**4 Электронная почта**

**4.1 Подтверждение электронной почты**

Электронная почта участвует в регистрации аккаунтов и страниц практически в любых сервисах, социальных сетях, приложениях и так далее. В большинстве случаев автоматическая система просит пользователя подтвердить то, что именно он владелец почтового ящика.

Процедура подтверждения почтового ящика присутствует практически на всех сайтах в обязательной или опциональной форме. У результата есть несколько полезных последствий:

* привязанный адрес позволит восстановить доступ к аккаунту в случае взлома или утери пароля;
* на почту приходят уведомления о новостях, связанных с сервисом, вашим аккаунтом;
* при попытке взлома или авторизации вы получите уведомление по почте;
* при авторизации с чужих или новых устройств сможете подтвердить статус владельца с помощью ссылки в письме на подтвержденной почте.

## 4.2 Принадлежность почты к аккаунту

Чаще всего пользователи сталкиваются с данной операцией или сразу после регистрации аккаунта, или при первой авторизации на странице/сервисе. В редких случаях вы должны самостоятельно привязывать Email через настройки аккаунта или приложения.

Сообщение со ссылкой или кодом для подтверждения отправляется на ваш почтовый ящик после ввода логина, пароля и прочих сведений, необходимых для регистрации.

**JavaMail** — это API , который используется для составления, написания и чтения электронных сообщений (электронной почты).

Средство JavaMail можно применять ко многим событиям. Его можно использовать во время регистрации пользователя (отправка уведомлений, таких как благодарность за ваш интерес к моему сайту), забытый пароль (отправка пароля на идентификатор электронной почты пользователя), отправка уведомлений о важных обновлениях и т. д. Таким образом, может быть различное использование API почты Java.

### Протоколы, используемые в JavaMail API. Есть несколько протоколов, которые используются в JavaMail API:

* SMTP;
* POP;
* IMAP;
* MIME;
* NNTP и др.

**SMTP** (Simple Mail Transfer Protocol) — протокол передачи почты. Он был представлен еще в 1982 году, но не теряет актуальности до сих пор.

У протокола две главные задачи:

1. Проверка корректности настроек системы и предоставление «разрешения» на отправку email-сообщения для определенного устройства.
2. Отправка исходящего сообщения на заданный адрес электронной почты и подтверждение успешной доставки. Если сообщение доставить не удается, отправитель получает соответствующее извещение.

Теоретически SMTP умеет работать с практически любыми протоколами так называемого транспортного уровня, включая TCP, UDP и другие. Еще на заре развития протокола за ним закрепили два номера порта:

1. Первый — это порт 25, посредством которого почта передается между почтовыми серверами.
2. Второй — порт 587, благодаря которому почта передается от почтового клиента на сервер.

В большинстве случаев протокол SMTP используется для передачи исходящей почты с использованием порта TCP 25. То есть можно сказать, что SMTP-порт — это как раз TCP 25, хоть и не всегда. Иногда задействуется еще порт 465. Так происходит, когда порт требует защищенного SSL-соединения.

Но в большинстве случаев используется лишь один транспортный протокол TCP с портом 25 (это SMTP-порт по умолчанию). Другие варианты применяются крайне редко, например, когда провайдеры по какой-то причине закрывают доступ к 25 порту. Они могут делать это, например, для блокировки спам-рассылок.

Maven:

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-mail</artifactId>

</dependency>

**4.3 Авторизация через почтовый сервер**

Для отправки писем с токеном верификации пользователя, требуется сконфигурировать JavaMailSender.

В качестве почтового сервиса в лабораторной работе используется mail.ru

* в папке resources создаем файл mail.properties;
* описываем в него следующие свойства:

mail.host=smtp.mail.ru

mail.port=465

[mail.username=some-username@mail.ru](mailto:mail.username%3Dsome-username@mail.ru)

mail.password=some-password

mail.protocol=smtps

mail.smtps.auth=true

где some-username — логин почтового ящика для рассылки писем, some-password — пароля от этого почтового ящика, соответственно.

Для использования почтового ящика для отправки писем из приложения, требуется пароль приложения, сгенерированный почтовым сервисом.

Конфигурация отправки письма:

@Configuration

@PropertySource("classpath:mail.properties")

public class MailConfig {

@Value("${mail.username}")

private String username;

@Value("${mail.password}")

private String password;

@Value("${mail.host}")

private String host;

@Value("${mail.port}")

private String port;

@Value("${mail.protocol}")

private String protocol;

@Value("${mail.smtps.auth}")

private String smtpsAuth;

@Bean

public JavaMailSender getJavaMailSender() {

JavaMailSenderImpl mailSender = configureMailSender();

mailSender.setJavaMailProperties(properties(mailSender));

return mailSender;

}

private JavaMailSenderImpl configureMailSender(){

JavaMailSenderImpl mailSender = new JavaMailSenderImpl();

mailSender.setHost(host);

mailSender.setPort(Integer.parseInt(port));

mailSender.setProtocol(protocol);

mailSender.setUsername(username);

mailSender.setPassword(password);

return mailSender;

}

private Properties properties(JavaMailSenderImpl mailSender){

Properties props = mailSender.getJavaMailProperties();

props.put("mail.transport.protocol", protocol);

props.put("mail.smtp.auth", smtpsAuth);

props.put("mail.smtp.starttls.enable", "true");

props.put("mail.debug", "true");

return props;

}

}

**@PropertySource** — аннотация, описывающая источник свойств для их дальнейшего внедрения с помощью аннотации @Value.

**4.4 Верификация пользователя**

Сущность верификации:

@Getter

@Setter

@ToString

@Entity(name = "verificationToken")

@Table(name = "verification\_tokens")

public class VerificationToken {

@Id

@GeneratedValue(strategy = GenerationType.AUTO)

private UUID id;

private String token;

@OneToOne(targetEntity = User.class, fetch = FetchType.EAGER)

@JoinColumn(nullable = false, name = "user\_id")

private User user;

private Date expiryDate;

private boolean confirmed;

}

* описывается идентификатор, значения генерируемоего токена, пользователя, который проходит верификацию по этому токену, дату истечения срока токена, состояние подтверждения токена;
* после истечения указанного срока годности, токен верификации должен быть удалён, поскольку пользователь не подтвердил указанный почтовый ящик.

Скрипт миграции:

CREATE TABLE if not exists verification\_tokens

(

id UUID NOT NULL,

token VARCHAR(255),

user\_id UUID NOT NULL,

expiry\_date TIMESTAMP WITHOUT TIME ZONE,

confirmed BOOLEAN NOT NULL,

CONSTRAINT pk\_verification\_tokens PRIMARY KEY (id)

);

ALTER TABLE verification\_tokens

ADD CONSTRAINT FK\_VERIFICATION\_TOKENS\_ON\_USER FOREIGN KEY (user\_id) REFERENCES users (id) on delete cascade;

DAO:

public interface VerificationTokenRepository extends JpaRepository<VerificationToken, UUID> {

Optional<VerificationToken> findVerificationTokenByUserId(UUID id);

}

CRUD:

@Service

public class VerificationTokenCrudService {

private final VerificationTokenRepository verificationTokenRepository;

@Autowired

public VerificationTokenCrudService(VerificationTokenRepository verificationTokenRepository) {

this.verificationTokenRepository = verificationTokenRepository;

}

public VerificationToken getByUserId(UUID id){

return verificationTokenRepository

.findVerificationTokenByUserId(id)

.orElseThrow(

()-> new EntityExistsException(

String.format("Verification token with user id: %s doesn't exists", id)));

}

public VerificationToken create(VerificationToken verificationToken){

if( verificationToken.getId() != null

&& verificationTokenRepository.existsById(verificationToken.getId())){

throw new EntityExistsException(

"Verification token with id:'"

+ verificationToken.getId()

+ "' already exists");

}

verificationToken.setExpiryDate(expirationDate());

return verificationTokenRepository.save(verificationToken);

}

private Date expirationDate(){

return new Date(new Date().getTime() + 24\*60\*60\*1000);

}

public VerificationToken merge(VerificationToken verificationToken){

return verificationTokenRepository.save(verificationToken);

}

public VerificationToken getById(UUID id){

return verificationTokenRepository.findById(id).orElseThrow(()->{

throw new EntityExistsException(String.format("Token with id: %s doesn't exists", id));

});

}

public Date getExpirationDate(UUID id){

return getById(id).getExpiryDate();

}

}

Сервис верификации:

public interface IVerificationService {

void confirm(String token);

void createVerificationToken(UUID userId);

boolean shouldRemove(UUID id);

}

@Service

public class VerificationService implements IVerificationService {

private final VerificationTokenCrudService verificationTokenCrudService;

private final UserService userService;

@Autowired

public VerificationService(VerificationTokenCrudService verificationTokenCrudService, UserService userService) {

this.verificationTokenCrudService = verificationTokenCrudService;

this.userService = userService;

}

@Override

public void confirm(String token) {

String username = SecurityContextHolder.getContext().getAuthentication().getName();

User user = userService.getByUsername(username);

VerificationToken verificationToken = verificationTokenCrudService.getByUserId(user.getId());

if(!token.equals(verificationToken.getToken())){

throw new IllegalArgumentException("Incorrect token for user: "+ user.getEmail());

}

verificationToken.setConfirmed(true);

verificationTokenCrudService.merge(verificationToken);

}

@Override

public void createVerificationToken(UUID userId) {

VerificationToken verificationToken = new VerificationToken();

verificationToken.setToken(createRandomTokenFromUUID());

verificationToken.setConfirmed(false);

User user = userService.getById(userId);

verificationToken.setUser(user);

verificationTokenCrudService.create(verificationToken);

}

@Override

public boolean shouldRemove(UUID id) {

return !isNotExpired(verificationTokenCrudService.getById(id).getUser().getEmail()) &&

!isConfirmed(verificationTokenCrudService.getById(id).getUser().getId());

}

public boolean shouldRemove(String username){

return shouldRemove(verificationTokenCrudService.getByUserId(userService.getByUsername(username).getId()).getId());

}

public String getVerificationTokenValueByUserId(UUID userId){

return verificationTokenCrudService.getByUserId(userId).getToken();

}

public boolean isConfirmed(UUID userId){

return verificationTokenCrudService.getByUserId(userId).isConfirmed();

}

public boolean isConfirmed(String username){

return isConfirmed(userService.getByUsername(username).getId());

}

public boolean isNotExpired(String username) {

VerificationToken verificationToken = verificationTokenCrudService

.getByUserId(

userService.getByUsername(username).getId()

);

return !verificationToken.getExpiryDate().before(new Date());

}

private String createRandomTokenFromUUID(){

return UUID.randomUUID().toString();

}

}

**4.5 События**

Spring позволяет создавать и публиковать пользовательские события, которые по умолчанию являются синхронными. Это имеет несколько преимуществ, таких как возможность прослушивателя участвовать в контексте транзакции издателя.

Чтобы опубликовать событие, издатель может просто внедрить ApplicationEventPublisher и использовать API publishEvent()

Мы будем работать над регистрацией и авторизацией пользователя, публикуя OnAuthorizationEvent/OnRegistrationEvent всякий раз, когда пользователь регистрируется или авторизуется в нашем приложении.

Мы позволим всем слушателям обрабатывать это событие и выполнять соответствующие действия (например, отправлять электронные письма при регистрации клиентов).

Авторизация:

@Getter

@Setter

public class OnAuthorizationEvent {

private AuthDto authDto;

public OnAuthorizationEvent(AuthDto authDto){

this.authDto = authDto;

}

}

Регистрация:

@Getter

@Setter

public class OnRegistrationEvent {

private RegDto regDto;

public OnRegistrationEvent(RegDto regDto){

this.regDto = regDto;

}

}

**4.6 Слушатели**

В рамках приложения нужен прослушиватель, который должен прослушивать публикуемое событие и действовать соответствующим образом, если событие, опубликованное службой, представляет интерес.

При публикации события OnAuthorizationEvent обработкой будет заниматься слушатель AuthorizationListener. Обработкой события OnRegistrationEvent, соответственно, будет заниматься OnRegistrationEvent.

Авторизация:

@Component

public class AuthorizationListener {

private final VerificationService verificationService;

private final UserCrudService userCrudService;

public AuthorizationListener(UserCrudService userCrudService,

VerificationService verificationService) {

this.userCrudService = userCrudService;

this.verificationService = verificationService;

}

@EventListener

@Transactional

public void onApplicationEvent(OnAuthorizationEvent event) {

checkConfirmed(event);

}

private void checkConfirmed(OnAuthorizationEvent event){

if(verificationService.shouldRemove(event.getAuthDto().getEmail())){

userCrudService.removeByUsername(event.getAuthDto().getEmail());

}

}

}

Регистрация с подтверждением почты:

@PropertySource("classpath:mail.properties")

@Component

public class RegistrationWithMailListener {

@Value("${mail.username}")

private String from;

private final JavaMailSender javaMailSender;

private final UserRegService userRegService;

private final VerificationService verificationService;

public RegistrationWithMailListener(JavaMailSender javaMailSender,

UserRegService userRegService,

VerificationService verificationService) {

this.javaMailSender = javaMailSender;

this.userRegService = userRegService;

this.verificationService = verificationService;

}

//@Transactional - аннотация на уровне и класса метода, которая позволяет отменить

//внесенные изменения базы данных в случае возникновения ошибки  
 @Transactional

@SneakyThrows

@EventListener

public void onApplicationEvent(OnRegistrationEvent event) {

verificationService.createVerificationToken(event.getRegDto().getId());

String verTokenValue = verificationService.getVerificationTokenValueByUserId(event.getRegDto().getId());

sendMail(verTokenValue, event);

}

private void sendMail(String verificationTokenValue, OnRegistrationEvent event){

String subject = "Registration Confirmation";

String message = "Registration Success";

SimpleMailMessage simpleMailMessage = new SimpleMailMessage();

simpleMailMessage.setTo(event.getRegDto().getEmail());

simpleMailMessage.setSubject(subject);

simpleMailMessage.setText(message + "\nconfirmation token: " + verificationTokenValue);

simpleMailMessage.setFrom(from);

javaMailSender.send(simpleMailMessage);

}

}

**4.7 Фильтр верификации**

Помимо фильтрации запросов на предмет наличия токена авторизации, введем фильтр для проверки пользователя на верификацию по почте. Т.е. подтвержден ли пользователь с помощью токена верификации отправленный ему на указанный email при регистрации.

**UserVerificationFilter** — фильтрует запросы по токену верификации для всех запросов, кроме запроса подтверждения токена.

@Component

public class UserVerificationFilter extends OncePerRequestFilter {

private final VerificationService verificationService;

private final UserCrudService userCrudService;

@Autowired

public UserVerificationFilter(VerificationService verificationService,

UserService userCrudService) {

this.verificationService = verificationService;

this.userCrudService = userCrudService;

}

@Override

protected boolean shouldNotFilter(HttpServletRequest request) {

return request.getRequestURI().equals("/auth/reg/mail/confirm");

}

@Override

protected void doFilterInternal(HttpServletRequest request,

HttpServletResponse response,

FilterChain filterChain) throws ServletException, IOException {

Authentication authentication = SecurityContextHolder.getContext().getAuthentication();

logger.info("auth = " + authentication);

boolean shouldRemoveUser = false;

String username = null;

if(authentication != null){

username = authentication.getName();

if(!verificationService.isConfirmed(username)){

if(!verificationService.isNotExpired(username)){

shouldRemoveUser = true;

}

response.sendError(HttpServletResponse.SC\_FORBIDDEN, "UNCONFIRMED ACCOUNT");

}

}

filterChain.doFilter(request, response);

if(shouldRemoveUser){

userCrudService.removeByUsername(username);

}

}

}

**ConfirmFilter** — фильтрует запросы по маршруту предназначенному для подтверждения токена верификации.

@Component

public class ConfirmFilter extends OncePerRequestFilter {

private final VerificationService verificationService;

private final UserCrudService userCrudService;

@Autowired

public ConfirmFilter(VerificationService verificationService, UserCrudService userCrudService) {

this.verificationService = verificationService;

this.userCrudService = userCrudService;

}

@Override

protected boolean shouldNotFilter(HttpServletRequest request) {

return !request.getRequestURI().equals("/auth/reg/mail/confirm");

}

@Override

protected void doFilterInternal(HttpServletRequest request,

HttpServletResponse response,

FilterChain filterChain) throws ServletException, IOException {

String username = SecurityContextHolder.getContext().getAuthentication().getName();

boolean shouldRemove = verificationService.shouldRemove(username);

if (shouldRemove) {

response.sendError(HttpServletResponse.SC\_FORBIDDEN, "TOKEN EXPIRED");

}

filterChain.doFilter(request, response);

if (shouldRemove) {

userCrudService.removeByUsername(username);

}

}

}

**4.8 Изменения**

WebSecurityConfig:

* добавляем маршрут для регистрации через почту в список доступных для всех маршрутов, поскольку регистрацию с подтверждением почты может использовать любой пользователь, в т.ч. анонимный;
* добавляем новые фильтры для проверки верификации пользователя при попытке совершить запрос:

@Configuration

public class WebSecurityConfig {

private final UserDetailsService userDetailsService;

private final JwtAuthenticationEntryPoint jwtAuthenticationEntryPoint;

private final JwtRequestFilter jwtRequestFilter;

private final UserVerificationFilter userVerificationFilter;

private final ConfirmFilter confirmFilter;

@Autowired

public WebSecurityConfig(UserDetailsService userDetailsService,

JwtAuthenticationEntryPoint jwtAuthenticationEntryPoint,

JwtRequestFilter jwtRequestFilter,

UserVerificationFilter userVerificationFilter, ConfirmFilter confirmFilter) {

this.userDetailsService = userDetailsService;

this.jwtAuthenticationEntryPoint = jwtAuthenticationEntryPoint;

this.jwtRequestFilter = jwtRequestFilter;

this.userVerificationFilter = userVerificationFilter;

this.confirmFilter = confirmFilter;

}

@Bean

SecurityFilterChain web(HttpSecurity http,

SuccessLogoutHandlerImpl successLogoutHandler) throws Exception {

http

.csrf()

.disable()

.authorizeHttpRequests(

(authorize) -> authorize

.antMatchers(

"/auth/login",

"/auth/reg/",

"/auth/reg/mail")

.permitAll()

.anyRequest().authenticated()

)

.logout(logout ->

Logout

.logoutUrl("/auth/logout")

.deleteCookies("JSESSION")

.invalidateHttpSession(true)

.logoutSuccessHandler(successLogoutHandler)

)

.userDetailsService(userDetailsService)

.exceptionHandling()

.authenticationEntryPoint(jwtAuthenticationEntryPoint)

.and()

.sessionManagement()

.sessionCreationPolicy(SessionCreationPolicy.STATELESS)

.and()

.addFilterAfter(userVerificationFilter, UsernamePasswordAuthenticationFilter.class)

.addFilterAfter(confirmFilter, UsernamePasswordAuthenticationFilter.class)

.addFilterBefore(jwtRequestFilter, UsernamePasswordAuthenticationFilter.class);

return http.build();

}

@Bean

AuthenticationManager authenticationManagerBean(AuthenticationConfiguration authenticationConfiguration)

throws Exception {

return authenticationConfiguration.getAuthenticationManager();

}

}

AuthRestController:

* добавление публикации события авторизации;
* добавление публикации события регистрации;
* ApplicationEventPublisher - интерфейс, который инкапсулирует функциональность публикации событий:

@RestController

@RequestMapping("/auth")

public class AuthRestController {

private final UserAuthService userAuthService;

private final UserRegService userRegService;

private final ApplicationEventPublisher applicationEventPublisher;

private final VerificationService verificationService;

public AuthRestController(UserAuthService userAuthService,

UserRegService userRegService,

ApplicationEventPublisher applicationEventPublisher, VerificationService verificationService) {

this.userAuthService = userAuthService;

this.userRegService = userRegService;

this.applicationEventPublisher = applicationEventPublisher;

this.verificationService = verificationService;

}

@PostMapping(path = "/login")

@RequestMapping(value = "/login")

public Token authorization(@RequestBody @Valid AuthDto authDto) throws Exception {

applicationEventPublisher

.publishEvent(new OnAuthorizationEvent(authDto));

return userAuthService.authorization(authDto);

}

@PostMapping

@RequestMapping("/reg")

public Token registration(@RequestBody @Valid RegDto user) throws Exception {

Token token = userRegService.registration(user);

user.setId(token.getUserId());

applicationEventPublisher.publishEvent(new OnRegistrationEvent(user));

return token;

}

@PostMapping("/confirm")

public void confirm(@RequestParam("token") String token){

verificationService.confirm(token);

}

}

**4.9 Пересоздание внешнего ключа сущности токен**

Чтобы корректно удалить пользователя после истечения срока верификации, требуется каскадное удаление зависимых сущностей (tokens, verification\_tokens). Для этого ко внешним ключам, связанным с сущностью users требуется добавить «on delete cascade».

* создаем новый скрипт миграции для сущность Token;
* сначала удаляем внешний ключ;
* затем он создается заново, однако теперь добавляется «on delete cascade»:

alter table tokens drop constraint tokens\_user\_id\_fkey;

alter table tokens add foreign key (user\_id) references users(id) on delete cascade;

Таким образом, после удаления пользователя, каскадно удалятся зависимые от него данные.

**4.10 Тестирование маршрутов**

Reg — маршрут для регистрации. Confirm — маршрут для подтверждения верификации (рис. 4.1).

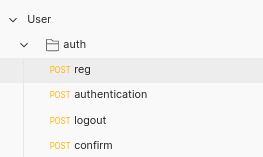


Рис. 4.1 — Расположение маршрутов регистрации и верификации в postman

Маршрут для регистрации (рис. 4.2) —http://localhost:8080/auth/reg/.

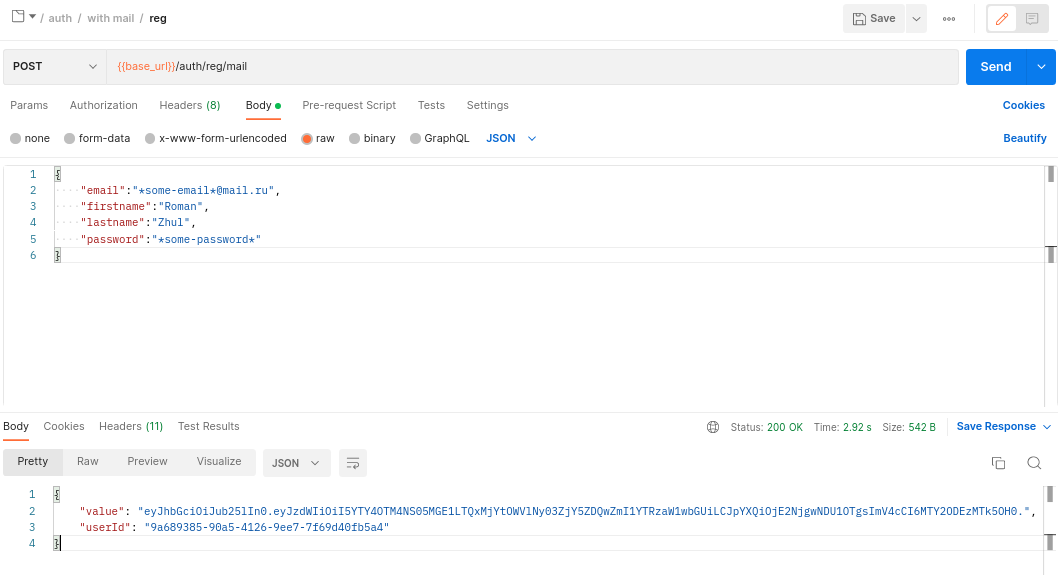


Рис. 4.2 - Пример запроса по маршруту регистрации в postman

Маршрут для подтверждения указанной почты — http://localhost:8080[/auth/confirm](http://localhost:8080/auth/reg/mail/confirm).

В запросе указывается параметр token, который содержит токен верификации, извлеченный из письма на почте (рис. 4.3).

В ответ приходит код 200, сообщающий об успешном верификации аккаунта (рис. 4.4).

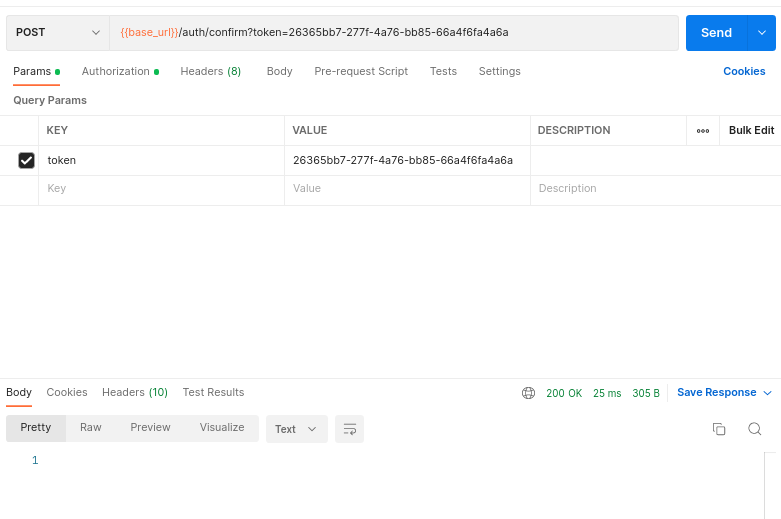


Рис. 4.3 - Пример подтверждения почты с помощью токена в postman

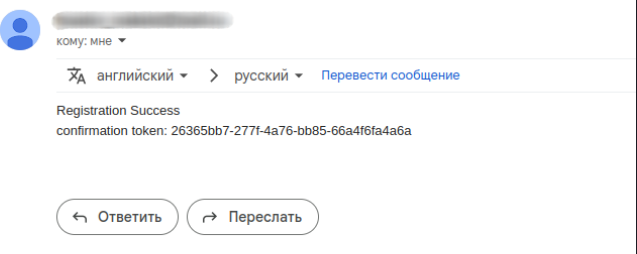


Рис. 4.4 - Код подтверждения поступивший в почтовый ящик

**4.11 Лабораторная работа №4 - Регистрация с подтверждением электронной почты**

Цель лабораторной работы: знакомство с верификацией пользователя в разрабатываемой системе путем подтверждения электронного адреса, а также знакомство с событиями внутри Spring Framework.

Задание:

1. Описать новую сущность для хранения состояния верификации пользователя.
2. Описать data access object (repository) для новой сущности.
3. Описать сервисные объекты.
4. Описать фильтрацию запросов для регулирования доступа к ресурсам при различных состояниях верификации пользователя (ConfirmFilter, UserVerificationFilter).
5. Описать события авторизации/регистрации и их слушатели.
6. Внести изменения в контроллер AuthRestController и класс конфигурации WebSecurityConfig.
7. Описать конфигурации отправки писем на почту.

Контрольные вопросы:

Вопросы к защите лабораторной работы включают процесс установки и настройки по каждому пункту задания, а также теоретический материал к данной работе.