f740825a651",
"messageId":"6008151ef4170d1c216c2247"
}
}
	{
	"command":"MESSAGE_DELETE",
	"data":{
	"userId":"6006c07a9da711035eec5958",
	"chatId":"6007fe99416f8f740825a651",
	"messageId":"6008151ef4170d1c216c2247",
	"textMessage":"Hello World!",
	"timestamp":123456786
	}
	}

{
"command":"MESSAGE_UPDATE",
"data":{
"userId":"6006c07a9da711035eec5958",
"chatId":"6007fe99416f8f740825a651",
"messageId":"600814d6f4170d1c216c2245",
"textMessage":"Привет Мир!"
}
}
	{
	"command":"MESSAGE_UPDATE",
	"data":{
	"userId":"6006c07a9da711035eec5958",
	"chatId":"6007fe99416f8f740825a651",
	"messageId":"600814d6f4170d1c216c2245",
	"textMessage":"Привет Мир!",
	"timestamp":123456786
	}
	}

{
"command":"VIDEO_CALL_START",
"data":{
"idUser":"5ffc8765a11fd",
"idChat":"5fff461dcf536",
"userName":"AndrewZ",
"chatName":"GroupChatNa",
"signal":"helloFromVide"
}
}
	{
	"command":"VIDEO_CALL_START",
	"data":{
	"idUser":"5ffc8765a11fd6689",
	"idChat":"5fff461dcf5368340",
	"userName":"AndrewZ",
	"chatName":"GroupChatName",
	"signal":"helloFromVide"
	}
	}

{
"command":"VIDEO_CALL_JOIN",
"data":{
"idUser":"5ffc87c6a11fd",
"idChat":"5fff461dcf536",
"userName":"ValekZ",
"chatName":"GroupChatNa",
"signal":"helloFromVide"
}
}
	{
	"command":"VIDEO_CALL_JOIN",
	"data":{
	"idUser":"5ffc87c6a11fd6689",
	"idChat":"5fff461dcf5368340",
	"userName":"ValekZ",
	"chatName":"GroupChatName",
	"signal":"helloFromVideoCal"
	}
	}

{
"command":"VIDEO_CALL_EXIT",
"data":{
"idUser":"5ffc8765a11fd",
"idChat":"5fff461dcf5368"
}
}
	{
	"command":"VIDEO_CALL_EXIT",
	" data":{
	"idUser":"5ffc8765a11fd6689",
	"idChat":"5fff461dcf536834"
	}
	}

{
"command":"FILE_SEND",
"data":{
"userId":"6006f98b42e4394a368080cc",
"chatId":"601294a5afd04c5caeb9f356",
"title":"ttt",
"bytes":[52, 53, 52, 53, 52]
}
}
	{
	"command":"FILE_SEND",
	"data":{
	"userId":"6006c07a9da711035eec5958",
	"chatId":"6007fe99416f8f740825a651",
	"fileId":"600814d6f4170d1c216c2245",
	"title":"Hello World!",
	"bytes":"NDU0NTQ="
	}
	}

{
"command":"FILE_DELETE",
"data":{
"userId":"6006c07a9da711035eec5958",
"chatId":"6007fe99416f8f740825a651",
"fileId":"600814d6f4170d1c216c2245"
}
}
	{
	"command":"FILE_DELETE",
	"data":{
	"userId":"6006f98b42e4394a368080cc",
	"chatId":"6012b685731c903dbcb6228f",
	"fileId":"6012b8ba1073d5598388f1e2"
	}
	}

ДОДАТОК Б

Приклади успішних команд SSE

Endpoint	Type	Request	Response	SSE

/room/getRoom/{roomId}	Get		{
			"id":"601024f882a0e326a46d9d78",
			"roomName":"brookRoom",
			"creatorUser":"brook",
			"users":[
			{
			"userName":"brook",
			"userId":"123421",
			"signal":"signal"
			}
			]
			}	

/room/getRooms	Get		[
			{
			"roomId":"6010232d82a0e326a46d9d77",
			"nameRoom":"brookChat"
			},
			{
			"roomId":"601024f882a0e326a46d9d78",
			"nameRoom":"brookRoom"
			}
			]	

/zipli/room/createRoom/	Post	{
		"userName":"brook",
		"roomName":"brookRoom"
		}
			{
			"id":"6010232d82a0e326a46d9d77",
			"roomName":"brookRoom",
			"creatorUserName":"brook",
			"usersInfo":[],
			"messages":[]
			}	

/zipli/room/subscribeMessage/{roomId}	Get			

/zipli/room/joinRoom/{roomId}	Post	{
		"userName":"brook",
		"userId":"123421",
		"signal":"signal"
		}
			{
			"id":"601024f882a0e326a46d9d78",
			"roomName":"brookRoom",
			"creatorUser":"brook",
			"users":[
			{
			"userName":"brook",
			"userId":"123421",
			"signal":"signal"
			}
			]
			}
				Id: 32241124452124
				event: JOIN_ROOM_EVENT
				data:{
				"signal": "signal",
				"userName":"Brook"
				}

/zipli/room/leaveRoom/{roomId}	Post	{
		"userName":"brook",
		"userId":"123421",
		"signal":"signal"
		}
			{
			"id":"601024f882a0e326a46d9d78",
			"roomName":"brookRoom",
			"creatorUser":"brook",
			"users":[]
			}
				id: 1221213232
				event: LEAVE_ROOM_EVENT
				data:{
				"signal":"signal",
				"userName":"brook"
				}

/zipli/room/deleteRoom/{roomId}	Post		{
			"report":"Ok"
			}
				id:272781678862
				event:
				DELETE_ROOM_EVENT

/zipli/room/newMessage/{roomId}	Post	{
		"userName":"brook",
		"textMessage":"hey"
		}
			{
			"authorUserName":"brook",
			"textMessage":"hey",
			"date":1611671769043
			}
				id:22124242142122
				event:NEW_MESSAGE_EVENT
				data:{
				“userName":"brook",
				"textMessage":"hey”
				}

/zipli/room/getMessages/{roomId}	Get		[
			{
			"authorUserName":"brook",
			"textMessage":"hey",
			"date":1611671769043
			}
			]	

ДОДАТОК В

Статус коди помилок REST

Response status codes if request contains bad syntax or cannot be fulfilled (4xx)

Error Status Code	Error	Endpoint
11	USER_ID_NULL	/zipli/myAccount/getUser/{userId}
		/zipli/myAccount/delete-avatar/{userId}
		/zipli/myAccount/delete-account/{userI
12	USER_ID_DOES_NOT_CORRECT	/zipli/myAccount/getUser/{userId}
		/zipli/myAccount/delete-avatar/{userId}
		/zipli/myAccount/delete-account/{userId}
		/zipli/myAccount/update-avatar
		/zipli/myAccount/update-nickName
		/zipli/myAccount/update-email
		/zipli/myAccount/update-password
13	DATA_IS_NULL	/zipli/myAccount/update-password
		/zipli/myAccount/update-email
		/zipli/myAccount/update-nickName
		/zipli/myAccount/update-avatar
14	EMAIL_DOES_NOT_CORRECT	/zipli/myAccount/update-email
		/zipli/auth/forgot_password
15	USER_DOES_NOT_EXIST	/zipli/auth/forgot_password
		/zipli/auth/reset_password
		/zipli/auth/signin
16	USER_DOES_NOT_PASS_EMAIL_CONFIR	/zipli/auth/signin
17	EMAIL_ALREADY_EXISTS	/zipli/auth/signup
		/zipli/myAccount/update-email
18	PASSWORD_INCORRECT	/zipli/auth/signin
19	TOKEN_INVALID_OR_BROKEN	/zipli/auth/confirm-mail
20	PASSWORD_IS_NULL	/zipli/auth/reset_password

ДОДАТОК Г

Статус коди помилок WS

Error Status Code	Error
21	USERS_DOES_NOT_EXIST
22	CHAT_ACCESS_ERROR
23	MESSAGE_ACCESS_ERROR
24	CHAT_ALREADY_EXISTS
25	CHAT_NOT_EXISTS
26	MESSAGE_NOT_EXISTS
27	UNEXPECTED_EXCEPTION
28	VIDEO_CALL_EXCEPTION
29	FILE_ACCESS_ERROR
30	GRIDFSFILE_IS_NOT_FOUND
31	FILE_WAS_NOT_LOADING_CORRECT
32	FILE_IS_NOT_IN_A_DB
33	INCORRECT_REQUEST
34	ROOM_NOT_EXIT
35	ROOM_ALREADY_EXISTS
36	PARAM_IS_NULL
37	USERS_DOES_NOT_EXIST_BY_PARA
38	PARAM_TOO_SHORT